Oleksandr Borysov

Exercise 4 (Parallel version using Rowwise Block-Striped Decomposition)

1) Task

1. Mapping strategy

   Each process has a part of matrix(rows) and the entire vector.

   Each process computes the inner product of a part of matrix with vector to get part of result vector

   All parts of result vector gather together and form result vector of multiplication.

2. n/p rows are assigned to each processes except last one, last process will process n - (n/p) * (p-1) rows. Where “n” is number of rows in matrix and “p” is number of processes.

3. The first process is advised for I/O because the lowest number of processes on which program can run starts from 1. That is why we definitely know that program will be run at least on the one process and using of the first process for I/O will not corrupt the program. (For example: we set the second process for I/O and start the program with processes parameter equal to one it’s mean that the I/O operation will be not executed because the code of the second process will be not executed)

2) Complexity analysis

What is the expected time for the computational portion of parallel program?

Expected time for computational portion of parallel program is $T = \chi * n^2/p$ (without communication) but MPI processes communicate with each other and it’s required additional time $\approx T = \chi * (n^2/p + n + log_2 p)$.

Show that the expected communication time for the all-gather step is) $O(n + log_2 p)$

The effective communication, each process send $log_2 p$ messages, the total number of elements transmitted is defined as: $n(p-1)/p$, where p is a power of two. Since p is assumed to be constant in terms of the equation, the total complexity of communication could be defined as : $O(n + log_2 p)$. In short all processes are sending $log_2 p$ to one process.

Conclude by giving the complexity of this parallel algorithm

By combining complexity of calculation($O(n^2/p)$) and MPI communication $(O(n + log_2 p))$ we get $O(n^2/p + n + log_2 p)$

Bonus: Compute the isoefficiency and the scalability function. Is this algorithm highly scalable?

Isoefficiency function is:

$n^2 \geq Cpn \Rightarrow n \geq Cp$

Scalability function is:

$M(Cp)/p = C^2p^2/p = C^2p$, where $M(n) = n^2$

Memory utilization must grow linearly with the number of processes that is why the algorithm is not highly scalable.

3) Task

```
void read_row_matrix(char *f, MPI_Datatype dtype, size_t *m, size_t *n, void ***M, MPI_Comm comm) {
    int p, id, typeSize, size, offset;
    MPI_Status st;
    void **matr;

    MPI_Type_size(dtype, &typeSize);
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &id);

    if (id == 0) {
      read_matrix(f, typeSize, m, n, &matr);
    }
    // Share m and n with other processes
    MPI_Bcast(m, 1, MPI_INT, 0, comm);
    MPI_Bcast(n, 1, MPI_INT, 0, comm);
    // Create buff(monolit block of memory)
    *M = createBuff(BLOCK_SIZE(id, p, *m), *n, typeSize);

    if (id == 0) {
       // Send matrices rows to the processes
           for (int i = 1; i < p; ++i) {
         size = BLOCK_SIZE(i, p, *m) * *n;
         offset = BLOCK_LOW(i, p, *m);
         MPI_Send(&matr[offset][0], size, dtype, i, 0, comm);
       }
       // Get matrices rows for the first process
       size = BLOCK_SIZE(id, p, *m) * *n;
       offset = BLOCK_LOW(id, p, *m);
       memcpy(&(*M)[0][0], &matr[offset][0], typeSize * size);
       freeBuff(matr);
    } else {
       // Receive matrices rows
```

```
        size = BLOCK_SIZE(id, p, *m) * *n;
        MPI_Recv(&(*M)[0][0], size, dtype, 0, 0, comm, &st);
    }
}
```

4) Task

```
void read_vector_and_replicate(char *f, MPI_Datatype dtype, size_t *n, void **v, MPI_Comm comm) {
    int p, id, typeSize;
    MPI_Type_size(dtype, &typeSize);
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &id);

    if (id == 0) {
      read_vector(f, typeSize, n, v);
    } else {
      *v = calloc(*n, typeSize);
    }
    // Share m and n between processes
    MPI_Bcast(n, 1, MPI_INT, 0, comm);
    MPI_Bcast(*v, *n, dtype, 0, comm);
}
```

5) Task

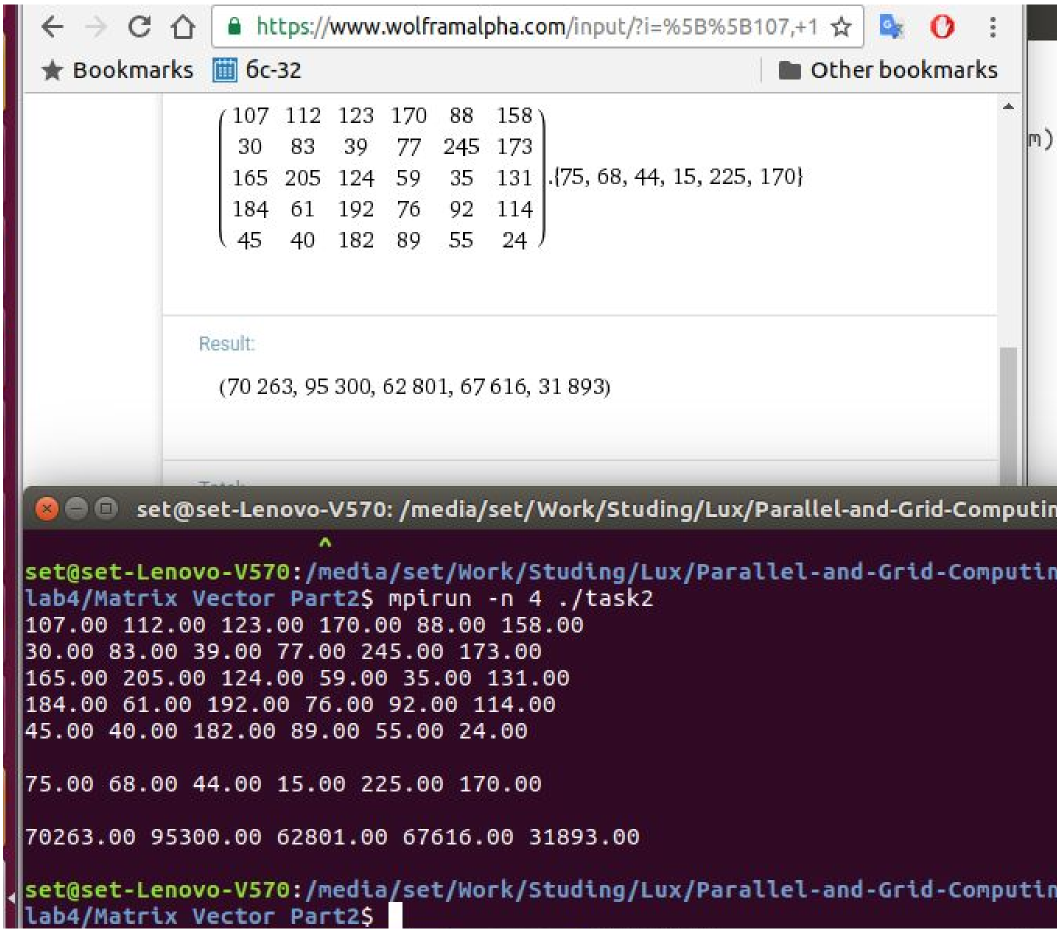
```
  void print_row_matrix(void **M, MPI_Datatype type, size_t m, size_t n, MPI_Comm comm) {
    int p, id, typeSize, size, offset;
    MPI_Status st;
    void **matr;
    int rowsNumber = BLOCK_SIZE(id, p, m);
    MPI_Type_size(type, &typeSize);
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &id);

    if (id == 0) {
     matr = createBuff(m, n, typeSize);
     // Get matrix's rows for the first process
     size = (BLOCK_SIZE(id, p, m) ) * n;
     offset = BLOCK_LOW(id, p, m);
     memcpy(&matr[offset][0], &(M)[0][0], size * typeSize);
     for (int i = 1; i < p; ++i) {
       size = BLOCK_SIZE(i, p, m) * n;
       offset = BLOCK_LOW(i, p, m);
       MPI_Recv(&matr[offset][0], size, type, i, 1, comm, &st);
     }
     print_matrix_lf(matr, type, m, n);
     freeBuff(matr);
    } else {
     size = BLOCK_SIZE(id, p, m) * n;
     MPI_Send(&M[0][0], size, type, 0, 1, comm);
    }
}

void print_row_vector(void *v, MPI_Datatype type, size_t n, MPI_Comm comm) {
    int p, id, typeSize, size, offset;
```

```
  MPI_Status st;
  void *vect;
  int rowsNumber = BLOCK_SIZE(id, p, n);

  MPI_Type_size(type, &typeSize);
  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &id);

  if (id == 0) {
    vect = calloc(n, typeSize);
    // Get vectors elements
    for (int i = 1; i < p; ++i) {
      size = BLOCK_SIZE(i, p, n);
      offset = BLOCK_LOW(i, p, n);
      MPI_Recv((vect + offset * typeSize), size, type, i, 2, comm, &st);
    }
    size = BLOCK_SIZE(id, p, n);
    offset = BLOCK_LOW(id, p, n);
    memcpy(&vect[offset], &(v)[0], size * typeSize);

    print_vector_lf(vect, type, n);
    printf("\n");
    free(vect);
  } else {
    size = BLOCK_SIZE(id, p, n);
    MPI_Send(v, size, type, 0, 2, comm);
  }
}
```

6) Task

```
void* callc_result_part(void **matr, void *v, int m, int n, MPI_Datatype type, MPI_Comm comm) {
  int p, id, typeSize, size, offset;
  void *buff;
  MPI_Type_size(type, &typeSize);
  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &id);

  int elemNumber = BLOCK_SIZE(id, p, m);
  buff = calloc(elemNumber, typeSize);
  for (int i = 0; i < elemNumber; ++i) {
    for (int j = 0; j < n; ++j) {
      if (type == MPI_INT)
        ((int *) buff)[i] += ((int *) v)[j] * ((int **) matr)[i][j];
      if (type == MPI_DOUBLE)
        ((double *) buff)[i] += ((double *) v)[j] * ((double **) matr)[i][j];
      if (type == MPI_CHAR)
        ((char *) buff)[i] += ((char *) v)[j] * ((char **) matr)[i][j];
    }
  }
  return buff;
}
```

Result and proof



7) Task

task2.c file contains the code.

random_matrix_m_n.dat file contains the matrix 10x10.

random_vector_n.dat file contains the vector 10.

PlotData.txt file contains data for the plot.

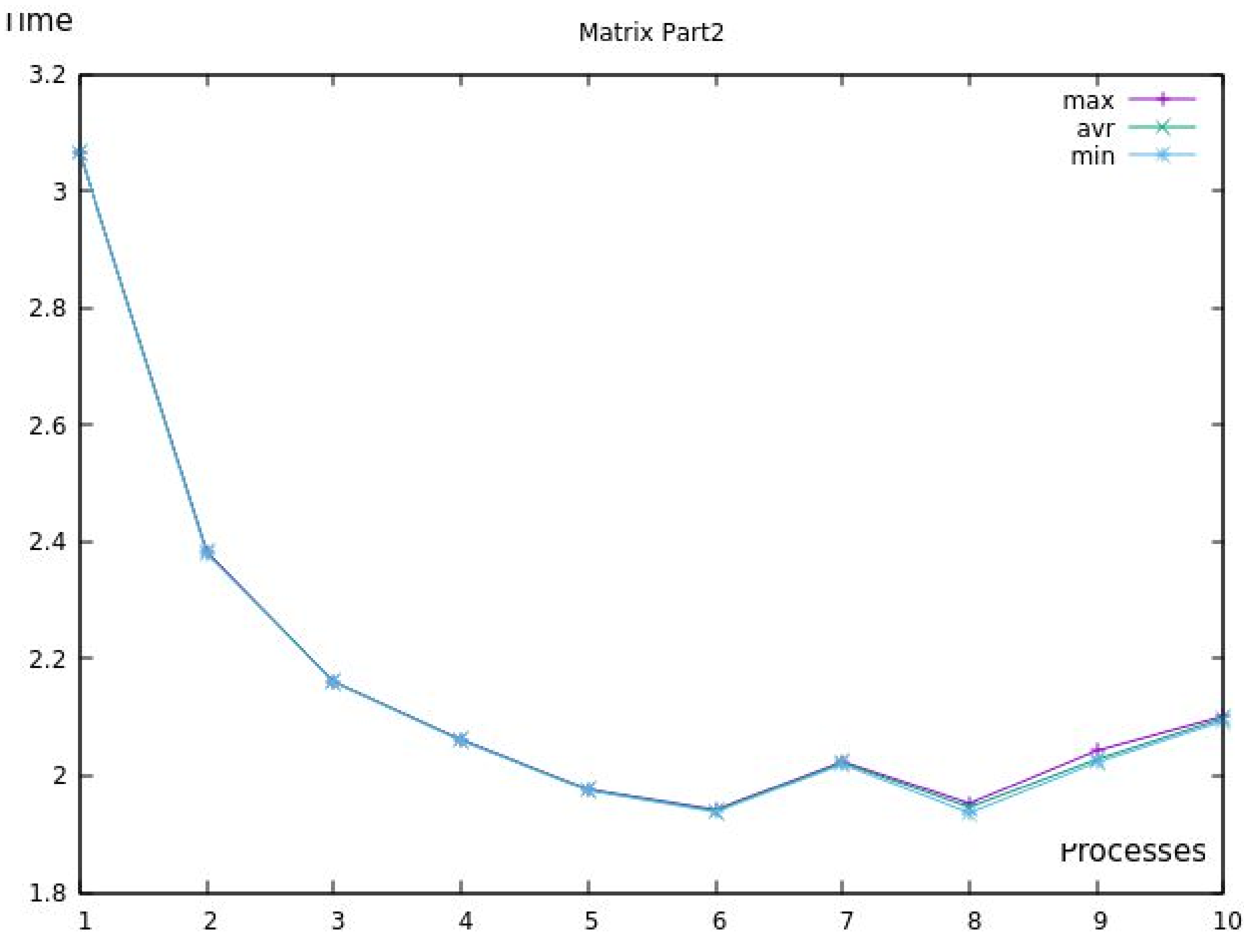
Compile command :

```
module load mpi/OpenMPI/1.6.4-GCC-4.7.2
mpicc task2.c -o task2 -std=c99
```

Execute command :

```
mpirun -n “number of processes” ./task2
```

Plot



(I used matrix 1000x1000 and vector 1000)