Nicolas Ferradas
Cristiano Caon

Parallel and Concurrent Programming: Assignment 2

Algorithm:

Our idea to implement the functionality of Barrier using MPI_Send()'s and MPI_Recv()'s consists on letting every other process know that the other processes have reached a certain portion of code. As we need to make this work in no more than $8\log_2 N$ steps, making each process send their data to every process was not an option. We decided to go with the following idea: We would divide the number of processes into chunks after every round of communication, matching processes with their corresponding opposite. So, for 8 processes we would start with one chunk of processes (0-7) and match them like this:

0	1	2	3	4	5	6	7

Matches: (0, 7), (1, 6), (2, 5), (3, 4)

This allows each half of the chunk to have the data from the other half. So now, processes in each chunk only need to communicate with the processes in their current chunk.

We divide the chunks further and perform the same idea:

Chunk 1			
0	1	2	3
Matches: (0, 3), (1, 2)			

Chunk 2			
4	5	6	7
Matches: (4, 7), (5, 6)			

Chunk 1	
0	1
Matches: (0, 1)	

Chunk 2	
2	3
Matches: (2, 3)	

Chunk 3	
4	5
Matches: (4, 5)	

Chunk 4	
6	7
Matches: (6, 7)	

By doing this, every process would know that the rest of the processes reached a certain part of code on $\text{Log}_2 N$ steps. Let's take process 0 as example:

1st Communication: 0 -> data from (0, 7)

2nd Communication: 0 -> data from (0, 7, 3, 4,) *since 3 gave 0 data from 4

3rd Communication: 0 - > data from (0, 1, 2, 3, 4, 5, 6, 7) *Now we are done.

Implementation:

To make this work, we decided to implement a base chunk idea, where we use Chunk 1 in every iteration to determine the pair for each process in other chunks. We start by setting a variable gap = np, and we are going to use the gap variable to determine the number of chunks at each step, the variable will be divided by two after every communication. It will serve as the divider in every step. With that, we know that we can find the matching process with the following formula: gap – pid – 1 = matchingPID, if and only if pid < gap. To use the same equation for every chunk we need to make each chunk satisfy the condition pid < gap, to do that we use the mod operation (%). Using the code below we would be able to get the matching pair for any process or pid.

Nicolas Ferradas
Cristiano Caon

```
basePID = pid % gap;     // This will give us the corresponding pid for the
                         base chunk

                         // for example, when pid = 5 gap = 4, basePID = 1
                         which is what we want.

basePairPID = gap – basePID – 1

matchingPID = basePairPID + gap (pid / gap)     // we multiply gap by the
                                                result of pid/gap to know to

                                                // what chunk we need to jump
                                                to find the match.
```

Now that we know the pair, we just send data from the pid to the matching pid and expect to receive data from that matching pid as well. Once the variable gap becomes 1, we know every process is at the barrier.