Programmation Parallèle
TP n°1 - Getting hands dirty with MPI

and so dirty will they become...

Oguz Kaya
oguz.kaya@lri.fr

8/11/2018

Part 1

Yyello world!

Question 1

a) Yeah, you know the drill. Boring, yet we do it for fudge cakes. Implement a yyello world MPI program from scratch, and print the rank as well as the total number of ranks available at each MPI process.

Part 2

Enough of theory, stepping into the reality...

Enough of theoretical nonsense! Now is the time to get some real work done with real MPI code. In this exercise, we will (actually) sort bitonic arrays using bitonic separators (or half-cleaners if you will). To save the precious time of the genious minds, your assistants did some labor work for you by creating a skeleton code `bitonic-sort-skeleton.c` that handles the basic setup, I/O, and verification nonsense (Yay!). You will only have to deal with the solution file `bitonic-sort-solution.c` that is automatically imported into the main skeleton code.

Now open up the solution code `bitonic-sort-solution.c` where you will implement everything. You will see at the top many commented out variables with descriptions for what they stand for. Do not uncomment any of them! They are already defined in the skeleton code, and are put there so that you can develop your code there without touching the skeleton code.

We will be sorting a bitonic array of size N in nondecreasing order using N processes (each of which holding a single number). You can assume that N is always a power of two for simplicity. For now, let us choose $N = 8$. Execute the script `gen-bitonic-array.py` with the parameters 8 and `bitonic-array.txt` to create a bitonic array of size 8:

```
./gen-bitonic-array.py 8 bitonic-array.txt
```

Now try to compile the skeleton code as:

```
mpicc -std=c99 bitonic-sort-skeleton.c -o bitonic-sort
```

Finally, run the executable for sequential sorting as:

```
mpirun -np 8 ./bitonic-sort bitonic-array.txt sequential
```

which should print out the original array and the sorted array. Now let's try to sort array in parallel by typing

```
mpirun -np 8 ./bitonic-sort bitonic-array.txt parallel
```

which does not sort the array of course, as you have not implemented anything in `bitonic-sort-solution.c`, yet!

Question 2

a) We assume that we have enough memory only in the master process (with rank) 0, who reads and stores the entire bitonic array in `arr` (already done by the skeleton code). It is forbidden to allocate any arrays in any other process, so do not cheat (I'll be watching you!). Therefore, expect in the solution code that `arr` is filled with the input array for the process with rank 0. First, we need to distribute each element `arr[i]` to the process i, as other processes have no data yet. We will use `MPI_Scatter` in order to perform this so that at the end, each process has its element correctly placed in the variable `procElem`. Refer to the MPI cheatsheet and documentation for the usage of `MPI_Scatter`.

b) Now that the array is distributed, we will iterate $\log_2 N$ steps of bitonic separators in order to sort the array. As you remember, at each iteration, each process should have a "pair" process, and the "lower" pair should always be receiving the smaller element while the "upper" pair getting the larger one after the comparison. You should determine the pair of each process at every level, then communicate the elements with the pair using `MPI_Send` and `MPI_Recv`. Refer to the MPI cheatsheet and documentation for the usage of these two routines. As a guideline, we provide the following figure depicting a bitonic sorting network for $N = 8$.
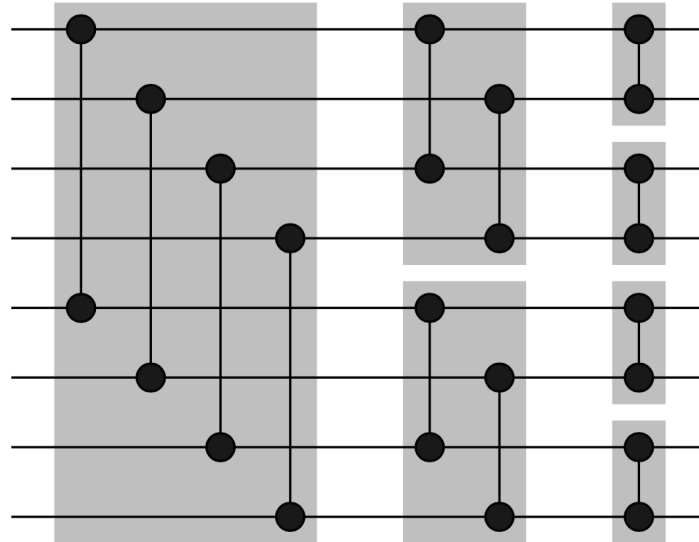
c) Is the bitonic array sorted now? Are you sure? Well, we will see about that in a moment... Now we will try to perform the "mirror image" of the communication that we did in the first part. We will "gather" these scattered (and hopefully sorted!) elements in processes in the `arr` array of the master process (with rank 0). Refer to the MPI cheatsheet and documentation for the usage of `MPI_Gather`. Once you do this, the skeleton code will automatically validate if the array is sorted, and print an error otherwise for you to debug your code accordingly. No bread and water to you until the code sorts correctly! Now that you validated your code working for $N = 8$, try to test it for powers of two, from $N = 2$ up to $N = 64$.

d) Instead of using `MPI_Send` and `MPI_Recv`, we can make use of `MPI_Isend` and `MPI_Irecv`, which should liberate us from having to validate that we send and receive calls are in the right order. This time, do the communication using these non-blocking variants. Do not forget to use `MPI_Wait` at the end to make sure that the communication is completed!

e) Instead of doing one `MPI_Send` and `MPI_Recv`, once can also perform `MPI_SendRecv` to accomplish both communications at the same time (which could potentially be done faster)! Try to replace sends and receives in your code with `MPI_SendRecv`, then make sure it works correctly.

Question 3

a) Try to implement a basic version of the MPI routine `MPI_Scatter` in which the data type is set to be `int` and the block size is always 1. You should be only using two MPI routines `MPI_Send` and `MPI_Recv`. The function signature is provided in the code `scatter-gather.c`. Try to implement the function there, then replace it with the `MPI_Scatter` you use in the previous exercise. This time, when you compile the code, do not forget adding `scatter-gather.c` to the list of source files in `mpicc`. Make sure that everything works as expected!

b) Do the same, this time for `MPI_Gather`.

┌─ Part 3 ───
│   **Do not forget to keep a copy of the precious code you developed for later!**
└──