Assignment #3 (HPC . ICS632 Oct. 2015) ----------- Ehsan Kourkchi

To compile:

```
$ make all

smpicc  bcast_skeleton.c -o bcast_default -D MODE=0
smpicc  bcast_skeleton.c -o bcast_naive -D MODE=1
smpicc  bcast_skeleton.c -o bcast_ring -D MODE=2
smpicc  bcast_skeleton.c -o bcast_ring_pipelined -D MODE=3
smpicc  bcast_skeleton.c -o bcast_ring_pipelined_isend -D MODE=4
smpicc  bcast_skeleton.c -o bcast_bintree_pipelined_isend -D MODE=5
```

To clean the previous compilation:
```
$ make clean
```

Question 1: Default, Naïve, and Ring broadcast on a ring

platform: **100-processor ring**

Table 1

Method	**Time (s)**
bcast_default	2.992
bcast_naive	9.859
bcast_ring	8.778

```
example$ smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:1Gf -np
100 -platform ring_100.xml -hostfile hostfile_100 ./bcast_default
```

- Why is not **bcast_ring** that much better than **bcast_naive**?
To answer this question, we run the same code using only 5 processors and we record the times when the send and receive processes are done.

As seen in tables 2 and 3, each send process has to wait until the pack of the sent data is completely received by the target processor.
So, at each time only one send process can be carried out.

In the naïve implementation, everything is send by processor 0.
In the ring implementation, each pack is sent to the neighbor processor.

So, the time of message passing is not dominated by network latency and it is dominated only by the sending-receiving time overhead. It really doesn't matter which processor is sending to which one. The important point is that, at each time, all processes are blocked until a "send" process is completed.

Table 2: Naïve

Processor No.	Time of send-begin	Time of send-end	From … To	Delta time
1	0.000	0.085	0 → 1	0.085
2	0.085	0.171	0 → 2	0.086
3	0.171	0.257	0 → 3	0.086
4	0.257	0.343	0 → 4	0.086

Total run-time = 0.557

Table 3: Ring

Processor No.	Time of send to	Time of receive	From … To	Delta time
1	0.000	0.085	0 → 1	0.085
2	0.914	0.999	1 → 2	0.085
3	0.999	1.084	2 → 3	0.085
4	1.084	1.169	3 → 4	0.085

Total run-time = 0.641

As seen in tables above, the actual time of sending package of data is fairy the same when we sent 0 → 2 or 1 → 2

Question 2: Pipelined Ring Broadcast on a Ring

```
example > smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:1Gf -np 25 -platform ring_25.xml -hostfile hostfile_25 ./bcast_ring_pipelined -c 100000000
```

Table 4 – Processing time (s) – MPI-Send

No. of processors	1,000 chunks	500 chunks	200 chunks	100 chunks	50 chunks	10 chunks	1 chunks
25	13.359	7.727	4.425	2.960	2.090	**0.958**	2.352
50	26.573	14.973	7.582	4.429	1.805	**1.190**	4.491
100	51.820	27.962	11.767	3.571	2.154	**1.673**	8.778

** The best processing times are highlighted.*

So, theoretically if everything goes well, the optimum number of chunks would be

$$\sqrt{m(p-2)\beta/\alpha}$$

where, α is the latency and β is the inverse of the bandwidth, when it is in terms of Byte/s. p is the number of processors, in this case 100 for example. We approximate p-2 with p since p is large. In this problem, bandwidth is 100Gbs, therefore β=0.8 * 10^-9 seconds.
α is also 20μs = 2* 10^-5 seconds.

Replacing all of these values gives the theoretical best number of chunks =~ 600 to send 10^8 bytes.
This is the theoretical value, assuming that buffering and send/receive process does not take any time.

In practice, these factors are important. For this problem, for all platforms, the best number of chunks is 10 chunks (10,000,000 bytes each).

Wall-clock time is:

$$(p-2+r)(\alpha+\beta m/r)$$

So, if we increase the number of chunks, r, the β term in second para thesis becomes smaller and comparable to the network latency. In another word, increasing the number chunks (decreasing the message size), latency plays an important role and becomes more dominant.

improvement in processing time compared to single-chunk process:

```
p = 25    (2.352-0.958) / 2.352    = 59%
p = 50    (4.491-1.190) / 4.491    = 74%

p = 100   (8.778-1.673) / 8.778    = 81%
```

In the case of p=100, the default MPI-broadcast, results in T=2.992
pipelining improves it a little bit:

improvement ratio = (2.992-1.673)/2.992 = 44%

Question 3: Pipelined Ring Broadcast with Isend on a Ring

```
example > smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:1Gf -np
25 -platform ring_25.xml -hostfile
hostfile_25 ./bcast_ring_pipelined_isend -c  100000000
```

Table 5 - Processing time (s) - MPI-Isend

No. of processors	1,000 chunks	500 chunks	200 chunks	100 chunks	50 chunks	10 chunks	1 chunks
25	5.891	3.098	1.557	1.087	0.953	**0.706**	2.261
50	11.716	6.016	2.734	1.677	1.248	**0.935**	4.394
100	23.380	11.861	5.021	2.876	1.808	**1.436**	8.760

** The best processing times are highlighted.*
*** The last columns show the best time improvement when using MPI-Isend over MPI-Send*
**** For smaller chunks this improvement can be as large as ~50%*

Table 6 - Improvement percentage

# of processors	1,000 chunks	500 chunks	200 chunks	100 chunks	50 chunks	10 chunks	1 chunks
25	56	60	65	63	54	26	04
50	56	60	64	62	31	21	02
100	55	58	57	19	16	14	00

**This table shows the relative benefit of using MPI-Isend over MPI-Send (in per cent) for different number of chunks. As seen, the relative benefit decreases as we increase the number of processors. The reason is that increasing the number of processor, increases the overhead time of send/receive processes, which is then comparable to the total network traveling time. So MPI-Isend is a bit less of help in dealing with huge number of nodes.*

My observation is that for large chunks, using MPI-Isend does not help to have a shorter running time. But for smaller chunks, this method improves the efficiency. The improvement of using MPI-Isend is ~50% for small chunks (50-1000 chunks).

Probably the reason is that for larger number of chunks, latency of the network is important and is dominant. When we are using MPI-Isend, we try to do some processes at the time data is traveling across the network and we use that dead-time. For bigger chunk-sizes, latency is NOT dominant and therefore this method does not help that much. Buffering overhead and internal processes to send big data is more time consuming compared to the time data is traveling across the network in that case.

improvement in processing time compared to single-chunk process:

```
p = 25    (2.261-0.706) / 2.261    = 69%
p = 50    (4.394-0.935) / 4.394    = 79%

p = 100   (8.760-1.436) / 8.760    = 84%
```

In the case of p=100, the default MPI-broadcast, results in T=2.992
pipelining improves it a little bit:
improvement ratio = (2.992-1.436)/2.992 = 52%

Question 4: Binary Tree Broadcast on a Tree

bcast_default → time: 1.481 (s)

Table 7: 100-processor binary tree

MODE	1,000 chunks	500 chunks	200 chunks	100 chunks	50 chunks	10 chunks	1 chunks
bcast_ring_pipelined_isend	3.390	2.139	1.604	1.519	**1.459**	2.093	**8.721**
bcast_bintree_pipelined_isend	0.869	0.637	0.497	0.452	**0.431**	0.436	**0.738**

Using 50 chunks, **bcast_ring_pipelined_isend** running time is almost the same as that of **bcast_default**.

It's not surprising that **bcast_bintree_pipelined_isend** (using 50 chunks) is 3 times more efficient than the other two methods. This shows how having a prior knowledge about the network topology can help to write better codes with better performances.

It's interesting that even with one chunk, bintree-pipelined is ~12 times faster.

The optimizing chunk-size is not the same as before, and dividing the data into 50-chunks sounds to be more efficient. It' obvious that changing the topology of the network, changes the pattern of data flow. In this case, the total number of send and receive that each processor needs to do is much less than the ring topology and therefore MPI-Isend saves more time when using smaller data sizes.

Question 5: "Real" Networks

64-node Crossbar

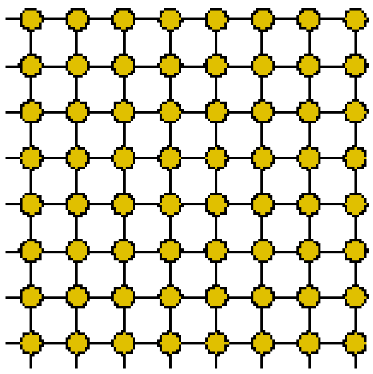
```
example > smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:1Gf -np
64 -platform cluster_crossbar_64.xml  -hostfile hostfile_64
./bcast_default
```

bcast_default → 0.419 s

Table 8: 64-node Crossbar

MODE	1,000 chunks	500 chunks	200 chunks	100 chunks	50 chunks	10 chunks	1 chunks
bcast_ring_pipelined_isend	0.807	0.576	0.450	**0.428**	0.457	0.860	5.601
bcast_bintree_pipelined_isend	1.471	1.005	0.728	0.637	**0.595**	0.602	1.102

For the best cases, the best chunk number is 50, however **bcast_default** has a better performance. The performance of both ring_pipelined and **bintree_pipelined** seem to be the same for 50 chunks.



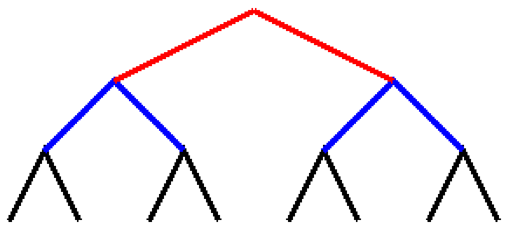
64-node Fat-tree

```
example > smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:1Gf -np
64 -platform fattree_64.xml  -hostfile hostfile_64 ./bcast_default
```

bcast_default → 0.505 s

Table 9: 64-node fat tree

MODE	1,000 chunks	500 chunks	200 chunks	100 chunks	50 chunks	10 chunks	1 chunks
bcast_ring_pipelined_isend	2.227	1.301	0.760	0.598	**0.531**	0.878	5.606
bcast_bintree_pipelined_isend	4.404	2.836	1.804	1.493	**1.365**	1.375	2.107



ring_pipelined_isend is faster than **bintree-pipelined** in this case fro 50 chunks.

The reason is that when traversing the nodes along a binary tree, as we go down, the physical distance of a parent to its children increases, by the nature of a fat-tree topology. So, down the binary tree, any parent needs to have more hops in order to communicate to its children. That's why bintree_pipelined is no longer efficient compared to **ring_pipelined.**

Surprisingly, **bcast_default** has a better performance in both case and in general on “real” networks. It seems that implementing specific codes for each of these topologies is useless.

Now, the questions is that, how **MPI_bcast** optimizes the chunk size for its own pipelining?

Another question is that which of these topologies is better (ft-tree or crossbar)? Having 64 nodes, the maximum distance between a pair of nodes is 6 for the fat-tree, and 14 for the crossbar. However, in a fat-tree the bandwidth is more limited, unless for different layers, different bandwidth is used. Also switch performance in a fat-tree is important. For higher level, faster switches are required.