HPC Laboratory Assignment

Lab 2: Parallel Programming with MPI

A Distributed Data Structure

Bioinformatics Degree – Second course, Third Term.

Done by Igor Trujnara and Ilia Lecha

Group identifiers: hpc1217, hpc1218

27/04/2023 – Campus Mar, Barcelona.

Table of contents

Introduction

In this lab 2 practical, the main goal is to understand and get familiar with data distributed programs implemented with Open MPI while keeping the exchanging segment boundaries. In addition, understanding the concept of ghost points, why they are required and/or useful and some new MPI directives.

This practical is mainly divided into two parts: One part dedicated to the analysis of some C programs implementing MPI for finding and understanding why the programs are not compiling and another part answering some technical questions related to the MPI environment.

Coding Section

In this part of the practical we have fulfilled some parameters that are missing on the MPI directives from the C programs of Lab 2 in order to make the programs functionals.

```
S1: MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

```
S2: MPI_Comm_size( MPI_COMM_WORLD, &size );
```

```
S3: MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD,  &status );
```

```
S4: MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
MPI_COMM_WORLD );
```

```
S5: MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
MPI_COMM_WORLD, &status );
```

```
S6: MPI_Reduce( &errcnt, &toterr,  1, MPI_INT , MPI_SUM , 0 ,
MPI_COMM_WORLD );
```

```
S7: MPI_Isend( xlocal[maxn/size][] , maxn, MPI_DOUBLE, rank + 1,
0, MPI_COMM_WORLD, &r[nreq++] );
```

```
S8: MPI_Irecv( xlocal[0][], maxn, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD,&r[nreq++] );
```

```
S9: MPI_Isend( xlocal[1][] , maxn, MPI_DOUBLE, rank - 1, 1,
MPI_COMM_WORLD,&r[nreq++] );
```

```
S10: MPI_Irecv( xlocal[maxn/size+1][], maxn    , MPI_DOUBLE, rank
+ 1, 1, MPI_COMM_WORLD, &r[nreq++] );
```

```
S11: MPI_Reduce( &errcnt, &toterr, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD );
```

```
S12: MPI_Sendrecv( xlocal[1], maxn  , MPI_DOUBLE  , prev_nbr, 1,
xlocal[maxn/size+1], maxn, MPI_DOUBLE, next_nbr, 1,
MPI_COMM_WORLD,   status );
```

```
S13: if (next_nbr >= size) next_nbr = MPI_PROC_NULL;
```

```
S14: MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
```

```
S15: MPI_Gather( xlocal[1], maxn * (maxn/size), MPI_DOUBLE, x,
maxn * (maxn/size), MPI_DOUBLE, 0, MPI_COMM_WORLD );
```

```
S16: MPI_Wait( &r[2], &status );
```

```
S17: MPI_Waitall( nreq, r, statuses );
```

```
S18: MPI_Wait( &r[3], &status );
```

```
S19: MPI_Iallreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD, &r[0] );
```

```
S20: MPI_Igather(xlocal[1], maxn * (maxn/size),MPI_DOUBLE, x, maxn
*   (maxn/size),   MPI_DOUBLE,   0,   MPI_COMM_WORLD,   &r[0]   );
```

```
S21:  return  (rowsTotal  /  mpiSize)  +   (rowsTotal  %  mpiSize  >
mpiRank)                                                          ;
```

```
S22: nrows  = getRowCount(maxn, rank , size );
```

```
S23: MPI_Gather( &lcnt, size   , MPI_INT   , recvcnts, 1, MPI_INT, 0
,MPI_COMM_WORLD                                                    );
```

```
S24: MPI_Gatherv( xlocal[1], size   , MPI_DOUBLE,

                 x, recvcnts   , displs, MPI_DOUBLE,

                                                     0,   MPI_COMM_WORLD   );
```

Questions section

Q1: Why do we need to use MPI_Allreduce instead of MPI_Reduce in all the codes in section 1.2?

We need to use MPI_Allreduce because the total error obtained in the matrix determines the convergence of the system, and thus the termination of the calculation. The partial results are stored in each process separately, and the total result needs to be known to all processes. This is what MPI_Allreduce does – it performs the specified reduction on the variable, and then stores the result into the specified buffer of every process in the communicator.

Q2: Alternatively, which other MPI call would be required if we had used MPI_Reduce in all the codes in section 1.2?

The same result could be achieved by using MPI_Reduce to find the final result and then MPI_Bcast to copy it to other processes.

Q3: We have said that we have a halo so that some of the values in the matrix are read but not written during the computation. Inspect the code and explain which part of the code is responsible for defining such halo.

As can be seen on the par_data_struct.c, the program is defining the halo before sending or receiving the data from other processes:

```
/* 2) Fill ghost points with -1 */

   for (j=0; j<maxn; j++) {

       xlocal[0][j] = -1;

       xlocal[maxn/size+1][j] = -1;

   }
```

Once these values are set, the program shall deal to pass and receive the row 1 (not the 0) and the row N-1 (maxn) which represents the last row before the ghost point.

Q4: Explain with your own words how the load is balanced across the different processes when the number of rows is not evenly divided by P.

We have seen that when the workload can be divided equally for all processes we just have to somehow calculate the amount of work to be done by each process.

But when it is the case that it cannot be divided equally, we apply the following function:

```
int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {

        /* Adjust slack of rows in case rowsTotal is not exactly divisible */

        return (rowsTotal / mpiSize) + (rowsTotal % mpiSize > mpiRank);}
```

This function uses integer division to divide the total number of rows by the number of processes to determine how many rows each process should handle on average. However, if the number of rows is not evenly divisible by the number of processes, there will be a remainder.

To handle this remainder, the function uses the modulo operator (%) to determine if the current process should handle an extra row (the slack). If the remainder (rowsTotal % mpiSize) is greater than the current process rank (mpiRank), then that process should handle the extra row, and so the function adds 1 to the average number of rows.