Assignment: matrix-vector multiplication

Neeraj Sathyan (12938262)

February 14, 2020

1 Matrix-vector Multiplication Sequential code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 8  // Size of matrix and vector

#define R 100  // R iterations

int matrix[SIZE][SIZE], vector[SIZE], result[SIZE];

int main(int argc, char *argv[])
{
  int rank, size, matrix_start, matrix_end, n;
  n = SIZE;
  clock_t tic = clock();
        printf("Inserting Matrix : \n");
            for (int i=0; i<n*n; i++) {
                for (int j=0;j<n;++j)
                    matrix[i][j] = i+j;
            }
            for (int i=0; i<n; i++)
                vector[i] = i;

            for (int i=0;i<n;++i) {
                result[i] = 0;
            }

  int j=0; //Row no..
  for (int p=0; p<R; ++p) {
      for (int i=0; i<n; i++) {
          result[i] = 0;
        for (int j=0;j<n;++j) {
            result[i] += matrix[i][j] * vector[j];
        }
      }
  }
```

```
  tic = clock() - tic;

    printf("Matrix: \n");
    for(int i=0;i<n;++i) {
        for (int j=0; j< n; ++j) {
            printf("%d\t",matrix[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    printf("\nVector: \n");
    for(int i=0;i<n;++i){
        printf("%d\t",vector[i]);
    }
    printf("\n");
    printf("\nResult: \n");
    for(int i=0;i<n;++i){
        printf("%d\t",result[i]);
    }
    printf("\n");
    printf("Elapsed time: %f seconds\n",((double)tic) / CLOCKS_PER_SEC);

  return 0;
}
```

2 Matrix-vector multiplication MPI/OpenMP code

```
  #include <stdio.h>
  #include <stdlib.h>
  #include <mpi.h>
  #include <omp.h>

  #define SIZE 3  // Size of matrix and vector
  #define R 10 //Iterations
  //mtx_t lock;
  #define chunk 10

  int matrix[SIZE*SIZE], vector[SIZE], result[SIZE];

  int main(int argc, char *argv[])
  {

    int rank, size, matrix_start, matrix_end, n;
    double maxtime;
    n = SIZE;
    //MPI_Status status;
    omp_set_num_threads(4);  //Setting pre-defined threads

    MPI_Init (&argc, &argv);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);        /* who am i */
MPI_Comm_size(MPI_COMM_WORLD, &size); /* number of processors */
MPI_Barrier(MPI_COMM_WORLD);  /*synchronize all processes*/
double mytime = MPI_Wtime();  /*get time just before work section */
if (n%size != 0 || n<size) {
  printf("%d %d",n,size);
  if (rank==0)
    printf("Number of processors not divisible by matrix size.\n");
  MPI_Finalize();
  exit(-1);
}

if (rank==0) {
  printf("Inserting Matrix : \n");
          #pragma omp parallel for
          for (int i=0; i<n*n; i++) {
              matrix[i] = i;
          }

          #pragma omp parallel for
          for (int i=0; i<n; i++)
            vector[i] = i;
}

matrix_start = rank * (SIZE*SIZE)/size;
//Split up of rows, starting row index for processor rank
matrix_end = (rank+1) * (SIZE*SIZE)/size;
//Split up of rows, ending row index for processor rank

//Local variable to store the scatterd array sub sections..
int *procRow = malloc(sizeof(int) * n*n/size);
if (procRow == NULL) {
    printf("Error in malloc for procRow in proc: %d",rank);
    exit(1);
}
//Local variable storing the result within a processor..
int *procResult = malloc(sizeof(int) * n/size);
if (procResult == NULL) {
    printf("Error in malloc for procResult in proc: %d",rank);
}

#pragma omp parallel for
for (int i=0;i<n/size;++i) {
  procResult[i] = 0;
}

//Broadcast the vector to all size processors..
MPI_Bcast(vector, n, MPI_INT, 0, MPI_COMM_WORLD);

//Scatter the array with proportion of
```

```
// n/P rows where P is the numner of processors and n is the rows.
MPI_Scatter(matrix, n*n/size, MPI_INT, procRow,
n*n/size, MPI_INT, 0, MPI_COMM_WORLD);

//MPI_Barrier(MPI_COMM_WORLD);
printf("computing slice %d (from element %d to %d)\n"
, rank, matrix_start+1, matrix_end);
printf("\n");
    int i,j;
    for (int p =0; p<R; ++p) {
#pragma omp parallel shared(procRow,n,size,vector) private(i,j)
      {
        #pragma omp for schedule(static)
        for (i=0; i<n/size; i++) { // Rows..
          int res = 0;
          for (j=0;j<n;++j) {
            //res += procRow[v_index++]*vector[j];
            res += procRow[i*(n/size)+j]*vector[j];
          }
          procResult[i] = res;
        }
      }
    }


//Gather the results from each processor to the master processor to stitch
MPI_Gather(procResult, n/size, MPI_INT, result, n/size,
MPI_INT, 0, MPI_COMM_WORLD);

mytime = MPI_Wtime() - mytime;
/*get time just after work section*/

//Get the maximum execution time between the processors..
MPI_Reduce(&mytime, &maxtime, 1, MPI_DOUBLE,
MPI_MAX, 0, MPI_COMM_WORLD);

if (rank==0) {

  printf("Matrix: \n");
  for(int i=0;i<n*n;++i) {
      if(i%n == 0)
          printf("\n");
      printf("%d\t",matrix[i]);
  }

  printf("\nVector: \n");
  for(int i=0;i<n;++i){
      printf("%d\t",vector[i]);
  }
  printf("\n");
```

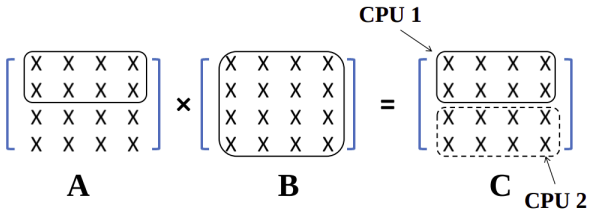

Figure 1: Architecture

```
    printf("\nResult: \n");
    for(int i=0;i<n;++i){
        printf("%d\t",result[i]);
    }
    printf("\n");
    printf("Running Time: %lf seconds\n",maxtime);

} MPI_Finalize(); return 0; }
```

3 Memory Layout and Algorithm

One of the ways of an efficient memory layout for a large matrix is equal distribution of rows among all the processors. In this layout, say we have P processors for calculating an NxN matrix M with Nx1 vector V, the following layout is put forward:

- N/P rows of matrix M is sent evenly to all the P processors as given in figure 1
- Vector V is broadcasted to all the P processors.
- Each processor calculates (N/P) X N sub-matrix with vector V to give N/P rows of the result vector.
- The N/P rows are all merged from all the processors ordered by the processor rank.

Algorithm based on figure 1

- Each processor computes N/P rows of C.
- Need entire B vector and N/P rows of A as input.
- Each processor now needs $O(N^2)$ communication and $O(N^2/P)$ computation.

This layout gives better granularity and performance compared to the following layouts:

- Giving entire matrix and vector to a processor (same as sequential)

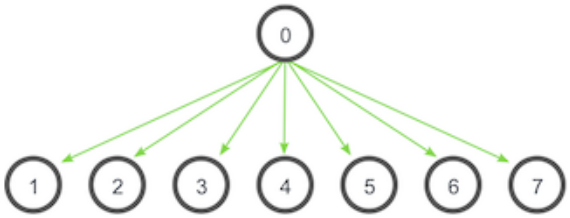

Figure 2: Broadcast in MPI

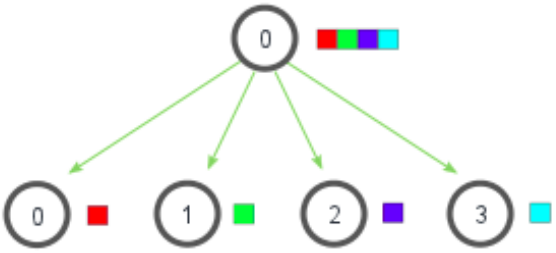

Figure 3: Scatter in MPI

- Providing 1 row and and vector to each processor (is faster in computation than this approach but requires more processor counts and there will be overhead in communication as granularity increases. In the end communication overhead outperforms the computation.)
- Assigning each matrix elements to a processor (More processors required and is suitable only till 4*4 matrix, as the limit is 16 processors. Will have higher communication overhead as each n*n processors have to send and receive data.)

4 MPI Collectives

MPI Broadcast and Collective communication modules were used to take advantage of large memory share, split among multiple processors in synchronization with all the workers. The MPI collectives used are MPI_Bcast, MPI_Gather, MPI_Scatter, MPI_Reduce.

5 Code execution and analysis

This assignment was executed and analysed in the **Cartesius** system instead of the Lisa system as Lisa clusters were pretty unstable and was failing to run slurm batches often. The following modules were loaded to make MPI and OpenMP functional in cartesius.

```
module load pre2019
module load mpi/impi
```

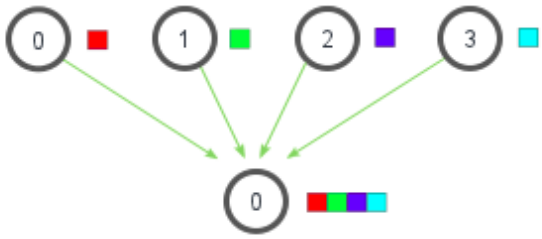

Figure 4: Gather in MPI

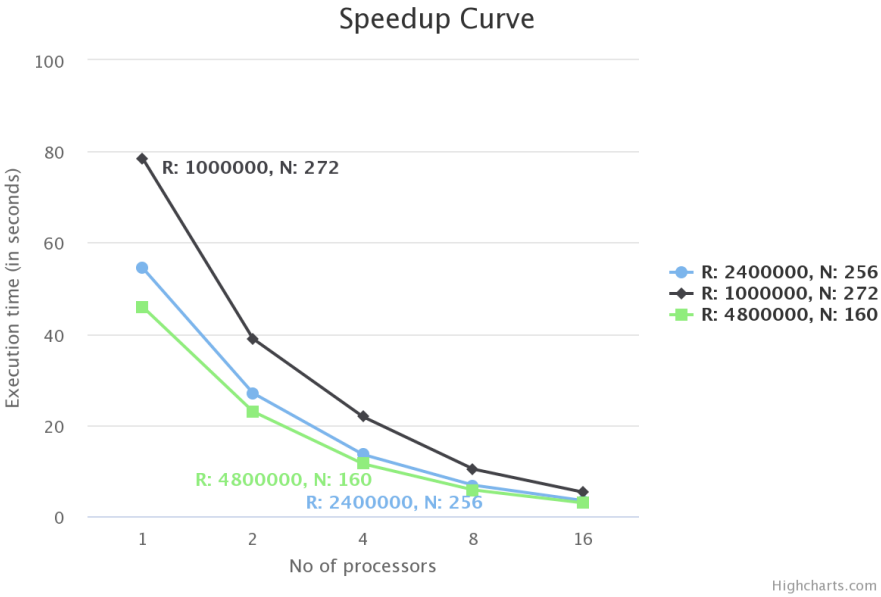

Figure 5: Analysis for different N,R configurations

N	R	processors	time (s)
256	2400000	1	54.447138
256	2400000	2	26.995326
256	2400000	4	13.513177
256	2400000	8	8.145505
256	2400000	16	3.409222
272	1000000	1	78.469719
272	1000000	2	38.92743759
272	1000000	4	21.78423803
272	1000000	8	10.31462117
272	1000000	16	5.23654332
160	4800000	1	45.915411
160	4800000	2	22.963299
160	4800000	4	11.465024
160	4800000	8	5.751984
160	4800000	16	2.904414

Figure 6: Analysis for different N,R configurations

OpenMP was also added above the MPI layers to also speed up the program in the thread level using thread level parallelism. The code was executed with three different N and R configurations for 1, 2, 4, 8 and 16 processors in a reserved node. Figure 5 provides a speed-up curve for the analysed configurations. The following analyses were made:

N: 256, R: 2400000

N: 272, R: 1000000

N: 160, R: 4800000

It was found that, as N increases the workload and communication overload increases per processors and hence the increase in execution time.

6 Performance

The main factor affecting performance in this designed architecture is the dimension N/P, where P is the number of processors. If N/P , i.e the number of rows assigned to each processors increases the execution time also increases, as there are more memory transfers per processor happening and the array size of

computation per processor. The sub-array matrix vector multiplication can be further parallelised in thread level using OpenMP. Its noted that the same program with 16 processors runs about **15.97071525** times faster than the same program with only one processor for the configuration N = 256, R = 2400000. That's a significant speedup.