CPDS Laboratory Assignment 1: Parallel Programming with MPI A Distributed Data Structure

Pau Adell Raventós: cpds1211
Jaya García Fernández: cpds1216

Master in Innovation and Research in Informatics (MIRI)

Concurrency, Parallelism and Distributed Systems

Facultat d'Informàtica de Barcelona (FIB)

Contact: pau.adell@estudiantat.upc.edu and jaya.garcia@estudiantat.upc.edu

2024-2025 Q1

11/11/2024

1 Introduction

This first assignment is an introduction to MPI message passing between shared data structures. Here we will take a look at different codes that implement this data structure along various processes and communicates between them to compute an approximate result to a linear system of equations.

2 A distributed data structure

In this first section we start working with a data structure that is distributed among different processes. The goal is to understand how MPI works when sharing some data between processes. We worked with 3 different files, `par_data_struct.c`, `par_data_struct_nonblocking.c` and `par_data_struct_sendrecieve.c`, where each have different sharing methodologies. In each file we filled different statements stated here:

```
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 );
```

3 A simple Jacobi iterative method

In this second section we work with the same data structure to compute an approximate result to the solution of a linear system of equations. For this we use the Jacobi iterative method.

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

   It is a condition that needs to globally determine whether the iterative method has converged or if it needs more iterations. To do so, we need to add all the values and store them in gdiffnorm and like this, all of the processes will be ready to check the condition and continue with its corresponding process.

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

   The MPI_AllReduce leaves us the final value in a variable in all the processes, so if instead we used MPI_Reduce, we would need to save this value (result) in a root process and then send it back to the rest

of the processes. This means that in addition a MPI_Bcast call should be used to distribute the result from the root to all other 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.**

   In this computations the halo sections correspond to the ghost areas of each process. In the code, this values are defined before any message is passed, initializing the first and last rows to -1. Afterwards, when the data transfer starts, each process receives the corresponding rows and stores them in these sections, but in the context of a process, this rows are only for reading.

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.**

   When the number of rows is not evenly divided by P, the load is balanced in the following way: first we assign to each process $maxn\ P$ rows, without counting ghost areas, and then for the remaining $maxn\%P$, we give one row to each process until no more are left, starting by the first process. For example, when $P = 7$ and $nmax = 22$, we assign $26/7 = 3$ rows to each process and one extra to the first $13\%5 = 5$ processes. The halos are dealt in the same manner as before.

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:  return (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
);
```