Lab Assignment 2
C Parallel Matrix-Vector Multiplication
An Huynh

Introduction

The goal of this assignment was to utilize Open MPI to parallelize our matrix vector dot product code from our lab assignment 1. In addition, our code needs to be able to run on very large matrices.

Design and Implementation

Most of the code is the same from LA1 with the addition of a subMatrixVectorProduct function to handle the dot product for each sub matrix and sub vector and MPI code in the main function.

1. **double* subMatrixVectorProduct(double* mat, double* vec, int n, int rowAmount)**

```
// Used for performing matrix vector product for each sub matrix and vector
double* subMatrixVectorProduct(double* mat, double* vec, int n, int rowAmount){
    double *result = (double*) malloc(rowAmount * sizeof(double));
    // Keep track of what "row" we're on for mat
    int totalNumOfElements = n * rowAmount;

    // Manually updating vecIndex - for vec and result
    int vecIndex = 0;
    for(int i = 0; i < totalNumOfElements; i+=n){
        double sum = 0.0;
        for(int j = i; j < (i + n); j++){
            sum += mat[j] * vec[vecIndex];
        }
        result[vecIndex] = sum;
        vecIndex = vecIndex + 1;
    }

    return result;
}
```

This method is the same as double* mvp1(double* mat, double* vec, int n) from LA1 with an additional parameter of type int, rowAmount. The purpose of this function is to perform the dot product for each of the sub matrices and vectors in each of the ranks for when we parallelize our code. The parameter rowAmount will help us keep track of how large each of the sub matrices and vectors are.

2.1. **Parallel Matrix Vector Multiplication Code Part 1**

```
//Main function - Where we parallelize matrix vector multiplication
int main(int argc, char **argv){
    // Get dimension of square matrix and vector from argv (In terminal:$ /_.x _pas
    if(argc == 1){
        printf("Please input an int for dimensions as parameter.\n");
    }

    // Need to convert argv[1] to an int since argv takes in chars
    int n = atoi(argv[1]);

    // Checking parameter
    if(n <= 0){
        printf("Please enter a positive integer greater than 0 as parameter.\n");
    }

    // Set up the vector and matrix
    double *vec2 = allocVec(n);
    double **mat2 = allocMat2(n);
    assignVec(vec2, n);
    assignMat2(mat2, n);

    // Initialize MPI
    MPI_Init(&argc, &argv);

    int numRanks, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Seeing if amount of rank is not equally divisible by dimension - class examp
    if(n % numRanks != 0){
        if(rank == 0){
            printf("Need another N.\n");
        }
        MPI_Finalize();
        return 0;
    }
```

2.2. **Parallel Matrix Vector Multiplication Code Part 2**

```
// Splitting up the matrix into seperate "sub matrices" for each rank
    int rowAmountPerRank = n / numRanks;
    double *subMatrix = (double*) malloc(n*rowAmountPerRank*sizeof(double));
    double* result = (double*) malloc(n * sizeof(double));
    double* matrixAs1DArr;

    // Start time
    double startTime = MPI_Wtime();
    if (rank == 0) {
        matrixAs1DArr = (double*) malloc(n*n*sizeof(double));
        // Convert 2d matrix to 1d matrx
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                matrixAs1DArr[i * n + j] = mat2[i][j];
            }
        }
        double endTimeForConvertingMat = MPI_Wtime();
        printf("Rank %d Converting time %f\n", rank, endTimeForConvertingMat - start
    }

    MPI_Scatter(matrixAs1DArr, rowAmountPerRank * n, MPI_DOUBLE, subMatrix, rowAmou

    double startTimeForCalculatingSubResult = MPI_Wtime();

    double* subResult = subMatrixVectorProduct(subMatrix, vec2, n, rowAmountPerRank
    double endTimeForCalculatingSubResult = MPI_Wtime();
    printf("Rank %d Sub result cal %f\n", rank, endTimeForCalculatingSubResult - st

    MPI_Gather(subResult, rowAmountPerRank, MPI_DOUBLE, result, rowAmountPerRank, M
    double endFullTime = MPI_Wtime();
    if (rank == 0) {
        printf("Full Time: %f\n", endFullTime - startTime);
    }

    MPI_Finalize();

    // Free memory
    freeMat2(mat2, vec2, result, n);

    return 0;
}
```

The main function here has the same structure as it did in LA1.

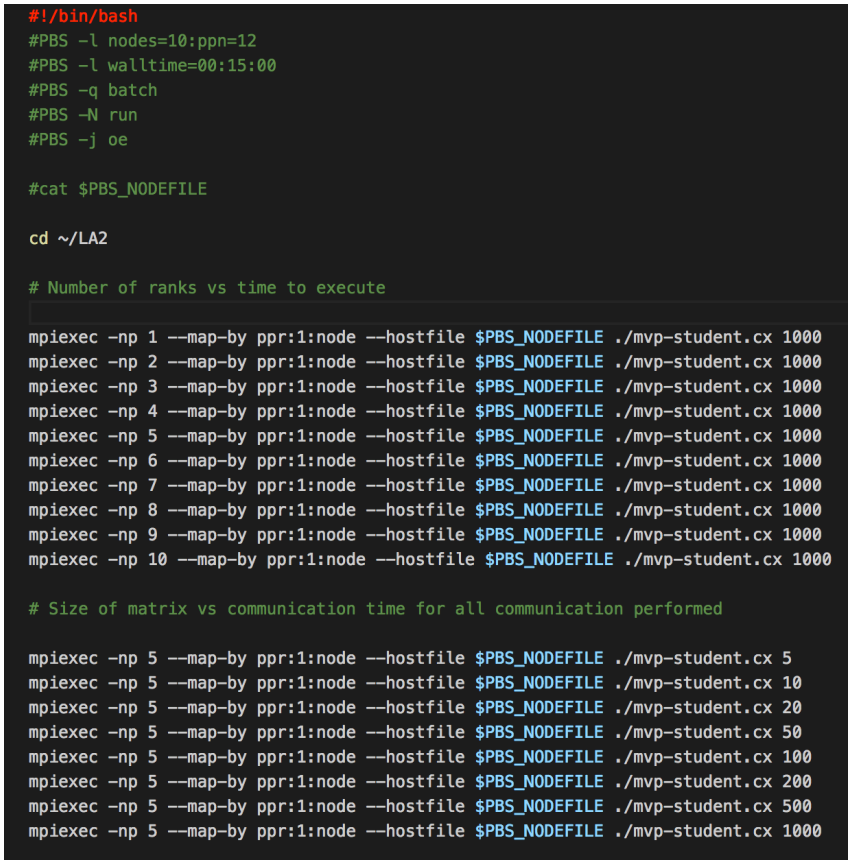
- Get dimension of square matrix and vector from argv. Check if it's a positive integer.
- Allocate space and assign values for matrix and vector using the double* allocVec(int n), double** allocMat2(int n), void assignVec(double* vec, int n), and void assignMat2(double** mat, int n) functions.
- Perform the matrix vector dot product.
- Print out results.
- Free allocated memory.

However, since we're running this application on more than one rank, this time we added the additional steps of dividing the matrix among each of the ranks and then getting the final result by collecting the "sub results" from all of the ranks. We did this by using a variable called rowAmountPerRank to keep track of how many rows each sub matrix will get. To make the process of sharing the original matrix easier, we convert it to a 1-D array and store it within matrixAs1DArr. We then set up two additional arrays subMatrix and subResult, where we will store the sub matrix and result for the dot product evaluation by each of the ranks. In order to send the needed data to each of the rank, we use MPI Scatter(matrixAs1DArr, rowAmountPerRank * n, MPI DOUBLE, subMatrix, rowAmountPerRank * n, MPI DOUBLE, 0, MPI COMM WORLD). This essentially sends the sub matrix to each of the ranks. Each rank will use the subMatrixVectorProduct function to perform the dot product between subMatrix and the vector. Finally, we use MPI Gather(subResult, rowAmountPerRank, MPI DOUBLE, result, rowAmountPerRank, MPI DOUBLE, 0, MPI COMM WORLD) to gather all of the subResults into the final result array.

Results

Below are some screen shots of my PBS script, results printed out in terminal, and graphs of the number of rank vs time it takes to execute my program and size of matrix vs communication time for all communication performed. Comparing the communication time to the time spent actually performing the calculations, the ratio appears to be 2/1. Using this, the Floating Point Operators per Second (FLOPS) that my application can achieve for the different numbers of ranks is around 15,931.68. To calculate FLOPS I used the equation:

$$2 * n^3/(Calculationtime)$$



```
#!/bin/bash
#PBS -l nodes=10:ppn=12
#PBS -l walltime=00:15:00
#PBS -q batch
#PBS -N run
#PBS -j oe

#cat $PBS_NODEFILE

cd ~/LA2

# Number of ranks vs time to execute

mpiexec -np 1 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 2 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 3 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 4 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 6 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 7 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 8 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 9 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
mpiexec -np 10 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000

# Size of matrix vs communication time for all communication performed

mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 5
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 10
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 20
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 50
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 100
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 200
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 500
mpiexec -np 5 --map-by ppr:1:node --hostfile $PBS_NODEFILE ./mvp-student.cx 1000
```

Figure 1: *Screen shot of my PBS script for this assignment.*

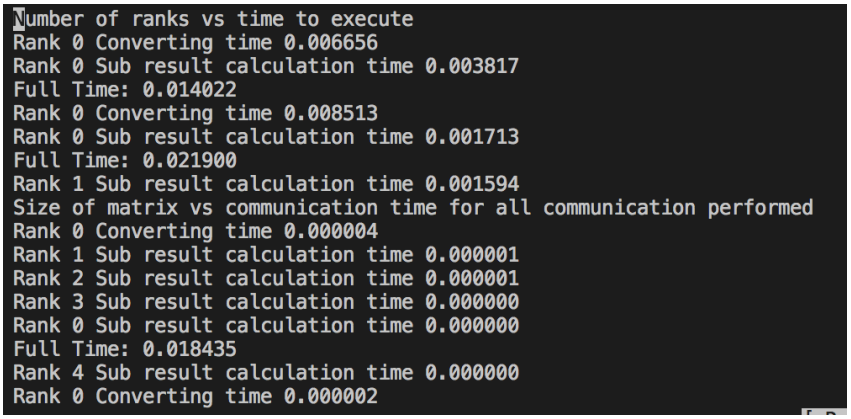


Figure 2: *Part of print output for my program*

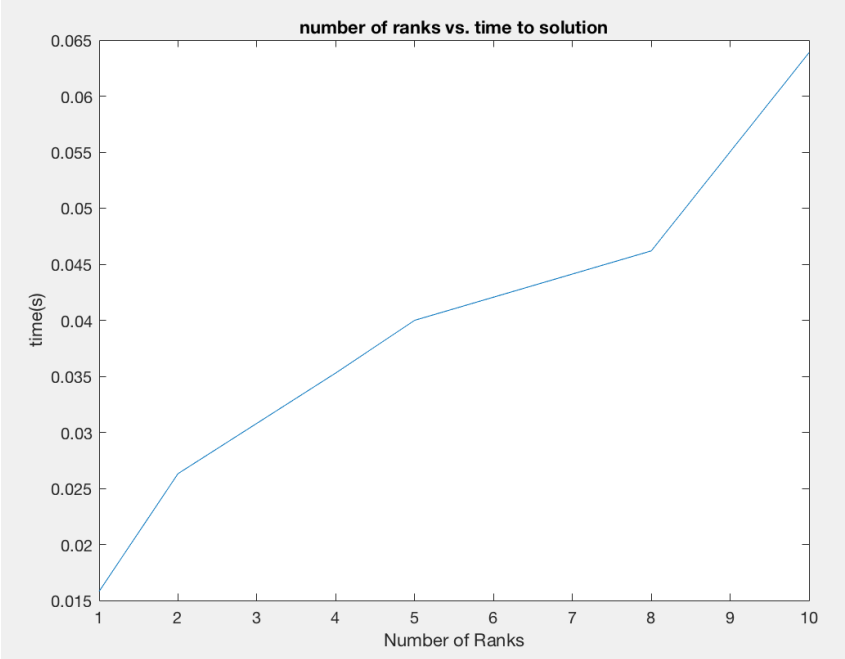


Figure 3: *Graph of number of rank vs time it takes to execute my program*

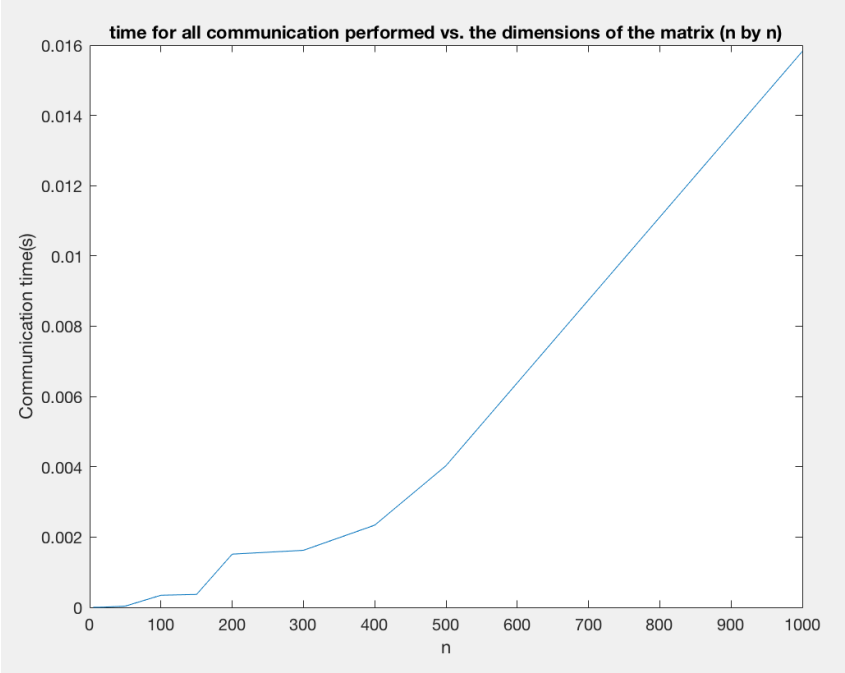


Figure 4: *Graph of size of matrix vs communication time for all communication performed*

Conclusion

In conclusion, my program outputs the expected results with any value for n (Dimension of matrix and vector). As noted by my graphs, the speed at which the application runs is relative to the size of the matrix and the amount of nodes being used.