Parallel and Distributed Algorithms and Programs
TP 2&3 - The Broadcast on the Rings

One Message to find them all. One Routine to send them,
One Routine to receive them all and in the buffers, bind them.

Valentin Le Fèvre
valentin.le-fevre@ens-lyon.fr

Tien-Nam Le
tien-nam.le@ens-lyon.fr

In this session, we will be dealing with the good-old one-to-all broadcast communication routine on a ring topology, and implementing/optimizing a couple of algorithms of different costs. We will broadcast the data available at the root process with rank 0 to all other processes. In this TP session and the following ones, we will start using SMPI/SimGrid as our testbench to be able to measure the performance of our code and better understand how algorithms perform according to different network parameters. Indeed, there is a lot of coding involved to test, implement, and verify these algorithms. But lucky you! Your teaching assistants have already done the most of this coding for you! You will only need to implement the algorithms in a *skeleton code* that we provide, and everything else (input, output, verification) will work like magic afterwards! Cool, huh?

You will find the skeleton code that we provide in `bcast_skeleton.c`. The code might look like a "bazaar" if you were curious enough to already look at it. If you have not, it is not a problem as you will not even need to touch it! Instead, try to open `bcast_solution.c` which includes some variables that are commented at the top. Do not uncomment them! These are variables already defined in the skeleton code, and the comments are there to help you develop your code without messing with the skeleton file. Here is a description of these variables:

- `num_procs`, `rank`: They speak for themselves. The number of MPI ranks, and rank of the current process.
- `bcast_implementation_name`: The name of the broadcast algorithm to be executed. The exact names you will use are given in the description of each section below.
- `chunk_size`: The size of each chunk in the pipelined algorithms. We will come to this later.
- `buffer`: A char array containing the data to be communicated.
- `NUM_BYTES`: The size of the buffer (in bytes, or the number of char elements).

We will use SMPI to simulate a ring topology consisting of 32 processors. Using SMPI is almost identical to using MPI, except that to compile and run your code you need to execute SMPI routines `smpicc` and `smpirun`. To create the executable `bcast`, you can simply compile your code by typing the following line:

```
smpicc bcast_skeleton.c -o bcast
```

As you might already expect, in order to "simulate" the code on a particular interconnection topology, we need to provide SMPI with the description of this particular topology. This description involves the list of available virtual nodes, or *hosts*, which is provided in a *host file*, and the interconnections

between these nodes, which is given in a *platform file*. Creating these topologies can be complicated, and that is precisely where your assistants come to rescue again![1] We provide you a python script called `smpi-generate-ring.py` that creates a ring topology for a given number of nodes, and allows you to specify the latency and the bandwidth of the links of the ring topology. To generate a 32-processor ring with a $1\mu s$ link latency and $100Gbps$ link bandwidth, simply type

```
python smpi-generate-ring.py 32 100 100 100Gbps 1us
```

which should generate the host file `ring-32-hostfile.txt` and the platform file `ring-32-platform.xml`. Normally, the second and the third parameters correspond to the computational power of your local machine, and each node in the simulated topology, respectively (both are set to 100 gigaflops per second in this example). However, in this exercise we will only be doing communications, so you can ignore these values for now.

Now that you have your hostfile and the platform file, you are finally ready to execute the code by typing

```
smpirun -hostfile ring-32-hostfile.txt -platform ring-32-platform.xml ./bcast
                        [bcast_implementation_name]
```

.We have already provided the default MPI broadcast algorithm `default_bcast` in the solution code, so you can go ahead and try the code to make sure that the skeleton code works as expected.

Part 1

Keep it simple, and stupid!

The path to wisdom always passes through idiocity, and so shall we do! In this exercise you are asked to implement the foolest and the simplest broadcast algorithm you can imagine. The root directly sends its buffer to all other processes one by one. Make sure to implement this with the name it deserves to the best, `naive_bcast` (or equivalently, expect this name in the variable `bcast_implementation_name` from the skeleton code).

Question 1

a) Implement a broadcast algorithm that simply sends the data residing at the root process 0 to all other processes one by one using `MPI_Send` and `MPI_Recv`.

b) Run the algorithm on a ring topology with $100Gpbs$ bandwidth and $1\mu s$ latency. Note the execution time.

c) Try to change the latency to $10\mu s$ and to $100\mu s$. How is does the runtime change? Is this expected?

d) Find the cost of this algorithm on a ring topology using the $\alpha - \beta$ model, assuming M being the message size, P being the number of processes, α corresponding to the latency cost of sending a message, and β representing the bandwith of each link in the interconnection network. Does this model justify your experimental results?

e) You just implemented the broadcast from the $\mathcal{P}_0$. How would you do the broadcast from an arbitrary process $\mathcal{P}_k$ (do not implement)? Does this require, however, a significant change in the implementation?

[1] My coffee preference is without sugar, just saying...

Part 2

(Don't) Use the ring!

You might have realized that sending all the messages from the root process $\mathcal{P}_0$ at each step could be inefficient, and that the naive algorithm exploits nothing related to the topology being ring. However, the ring may bestow unprecedented powers to thoso who know how to wield it! For example, once a process $\mathcal{P}_k$ receives the message on a ring, what is preventing it from simply passing it to its neighbor $\mathcal{P}_{k+1}$?

Question 2

a) Implement the described ring algorithm with the name `ring_bcast`. The process $\mathcal{P}_k$ should always expect and receive the message from its left neighbor $\mathcal{P}_{k-1}$, then store it in its local buffer, and finally pass it over to its right neighbor $\mathcal{P}_{k+1}$.

b) Run your implementation on the same topology with the latency $1\mu s$. How much of an improvement did you get? Are you happy? If not, try increasing the latency to $10\mu s$, and $100\mu s$, just as before. How does the performance compare to the naive broadcast this time?

c) Find the cost of this algorithm using the $\alpha - \beta$ model as before. Does it justify the results that you just obtained?

Part 3

One bite at a time...

One major problem with the previous ring broadcast algorithm is that it involves $P - 1$ send/receive rounds, in each of which only one of the links in the network is active. In order to make use of the bandwidth available in all other links at the same time, the idea of pipelining comes into play. Instead of sending M bytes of data in one chunk, we can divide it into chunks of size C, and send it in $\lceil M/C \rceil$ rounds that can be pipelined. For instance, once $\mathcal{P}_1$ receives the first chunk from $\mathcal{P}_0$, it can pass it to $\mathcal{P}_2$ and start receiving the second chunk from $\mathcal{P}_0$, and so on. If we continue in this manner, after $P - 1$ steps all links in the network will start to be actively used.

Question 3

a) Implement the described algorithm with the name `pipelined_ring_bcast`. In the executable, use the option `-c [chunk_size]` to fill in the varible `chunk_size`.

b) Experiment with the algorithm using the same ring network with latencies $1\mu s$, $10\mu s$, and $100\mu s$. Try to find the best chunk size for each latency, and note the broadcast times. How does the performance compare to `ring_bcast`?

c) Find the cost of this algorithm using the $\alpha - \beta$ model as before. Does it justify the results that you just obtained?

Part 4

Wait for it, wait for it!

So far we have only used `MPI_Send` and `MPI_Recv` in communications. One peculiarity about these routines is that they are "blocking" calls, meaning that once the process calls the function, it will "wait" and stall until this blocking send or receive operation entirely finishes. This is bad in terms of efficiency, because in this case the receive link, or the send link of each node in the ring will stay idle (depending on whether the node is performing a send, or a receive at that time, respectively). To

prevent this, we will employ asynchronous communication routines that allow us to hold two melons with one hand!

Question 4

a) Implement the described algorithm with the name `asynchronous_pipelined_ring_bcast`. You need to properly replace the `MPI_Send` calls with `MPI_Isend` routines, and use `MPI_Wait` on the communication handlers to make sure that the communications take place and finish correctly. You can keep the blocking `MPI_Recv` calls as they are, as establishing asynchronicity in one end should suffice to enable overlapping sends and receives.

b) Experiment with the algorithm using the same ring network with latencies $1\mu s$, $10\mu s$, and $100\mu s$. Is the algorithm always faster? If not, or what are the cases in which it runs slower? How would you explain this in any case?

Part 5

To recurse, or not to recurse, which one is the curse?

You managed to finish all the exercises and get this far? No-freaking-way! If you are procrastinating here not having finished the other tasks, go back to where you were and do some productive sh.t! Otherwise, that is wonderful, and here is your last exercise that serves as a marvelous closing scene to this TP session.

One brilliant and simple idea to perform a broadcast on a ring of size $P = 2^k$ for some integer $k > 1$ goes as follows. First, the process $\mathcal{P}_0$ sends the message to its "pair" $\mathcal{P}_{P/2}$, and make it responsible for broadcasting the message to the second half of the ring, i.e., to processes $\mathcal{P}_{P/2+1} \dots \mathcal{P}_{P-1}$. Afterwards, we can recursively apply this same idea to the first and the second halves of the ring (which are also rings of size $P/2$) to broadcast the data to the rest of the processes.

Question 5

a) Implement the described algorithm with the name `asynchronous_pipelined_bintree_bcast`. In the executable, use the option `-c [chunk_size]` to fill in the varible `chunk_size`. Try to make it pipelined, as the name suggests, if you have enough time.

b) Experiment with the algorithm using the same ring network with latencies $1\mu s$, $10\mu s$, and $100\mu s$. Try to find the best chunk size for each latency, and note the broadcast times. How does the performance compare to `asynchronous_pipelined_ring_bcast`? Does pipelining seem to help? Why, or why not?

c) Find the cost of this algorithm using the $\alpha - \beta$ model as before. Does it justify the results that you just obtained?

Part 6

Make sure to get a copy of your files as it might serve a good reference in the future!