Assignment #3 Message Passing

Xun Zhu

Activity 1: Naive Broadcast and Ring Broadcast

The two algorithms are implemented in the `main.c` file.

What are the (simulated) wall-clock time of the three implementations on the 50-processor ring?

Default Bcast	Naive Bcast	Ring Bcast
1.073	4.194	4.188

How far are your "by hand" implementations from the default broadcast?

It's pretty far—both took almost 4x as long.

You may observe that `ring_bcast` does not improve a lot over `naive_bcast`, which should be surprising to you. After all, `naive_bcast` sends long-haul messages while `ring_bcast` doesn't. What do you think the reason is? To answer this question, you can instrument your code and run it on smaller rings to see when events happen and try to understand what's going on. Given that we're using simulation, you should take advantage of it and experiment with all kinds of platform configurations to gain understanding of the performance behavior. For instance, you can modify link latencies and bandwidths. The `MPI_Wtime` function is convenient to determine the current (simulated) date. This function returns the date as a double precision number (and is in fact already used in `bcast_skeleton.c`).

Figure 1 shows the event timeline for a 5-node ring network. As we can see the network latency doesn't take nearly as much time than does the sending/receiving the big chunk of data itself. This results in a negligible improvement eliminating long distance communication. Figure 2 is a similiar timeline, but with the link latency greatly exaggerated (10,000x, to 100,000us.) As we can see the improvement is much more evident.

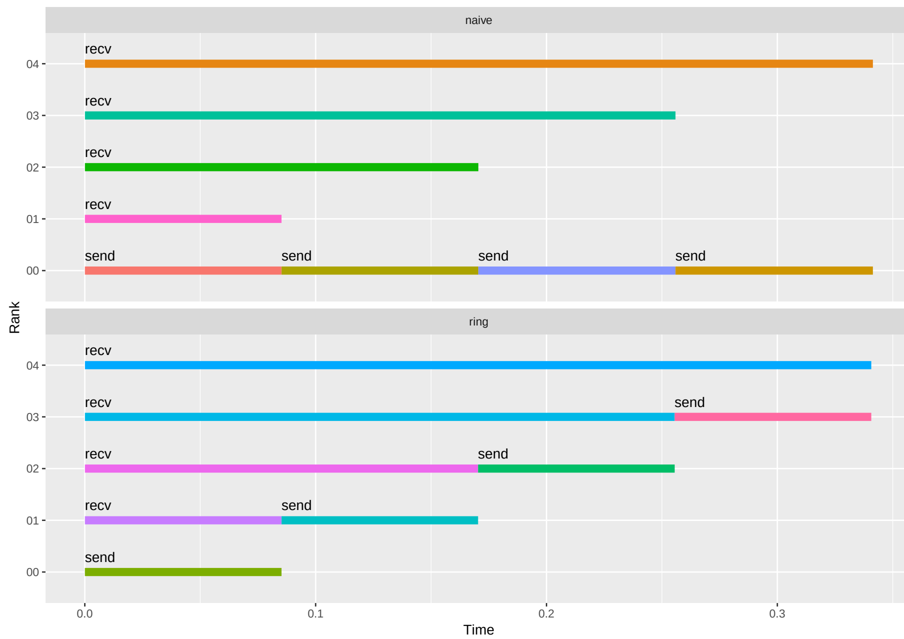

Figure 1: Event timeline for when link latency is at 10us.

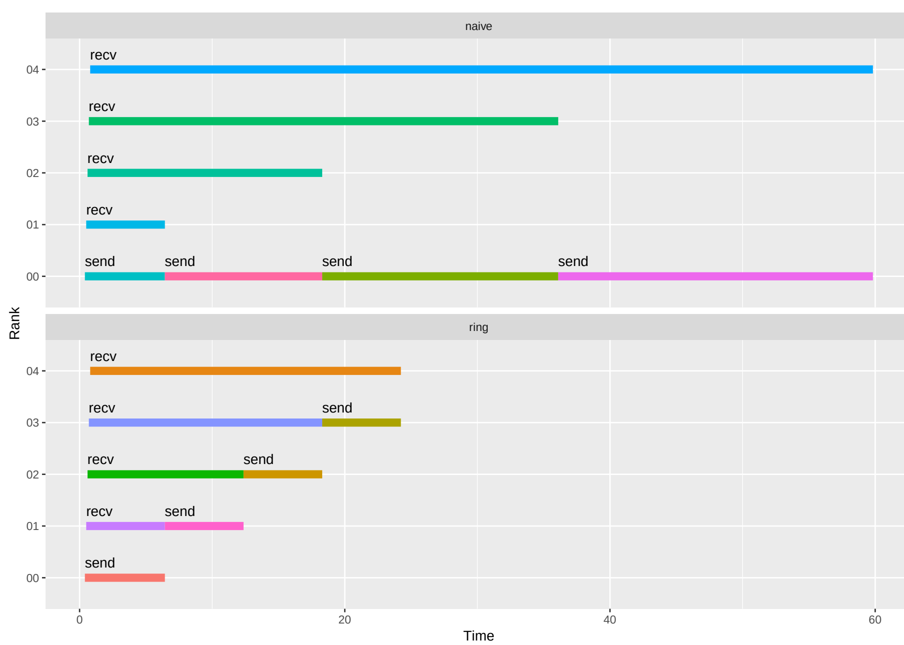

Figure 2: Event timeline for when link latency is at 100,000us.

Activity 2: Pipelined Broadcast

The algorithm is implemented in the `pipelined_ring_bcast` branch in the `main.c` file.

What is the best chunk size for each of the three platforms?

As shown in Table 1, the best chunk size is always 1,000,000 for all three platforms.

Number of nodes	Chunk size	Sim. total time
20	100000	0.408
20	500000	0.227
20	1000000	**0.211**
20	5000000	0.250
20	10000000	0.320
20	50000000	0.896
20	100000000	1.619
———	———	———
35	100000	0.416
35	500000	0.236
35	1000000	**0.226**
35	5000000	0.316
35	10000000	0.450
35	50000000	1.537
35	100000000	2.897
———	———	———
50	100000	0.429
50	500000	0.249
50	1000000	**0.241**
50	5000000	0.383
50	10000000	0.580
50	50000000	2.177
50	100000000	4.175
———	———	———
100	100000	0.505

Number of nodes	Chunk size	Sim. total time
100	500000	0.324
100	1000000	**0.317**
100	5000000	0.603
100	10000000	1.014
100	50000000	4.311
100	100000000	8.436

Table 2: The simulated total running time of various combinations of numbers of nodes and chunk sizes. The best time with a given number of nodes is bolded.

Does this chunk size depend on the platform size?

No.

Althought this is is a close call, as chunk size 500,000 is always very close in all three 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.

When the chunk size is too large, we spend most of the time starting the first chunk and waiting for the last chunk to finish. When the chunk size is too small, the actual sending/receiving doesn't take long but most of the time is spent on waiting for the network delay. And since the chunk size being small implies that we have a large number of chunks, the total delay incurred is also going to be large.

For a 100-processor ring, with the best chunk size determined above, by how much does pipelining help when compared to using a single chunk?

From Table 1, we see that it improved the total simulated running time from 8.436 to 0.317, which is a 26.6x speed-up.

How does the performance compare to that of the default MPI broadcast for that platform as seen in the previous question?

The `default_bcast` algorithm took 2.204 seconds. Comparing to that, our `pipelined_ring_bcast` took 0.317, which is a 6.95x speed-up.

What do you conclude about the use of pipelined communications for the purpose of a ring-based broadcast on a ring-shaped physical platform?

Choosing the right chunk size, it can achieve even better performance than the default `MPI_Bcast` function. This is a good example that hand-written messaging algorithms that respects the topology of the network can be faster than the highly-opitimized generic algorithm.

Activity 3: Asynchronous Communication

The algorithm is implemented in the `asynchronous_pipelined_ring_bcast` branch in the `main.c` file.

Is the best chunk size the same as that for `pipelined_ring_bcast`?

Number of nodes	Chunk size	Sim. total time
50	100000	0.228
50	500000	**0.141**
50	1000000	0.146
50	5000000	0.300
50	10000000	0.502
50	50000000	2.134
50	100000000	4.175

Table 3: Same as Table 1, but for the asynchronous version of the algorithm.

No. As shown in Figure 2, the best chunk_size is 500,000. However, 1,000,000 is extremely close.

When using the best chunk size for each of the two implementations (which may be the same if the answer to the previous question is "no"), does the use of `MPI_Isend` help? By how much?

Yes. The improvement is from 0.241 to 0.141 seconds, which is a 1.71x speed-up.

Is asynchronous_pipelined_ring_bcast faster than default_bcast or slower? By how much?

Faster, by 7.61x (from 1.073 to 0.141 seconds.)

What do you conclude about the use of asynchronous communications for the purpose of a ring-based broadcast on a ring-shaped physical platform?

It is an improvement over the already quite fast pipeline algorithm, further showcasing the potential of topology-specific optimizations.

Activity 4: Logic Binary Tree

The algorithm is implemented in the `asynchronous_pipelined_bintree_bcast` branch in the `main.c` file.

On the 50-processor ring platform, how does `asynchronous_pipelined_bintree_bcast` compare to `asynchronous_pipelined_ring_bcast`? Is it unexpected?

`asynchronous_pipelined_bintree_bcast` took 2.185 seconds, comparing to `asynchronous_pipelined_ring_bcast`'s 0.141 seconds, it took 15.5x as long.

This is completely expected as most of the sending/receiving operations will become long-distance communication instead of neighbor node communication.

On the 50-processor binary tree platform, how do the three implementations compare? Does it seem worth it to implement your own binary tree broadcast on a binary tree physical platform?

`asynchronous_pipelined_bintree_bcast` took 0.103 seconds, whereas `asynchronous_pipelined_ring_bcast` took 0.959 seconds, and `default_bcast` took 1.073 seconds.

Yes. The improvement is significant.

Activity 5: Logic Binary Tree

Report the (simulated) wall-clock time of `default_bcast`, `asynchronous_pipelined_ring_bcast`, and `asynchronous_pipelined_bintree_bcast`, on the following two platforms:

A 64-processor cluster based on a single crossbar switch: `cluster_crossbar_64.xml`.

Default bcast	Asynchronous pipelined ring bcast	Asynchronous pipelined bintree bcast
0.205	0.213	0.223

A 64-processor cluster with a shared backbone: `cluster_backbone_64.xml`.

Default bcast	Asynchronous pipelined ring bcast	Asynchronous pipelined bintree bcast
3.230	3.206	3.124

Overall, does it seem like it's a good idea to use the default MPI broadcast on these more standard platforms, or should one implement one's own? Note that the default broadcast has to pick a particular chunk size while we are "cheating" by picking an empirically good chunk size for our implementations!

Looks like the manually optimized algorithms are no faster than the default bcast on standard platforms, and therefore using the default is good enough.

Running on the Cray

The following results are based on running the program on 2 nodes, with a total of 40 tasks. The result time is based on the `MPI_Wtime` calls.

| Default bcast | 49.094 | | Naive bcast | 55.939 | | Async ring bcast | segfault | | Async bintree bcast | segfault |

Despite my effort, I couldn't find the reason why the latter two methods always gave segfault when run on CRAY. Reducing the data size didn't help. I used `openmpi/1.10.0` because the mpich on CRAY is broken.