Lab Guide 9

MPI-Based Parallelization

Objective:

- Learn the basics of program parallelization with distributed memory programming (message passing)
- Understand the basic types of parallelization: 1) functional versus 2) data parallelization

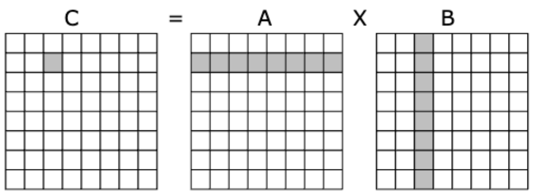
Introduction

This lab session aims to apply the basic MPI communication concepts studied in the previous session to parallelize simple applications.

Exercise 1 – Distributed memory matrix multiplication with MPI collective operations

Download the matrix multiplication code from lab session 0 (base DOT version in the figure).

$$\boxed{C_{ij} = DOT_{linha\_A_i,coluna\_B_j} = \sum_{k=0}^{n-1}(A_{ik} * B_{kj})}$$



```
foreach line of A
       foreach column of B
         $C_{line,column}$ = $DOT_{line of A,column of B}$
```

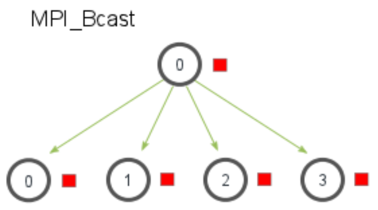
a) Identify a strategy to implement the matrix multiplication with MPI:

1. Each process will compute a subset of the elements of matrix C. Identify the main alternatives to partition the computation of this matrix among processes.
2. Select the simplest alternative to partition the computations of C. What data from matrix A and matrix B is required to perform the computation of each part of C?

b) Implement the selected approach using MPI collective operations. The relevant collective operations are:

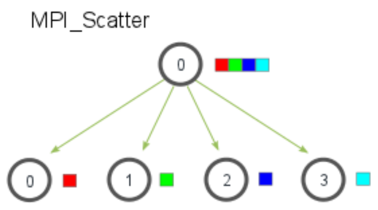
1. **Broadcast** the data from root to all other processes:

```
int MPI_Bcast(void* buffer, int count,
  MPI_Datatype datatype, int root, MPI_Comm comm)
```



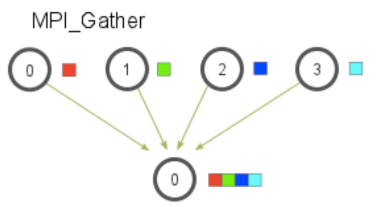
2. **Scatter**: scatters data from root into all other processes

```
int MPI_Scatter(void* sbuf, int scount,
  MPI_Datatype stype, void* rbuf, int rcount,
  MPI_Datatype rtype, int root, MPI_Comm comm )
```



3. **Gather**: Joints data from all processes into the root

```
int MPI_Gather(/* same signature of Scatter*/)
```



Exercise 2 – Prime calculation using the Sieve of Eratosthenes

Consider the following sequential program, which finds all prime numbers up to a given `MAXP`:

```
int MAXP = 1000000;
int SMAXP = 1000;
int pack=MAXP/10;

PrimeServer *ps1 = new PrimeServer();
PrimeServer *ps2 = new PrimeServer();
PrimeServer *ps3 = new PrimeServer();

ps1->minitFilter(1,SMAXP/3,SMAXP);
ps2->minitFilter(SMAXP/3+1,2*SMAXP/3,SMAXP);
ps3->minitFilter(2*SMAXP/3+1,SMAXP,SMAXP);

int *ar = new int[pack/2];
for(int i=0; i<10; i++) {
    generate(i*pack, (i+1)*pack, ar);
    ps1->mprocess(ar,pack/2);
    ps2->mprocess(ar,pack/2);
    ps3->mprocess(ar,pack/2);
}
ps3->end();
```

Copy the `/share/cpar/P09_Codigo` folder to your home directory in the SeARCH cluster.

Compile the program in the cluster frontend using the `mpic++ -O2 -o primes PrimeMain.cpp` command.
Use the `sbatch primes_mpi.sh` command to run the application.
The `primes_mpi.sh` file should specify the required resources and should run the MPI application.
The following example requests three PUs and spawns three MPI processes:

```
[search7edu]$ cat primes_mpi.sh
#!/bin/bash
#SBATCH --time=1:00
#SBATCH --ntasks=3
#SBATCH --partition=cpar
mpirun -np 3 ./primes
```

The number of requested resources (`--ntasks`) must be the same as the number of processes (`-np`) used in the `mpirun` command.

a) Parallelize the code using MPI through the implementation of a pipeline of processes that receives an array of integers, created by the `generate` function, and each process filters out a subset of the input.
The `mprocess` method implements the filtering of the primes, and `end` prints the final amount of primes found. This pipeline should have 3 processes, one for each instance of `PrimeServer` performing the filtering.

b) (*) Modify the parallelization implemented in **a)** to work with an arbitrary number of processes and messages.

c) (*) Parallelize the sequential application through the implementation of a farm of processes behaving in a "work sharing" paradigm with dynamic scheduling.