Assignment 1 - CPDS Parallelism
Parallel Programming with MPI
A Distributed Data Structure

Roberto Meroni
email: roberto.meroni@estudiantat.upc.edu

October 2023

Section 1.1

1. *Write the solution for 12 statements, identified as S1 . . . S12 in the source codes, that have some missing parameters in some calls to MPI primitives. Indicate the identifier of the statement and how you have filled it in.*

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);
```

Section 1.2

1. *Why do we need to use MPI Allreduce instead of MPI Reduce in all the codes in section 1.2?*

   In all of the 3 codes (with a variation on *itcnt* for the third one) the exit condition from the loop is:

   ```
   while (gdiffnorm > 1.0e-2 && itcnt < 100);
   ```

   gdiffnorm is a sum of variables across different processes and is received as a reduction result. In order to have every process to exit from the loop consistently with our stopping criteria, we need to send *gdiffnorm* to every process, using MPI_Allreduce (with MPI_Reduce it would have been received only by 1 process).

2. *Alternatively, which other MPI call would be required if we had used MPI Reduce in all the codes in section 1.2?*

   An alternative is to replace the line

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

   with the lines:

   ```
   MPI_Reduce( &diffnorm, &gdiffnorm, 1,
   MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD );

   MPI_Bcast( &gdiffnorm, 1, MPI_DOUBLE, 0,
   MPI_COMM_WORLD);
   ```

   This way, only one process (here process 0) receives the value of the norm that was computed by collecting and summing partial results from all processes. In order to synchronize all processes and enable them to check for convergence, the same process that received the value of the norm then sends *(broadcasts)* it back to all processes.

3. *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.*

   The lines responsible for the definition of the halo are the lines:

```
i_first = 1;
i_last  = maxn/size;

if (rank == 0)         i_first++;
if (rank == size - 1) i_last--;


       .......


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]);
        }

for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++)
        xlocal[i][j] = xnew[i][j];
```

   With that, for the process responsible for the first segment of the mesh, the row index of the local matrix to be updated starts at 2, so it leaves unmodified row 0 (reserved for ghost points even though the first process doesn't need it) and row 1 (effective first row). Similarly, for the process responsible for the last segment of the mesh, the last row of the local matrix to be updated is *maxn/size - 1*, so it leaves unmodified the effective last row.

   Also, column index j varies from 1 to *maxn - 2* for every process, so the first column of the mesh (column 0) and the last column (column *maxn - 1*) are left unmodified.

4. *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.*

   If the total number of rows can be exactly balanced between all processes we can calculate the number of rows to assign to each process with:

   ```
   (rowsTotal / mpiSize)
   ```

   Otherwise, if the total number of rows (rowsTotal) is not exactly divisible by the number of processes (mpiSize), we want to assign one extra row to some processes, until we have assigned all the rows. Therefore, we add the following term to the previous expression:

   ```
   (rowsTotal % mpiSize > mpiRank)
   ```

   The modulo returns the extra rows to assign and is compared with the rank of each process (mpiRank). This is a boolean expression, which returns 1 if the number of extra rows is greater than a process rank, otherwise 0. We use the *greater than* operand, as opposed to *greater than or equal* as process identifiers (ranks) start from 0. Thus, 1 extra row is assigned to each process with a rank lower than the number of extra rows. The final expression is:

   ```
   (rowsTotal / mpiSize) + (rowsTotal % mpiSize > mpiRank)
   ```

5. *Write the solution for the statements identified as S13 . . . S24 in the source codes, which have errors or some missing parameters in some calls to MPI primitives. Indicate the identifier of the statement and how you have filled it in:*

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

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

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

S24

```
MPI_Gatherv( xlocal[1], lcnt  , MPI_DOUBLE,
x, recvcnts  , displs, MPI_DOUBLE, 0,
MPI_COMM_WORLD );
```