ICS632 Assignment #3

For this assignment turn in an archive of a single directory with your code (all source files and Makefile or whatever scheme/script you want to you to build executables) and your report. For your report a `README` plain text file is absolutely fine, but if you want to turn in a `README.pdf` that's fine (no points awarded for prettiness, only for readability and content).

For this assignment you must have installed SimGrid (with SMPI enabled), so that `smpicc` and `smpirun` are in your path.

Overview

The objective of the assignment is to implement several versions of a broadcast collective communication, and to compare them in terms of performance on various platform setups.

We consider several kinds of physical platforms: a ring, a binary tree, a cluster with a crossbar switch, and a fat tree. I provide you with a Python script called `generate_xml_and_host_files.py` that generates all the XML platform files and hostfiles you need to run simulations. The script takes the number of processors as a single command-line argument. All network links in the generated platforms have 10 Gbit/sec bandwidth and 20 μsec latency.

Also provided is a skeleton of the program, `bcast_skeleton.c`, to evaluate the broadcast implementation performance when broadcasting 10^8 bytes. The root process is always rank 0. The program takes two optional arguments:

1. `-c <integer>`: The chunk size used to split the full message and achieve communication pipelining.
2. `-s <message string>`: An arbitrary string (but which contains no white spaces) that will be printed out by the program along with the wall-clock time of the broadcast. This may be convenient so that the output of your program is simpler to understand/analyze.

This program is to be invoked as:

```
smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:1Gf
        -np <num processes> -platform <XML platform file>
        -hostfile <host file> ./bcast_skeleton
        [-c <chunk size>] [-s <message string>]
```

You should try to compile/run the skeleton program as is to double check that your setup is all ok. As it is, the program will print some message to say that the broadcast was not successful (which makes sense since no broadcast is implemented).

All the code you have to write in this assignment goes in the portion of the code in between the "`// TO IMPLEMENT: BEGIN`" and "`// TO IMPLEMENT: END`" comments. You can create multiple copies of the skeleton code for each implementation, or use C `#ifdef` statements to maximize code reuse among the different versions, up to you. The important thing is that I can make sense of where your code is (you can describe this in your README file if not obvious), and that your executables have the correct names, as specified in the questions hereafter.

WARNING: If you machine doesn't have a lot of RAM, you may not be able to run the larger simulations. In this case, simply reduce the scale of the simulations in a reasonable manner. In extreme cases come talk to me so that I can give you a temporary account on a machine with a lot of RAM.

Question #1: Default, Naïve, and Ring Broadcast on a Ring (25%)

Using the skeleton program as a starting point, implement code and accompanying Makefile (or whatever scheme to build code, which I must be able to use) to generate three separate executables, `bcast_default`, `bcast_naive`, and `bcast_ring`:

- `bcast_default` uses `MPI_Bcast` to perform the broadcast.
- `bcast_naive` uses a simple for loop with a single `MPI_Send` call from the root process to each other process, which does an `MPI_Recv`.
- `bcast_ring` implements a (non-pipelined) ring broadcast in which process i receives from process $i-1$ and sends to process $i+1$ (using a single call to `MPI_Recv` and to `MPI_Send`).

All three programs should ignore the optional `-c` command-line argument even if it is present on the command-line.

Experimental Evaluation: Report the (simulated) wall-clock time of the three implementations using a **100-processor ring**. How far are your implementations from the default broadcast? You should observe that `bcast_ring` does not improve much over `bcast_naive`. What do you think the reason is? (To answer this question, you can instrument your code and run it for small platforms to see when events happen and try to understand what's going on – the `MPI_Wtime()` function is convenient to know the current date.)

Question #2: Pipelined Ring Broadcast on a Ring (25%)

One of the reasons why the default MPI broadcast is fast is likely that it does some pipelining of communication. Augment your code with an implementation that generates an executable `bcast_ring_pipelined` that uses the chunk size passed as a command-line argument to split the message into multiple chunks. The chunk size may not divide 10^8, in which case the last chunk will be smaller.

Experimental Evaluation: Run `bcast_ring_pipelined` using chunk sizes of 100000 (1,000 chunks), 200000 (500 chunks), 500000 (200 chunks), 1000000 (100 chunks), 2000000 (50 chunks), 10000000 (10 chunks), and 100000000 (1 chunk) on a **25-processor ring**, a **50-processor ring**, and a **100-processor ring**. What is the best chunk size for each of the four platforms? You should observe that the wall-clock time is minimized for a chunk size that is neither the smallest nor the largest. Discuss why you think that is.

Assuming the best chunk size is used for the 100-processor ring, by how much does pipelining help when compared to using a single chunk? How does the performance compare to that of the default MPI broadcast for that platform as seen in the previous question?

Question #3: Pipelined Ring Broadcast with Isend on a Ring (20%)

In your current code, while a process is sending data to another it cannot do anything else. Augment your code with an implementation that generates an executable `bcast_ring_pipelined_isend` that is similar to the one in the previous question but attempts to overlap receiving with sending: post a send via `MPI_Isend`, then do a receive via `MPI_Recv`, and once the receive completes then check on completion of the send via `MPI_Wait`. Ensure that no "dangling" send remains unchecked. The hope is that more network communications can happen in parallel.

Experimental Evaluation: Run the same experiments as in Question #2 but for `bcast_ring_pipelined_isend`. Discuss the results. By how much does the use of `MPI_Isend` help? Is the best chunk size the same? Does it look like the relative benefit of using `MPI_Isend` increases or decreases with platform scale? How much faster than the default broadcast is your implementation on the 100-proc platform?

Question #4: Binary Tree Broadcast on a Tree (20%)

It turns how that some real-world supercomputers have provided tree network topologies. Augment your code with an implementation that generates an executable `bcast_bintree_pipelined_isend` that implements the broadcast along a binary tree, splitting the message into chunks for pipelining, and using `MPI_Isend` to increase communication throughput. The chunk size may not divide 10^8, in which case the last chunk would be smaller. Also the binary tree won't be complete if the number of processors is not of the form $2^n - 1$. Use a level order traversal numbering of the nodes in your tree: rank 0 has rank 1 as its left child and rank 2 as its right child, rank 1 has rank 3 as its left child and rank 4 as its right child, etc. (This is the order used to number processors in the binary tree platform file we use in the experimental evaluation.)

Experimental Evaluation: Report the (simulated) wall-clock time of `bcast_default`, `bcast_ring_pipelined_isend`, and `bcast_bintree_pipelined_isend` on a **100-processor binary tree**. For `bcast_ring_pipelined_isend` and `bcast_bintree_pipelined_isend` use chunk sizes of 100000 (1,000 chunks), 200000 (500 chunks), 500000 (200 chunks), 1000000 (100 chunks), 2000000 (200 chunks), 10000000 (10 chunks), and 100000000 (1 chunk). How does `bcast_bintree_pipelined_ring_isend` compare to the default MPI broadcast? What about `bcast_ring_pipelined_ring`? What about the best chunk sizes?

Question #5: "Real" Networks (10%)

Run `bcast_default`, `bcast_ring_pipelined_isend`, and `bcast_bintree_pipelined_isend` on a **64-node cluster with a crossbar switch** and on a **64-node fat tree**. For the pipelined algorithms use chunk sizes of 100000 (1,000 chunks), 200000 (500 chunks), 500000 (200 chunks), 1000000 (100 chunks), 2000000 (200 chunks), 10000000 (10 chunks), and 100000000 (1 chunk). How do the algorithms compare on each platform (assuming the best chunk size is used for the pipelined algorithms)?

Overall, does it seem like it's a good idea to use the default MPI broadcast? Or should one implement one's own? Note that the default broadcast has to pick a particular chunk size while we're "cheating" by looking for the best chunk size for our implementations!

Beyond this assignment towards a project?

The "inverse" of the broadcast is the *reduce* operation, by which an associative and commutative operator is applied to data items located at each processor (with the final result available, say, at the process of rank 0). An interesting project is to explore efficient reduce algorithms, i.e., efficient orchestration of communications (the time to apply the operator can be ignored for simplicity). This project idea is inspired by work by several researchers, e.g., Bradley Lowery at the Univ. of Colorado, Denver.

Here is a possible setting:

- Processors are interconnected via a crossbar switch

- At a give time step a process (one per processor) can either send or receive, but not both
- There are m data items at each process, and these can be split into whatever chunks
  - A seemingly simply, yet still interesting case, is to assume uniform chunks.
  - Interesting effects may be achieved by allowing chunks of different sizes.

Here are three approaches that could be compared:

- Binomial tree without chunking
  - Step 0: Each odd numbered process sends to its left neighbor
  - Step 1: Each even-numbered process sends to its 2nd left neighbor
  - ...
- Pipeline tree
  - Chunk the data
  - Step 0: process n sends chunk 0 to process $n-1$
  - Step 1: process $n-1$ sends chunk 0 to process $n-2$, and process n sends chunk 1 to process $n-1$
  - ...
- Binary tree
  - Chunk the data
  - Step 0: processes n and $n-1$ send chunk 0 to process $n-2$ (two communications that happen in sequence), processes $n-3$ and $n-4$ send to process $n-5$, ...
  - Step 1: processes n and $n-1$ send chunk 1 to process $n-2$ (two communications that happen in sequence), processes $n-3$ and $n-4$ send to process $n-5$, ...
  - ...

and many others. The comparison could be both analytical (wall-clock time as a function of the number of processors, the number of messages, the chunk size, the communication latency, and the communication bandwidth) and simulation-based.

In his research (see for instance slides at: `http://perso.ens-lyon.fr/evelyne.blesle/pittsburgh/SLIDES/Brad_Lowery.pdf`), B. Lowery proposes another strategy he calls a *Greedy Tree*, which is optimal for uniform segmentation (based on older work). It would be great in this project to try to implement the greedy tree and to verify his findings.