Solutions to exercises, week 5

March 1, 2017

Exercise 1

This problem is trivial. Just remember that we always operate in base two in computer science. You should understand why.

Exercise 2

The algorithm and an in-depth description also appears in the course book, pages 155-157.

A brief explanation of AND / XOR

The bitwise **AND**/**XOR** operators work as follows: given two integers, convert them to binary format, e.g. 14 and 15 become 1110 and 1111 respectively. The i'th bit of x **AND** y is the multiplicative product of the i'th bits of x and y (that is, the product is 1 if they're equal, otherwise it's 0). The i'th bit of x **XOR** y is 1 if the bits are different from each other, otherwise it's zero. Hence, using 14 and 15 as examples,

```
    1110    |       1110
AND 1111    |  XOR  1111
--------    |  ---------
=   1110    |  =    0001
```

In C and Python, the bitwise operators are `&` and `^` respectively.

Two properties of the XOR operator

First, **XOR** applied twice cancels, i.e.

```
x ^ y ^ y = x;
```

In the algorithm below, we'll use it to switch back and forth between `virtual_id`s and `id`s.

Second, it's a nice way to model the "binary format distance" between numbers. For example, if we compare 10 with 5,

```
1010 ^ 0101 = 1111;
```

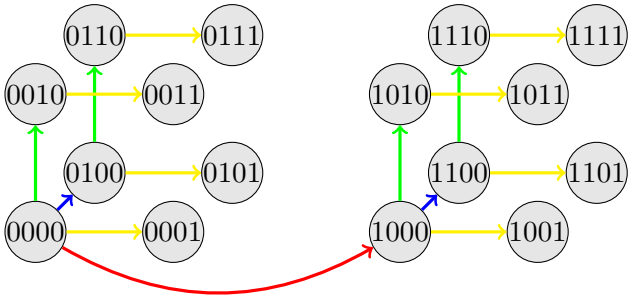
there are four ones in the result, so we have to travel along four dimensions on a hypercube to get from node 5 to node 10.

The explanation

This explanation is just meant as an aid, so you should go through the algorithm on your own. If we use the standard four-dimensional hypercube setup as an example, with *source = 0*, then the send/receive routine is just the song and dance demonstrated in the picture below. It should be familiar from the course, or at least you should be able to convince yourself that this is a clever routine indeed.

Now, the first step of the algorithm is obtaining `my_virtual_id`, defined as `my_id ^ source`. Note carefully that if `my_id = source`, then `my_virtual_id` becomes 0 (0000 in binary), and that the further away `my_id` is on the hypercube from the source, the more ones appear in `my_virtual_id` (again in binary format). The upshot is that we may as well assume that the source is process 0000. The picture was right all along!

The broadcast is completed in $d = 4$ iterations (i.e. $i = 3, 2, 1, 0$), and during each iteration we'll send along one dimension (first along the fourth, then the third, and so on), see below. A comparison vs. the `mask` variable is used to check whether a given node is about to participate in sending/receiving of data, and finally, if so, we determine whether this node is on the sending or the receiving end. Looking at the picture, you should be able to convince yourself that the node should *send* if bit i of the node's virtual id is 0, and *receive* if it's 1.



In the picture, the red arrow represents the first send, blue the second, then green, then yellow.

Exercise 3

Assume now that the number of processes p is *not* a power of 2. We'll want to stick as closely to the original algorithm as possible. With p processes, we have $d = \lceil \log p \rceil$, corresponding to a d-dimensional hypercube with the final nodes deleted.

If we first try to go through the algorithm with *source* = *0*, we shouldn't have to change *that* much, because the missing nodes are all at the end. Hence, it suffices to add a check before sending to confirm that the destination node exists.

If on the other hand we were to try, say, *source* = *4*, and $p = 12$, and calculate the `virtual ids` of the missing nodes,

```
1100 ^ 0100 = 1000;
1101 ^ 0100 = 1001;
1110 ^ 0100 = 1010;
1111 ^ 0100 = 1011;
```

then we see that the algorithm fails completely, because it *needs* those missing nodes.

Having said that, we'll use the following fix: replace the way we compute `virtual_ids`. Specifically, we'll let the virtual identities be the original identities, but shifted to the left by `source` (modulo p). That is,

```
my_virtual_id = ((my_id - source) + p) % p;
```

Given a virtual identity, we recover the original id by shifting it by *source* units to the right,

```
my_id = (my_virtual_id + source) % p;
```

In this way we ensure that each the virtual ids $0, 1, \cdots, p-1$ corresponds to an actual process.

In summary, we require two changes to the algorithm:

- First, we replace the way we compute the virtual ids by using a left-shift.
- Next, we make a slight addition to the send routine, wherein we simply check that the destination exists (i.e. that `virtual_dest < p`).

There's a downside to this, as you might have guessed: the remainder operator isn't really compatible with how the hypercube works. For example, if we were to try the algorithm with $p = 16$ and *source = 1*, and observe the send/receive output given by process 0, we might get something like

```
doing broadcast rank 0 out of 16 processes with source 1
doing nothing
doing nothing
doing nothing
Message received from process 15
```

So we see that the data must take potentially quite long detours. Arguably this structure is more ring-like than hypercube-like.

Exercise 4

See `ex4.c`. As expected, a reduction routine is really just a broadcast in reverse, so this is just Exercise 3 backwards.

You can also find an algorithm implementing an all-to-one reduction in the course book on page 158.

Additional exercises

See programs in the folder named `additional`. You can use monochrome pictures from the course page to test, and you should have no trouble unravelling the mysteries of image flipping.

Using the embedded library in C is kind of weird and complicated, as you've no doubt seen. The solution in this code is mostly taken from the template for the first mandatory assignment. It's a two-step implementation, where you first create something called a *static library*, and then linking it to your code via by passing a `-l<LIBRARY HERE>` flag to the compiler. You can run the Makefile like this:

```
>make library
>make main
```