Parallel Programming
TP - Getting hands dirty with MPI

and so dirty will they become...

Oguz Kaya
oguz.kaya@lri.fr

Part 1

Yyello world!

Exercise 1

a) Yeah, you know what it is. Boring, yet we do it anyway. Implement a hello world MPI program that prints the rank of each process as well as the total number of processes.

Part 2

Bitonic sort

Now is the time to get some real work done with real MPI code. In this exercise, we will sort bitonic arrays using bitonic separators (or half-cleaners if you will). To save the precious time of genious minds, your professors did some labor work for you by creating a skeleton code `bitonic-sort.cpp` that handles the basic setup, I/O, and verification nonsense (Yay!). You will only have to implement the relevant functions in this file (indicated by `\\...` in the code).

A bitonic array A of size N is an array that is monotonically increasing up until an index $1 \leq k \leq N$, and monotonically decreasing from then on. For example, $A = [1, 2, 4, 7, 10, 9, 8, 5]$ is bitonic whereas $A = [1, 4, 7, 10, 8, 5, 6, 2]$ is not. 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. In the first phase, every element with index i is compared with the element at $N/2$ distance from it ($i + N/2$ or $i - N/2$, depending on whether i is in the lower or upper half of the array). The smaller of the two stays in the lower part, and the larger of the two stays in the upper part of the array after the comparison. This will make sure that the smallest $N/2$ elements of the array reside in the lower part and form a bitonic array, whereas the largest $N/2$ elements reside in the upper part and form a bitonic array, so we can sort each half independently afterwards. In the next phase, the same procedure is applied recursively to each half of the array. The array becomes entirely sorted at the end of $\log_2 N$ phases as shown in Figure 1.

Now open up the skeleton code `bitonic-sort.cpp` where you will implement everything. You will not need to modify any other parts of the code except the parts indicated by the commented lines (`\\...`) . 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:

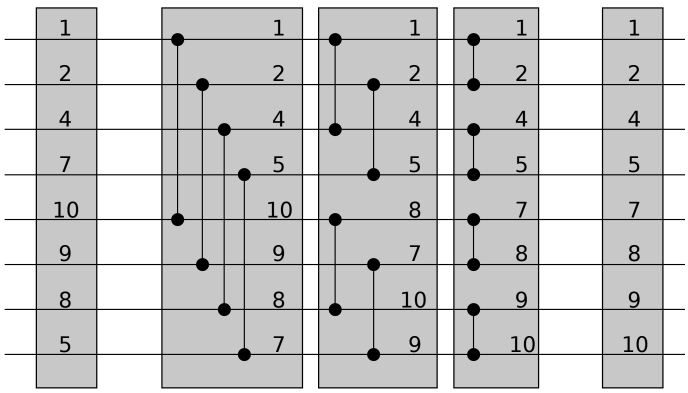

Figure 1: Bitonic separator sorting a bitonic array.

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

Now try to compile the skeleton code as:

```
mpic++ -std=c++11 bitonic-sort.cpp -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 sequentially 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.cpp`, yet!

Exercise 2

a) We will now start implementing the bitonic sort in the function `bitonicSort(int *arr, int N)`. 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 that `arr` is filled with the a bitonic array in 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`. Verify that each process received the correct element.

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, as shown in Figure 1. 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.

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 $N = 64$ (and make sure to use the same number of processes when running `mpirun -np ...`).

d) 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.

Exercise 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 implement this using `MPI_Send` and `MPI_Recv` functions. The skeleton of this function is provided in the code as `MPI_ScatterSingleInt(...)`. Try to first implement the function there, then replace it with the `MPI_Scatter` you use in the previous exercise. Make sure that everything works as expected!

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