CONCURRENCE, PARALLELISM AND DISTRIBUTED SYSTEMS

Assignment 1
Parallel Programming with MPI
A Distributed Data Structure

Mehdi Hassanpour - cpds1124
mehdi.hassanpour@upc.edu

Fall 2024-25
Polytechnic University of Catalonia

05-10-2024

1 A distributed data structure

```
S1  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* Statement S1 */
S2  MPI_Comm_size(MPI_COMM_WORLD, &size); /* Statement S2 */
S3  MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &status);
        /* Statement S3 */
S4  MPI_Send(xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD); /*
        Statement S4 */
S5  MPI_Recv(xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD,
        &status); /* Statement S5 */
S6  MPI_Reduce( &errcnt, &toterr, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD); /*
        Statement S6 */
S7  MPI_Isend(xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, &
        r[nreq++]); /* Statement S7 */
S8  MPI_Irecv(xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &r[nreq
        ++]); /* Statement S8 */
S9  MPI_Isend(xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD, &r[nreq
        ++]); /* Statement S9 */
S10 MPI_Irecv(xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD,
         &r[nreq++]); /* Statement S10 */
S11 MPI_Reduce(&errcnt, &toterr, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD); /*
        Statement S11 */
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 );   /* Statement
        S12 */
```

all 3 files successfully compiled and executed with no errors.

2 A simple Jacobi iterative method

1. MPI_Allreduce behaves the same as MPI_Reduce except that the result appears in the receive buffer of all the group members. That is why we have to use MPI_Allreduce; We want to have this reduction to be shared with all the processes.

2. If we were to use MPI_Reduce instead, we had to send the results to others using a broadcast, MPI_Bcast.

3. First is initializing the halo (ghost rows) for each process. Specifically, $xlocal[0][j]$ stores data from the previous process, and $xlocal[maxn/size+1][j]$ stores data from the next process.

   ```
   for (j = 0; j < maxn; j++) {
       xlocal[i_first - 1][j] = -1;
       xlocal[i_last + 1][j] = -1;
   }
   ```

   Sending and receiving rows to and from neighboring processes establishes and updates the halo regions, ensuring that each process has the necessary ghost points ($xlocal[0][j]$ and $xlocal[maxn/size+1][j]$) to perform computations involving neighboring elements correctly. The MPI_Send and MPI_Recv calls exchange boundary data between neighboring processes, thus updating the halo. To manage communications efficiently, MPI_PROC_NULL is used for out-of-bound neighbors (prev_nbr and next_nbr), eliminating the need for additional if statements. These halo regions are updated during communication but are only read during computation, ensuring correctness without being modified during the calculation itself.

```
MPI_Send(xlocal[maxn / size], maxn, MPI_DOUBLE, next_nbr, 0,
    MPI_COMM_WORLD);
MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, prev_nbr, 0, MPI_COMM_WORLD, &
    status);
MPI_Send(xlocal[1], maxn, MPI_DOUBLE, prev_nbr, 1, MPI_COMM_WORLD);
MPI_Recv(xlocal[maxn / size + 1], maxn, MPI_DOUBLE, next_nbr, 1,
    MPI_COMM_WORLD, &status);
```

Lastly, During the computation, the ghost points (xlocal[i - 1][j] and xlocal[i + 1][j]) are read but not updated. This means that each process has access to the required boundary data from its neighboring processes, which defines the halo.

```
for (i = i_first; i <= i_last; i++)
    for (j = 1; j < maxn - 1; j++) {
        xnew[i][j] = (xlocal[i][j + 1] + xlocal[i][j - 1] +
                      xlocal[i + 1][j] + xlocal[i - 1][j]) / 4.0;
        diffnorm += (xnew[i][j] - xlocal[i][j]) * (xnew[i][j] - xlocal[i
    ][j]);
    }
```

4. When the number of rows is not evenly divisible by the number of processes, the workload is distributed to ensure it remains balanced. Each process is assigned row-sTotal / mpiSize rows, and the first few processes receive an additional row if there is a remainder (rowsTotal % mpiSize > rank). This approach ensures that the overall workload is distributed as evenly as possible, with any extra rows allocated to the first few processes, keeping the load balanced across all processes.

5. Below you can find the completed statements for Jacobi codes.

```
S13 if (next_nbr >= size) next_nbr = MPI_PROC_NULL; /* Statement S13 */
S14 MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
         /* Statement S14 */
S15 MPI_Gather(xlocal[1], maxn * (maxn/size), MPI_DOUBLE, x, maxn * (maxn/size),
        MPI_DOUBLE, 0, MPI_COMM_WORLD ); /* Statement S15 */
S16 MPI_Wait(&r[2], &status); /* Statement S16 (Fix the error) */
S17 MPI_Waitall(nreq, r, statuses); /* Statement S17 */
S18 MPI_Wait(&r[3], &status); /* Statement S18 */
S19 MPI_Iallreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD,
         &r[0]); /* statement S19 */
S20 MPI_Igather(xlocal[1], maxn * (maxn/size), MPI_DOUBLE, x, maxn * (maxn/size),
         MPI_DOUBLE, 0, MPI_COMM_WORLD, &r[0]); /* Statement S20 */
S21 return (rowsTotal / mpiSize) + (rowsTotal % mpiSize > mpiRank); /* Statement
        S21 */
S22 nrows  = getRowCount(maxn, rank, size); /* Statement S22 */
S23 MPI_Gather(&lcnt, 1, MPI_INT, recvcnts, 1, MPI_INT, 0, MPI_COMM_WORLD ); /*
        Statement S23 */
S24 MPI_Gatherv(xlocal[1], lcnt, MPI_DOUBLE, x, recvcnts, displs, MPI_DOUBLE, 0,
        MPI_COMM_WORLD ); /* Statement S24 */
```