NBC Library 1.0 Performance Analysis and Application

“It is human nature to find ways to convert synchronous communications into asynchronous forms.” – The Road Ahead, by Bill Gates, Nathan Myhrvold, and Peter Rinearson

Introduction

NBC Library (LibNBC) is an implementation of the non-blocking equivalents of the collective operations in MPI-2, developed by Torsten Hoefler from the Indiana University, who is also a participant of the MPI-2 Forum. It is based on the MPI standard itself (in particular MPI_Isend/Irecv), and release 1.0 is written in C++ using POSIX Threads. To the best of my knowledge, it’s the only existing portable library for non-blocking communication, and the author’s publications demonstrated its promise. However, the testing and analysis reported here also exposed significant problems and inconvenience in the usage.

The main content of this report is the performance testing and analysis of a selection of its non-blocking collective operations, named with the convention NBC_I(operation name) (e.g. NBC_Ibcast, NBC_Ireduce...). The tests were performed on the Nyx cluster using the combination of Open MPI 1.0.2 and pgi 6.1. If not specified otherwise, the tests were performed with 64 processor run and with 262144 bytes (32768 double precision floating point numbers, each occupies 8 bytes in the testing environment specified above) as the size of the individual messages involved, with the implication that the size of the receive buffer of operations like NBC_Igather and NBC_Ialltoall will be p*262144 bytes. In particular, the walltime by which all the processors finish the specific collective operation followed by a large number of floating-point operations (flop’s) is compared to the equivalent using the standard MPI operations. In cases where it’s relevant, the latency of the NBC in the blocking setting is also compared with that of MPI.

This report is capped off by the result of applying LibNBC to the dynamic network problem of Homework 4 and a short conclusion. Some benchmarks of LibNBC generated by the benchmark suite provided by the author (NBCBench 1.0) are given in the Appendix. As we can see, some of them do not reflect the more realistic picture of LibNBC’s performance.
Performance Analysis

NBC_Ibcast:

As the name indicates, this is the non-blocking version of MPI_Bcast. Although NBCBench 1.0 shows that it compares favorably with MPI_Bcast in terms of the CPU time required for the operation, tests in more realistic settings uncovered a major flaw in its implementation: it does not properly release the CPU from the operation, even after the broadcast is finished:

![Walltime (s) vs. # of Flop’s following Bcast Operation](image)

For these tests, 64 processors run the sequence MPI_Barrier – MPI_Wtime (for root) – broadcast – floating-point flop’s – NBC_Wait (for Ibcast) – MPI_Barrier - MPI_Wtime (for root). As indicated by the walltimes, the flop’s after the NBC_Ibcast are drastically slower than the ones after the MPI_Bcast well after the broadcast is finished. This discrepancy persists even when the Flop’s are placed after NBC_Wait or after MPI_Barrier. Although the NBC_Ibcast operation itself is efficiently implemented with minimal overhead compared to MPI_Bcast, this flaw makes it unusable.

Some “dragger” tests were also carried out to determine whether NBC_Ibcast is properly non-blocking. For these tests, Only the processor with rank 1 carries out
10 million Flop’s, between the first MPI_Barrier and the NBC_Ibcast. The times reported by MPI_Wtime by which each processor finishes the NBC_Ibcast statement and the NBC_Wait statement right after it are then collected separately for analysis. The analogous tests were also carried out for MPI_Bcast for comparison.

Times after NBC_Ibcast:

\[ p = 64 \]

Size of the Ibcast = 32768 double's w/ initialization & rank 0 write-over, dragger test NOWAIT

...  
Walltime = 0.093039  
Finish times for procs:

\[
\begin{align*}
9.08375e-05 & \quad 0.091691 & \quad 0.00402784 & \quad 0.00401998 & \quad 0.00419998 & \quad 0.00423288 & \quad 0.00561905 \\
0.00563097 & \quad 0.00455284 & \quad 0.00455785 & \quad 0.00548291 & \quad 0.00547791 & \quad 0.00476193 & \quad 0.00482702 \\
0.00501585 & \quad 0.0050199 & \quad 0.0053699 & \quad 0.00539994 & \quad 0.00464296 & \quad 0.00465298 & \quad 0.00529695 \\
0.0053339 & \quad 0.00545883 & \quad 0.00546384 & \quad 0.00619197 & \quad 0.00618482 & \quad 0.00538397 & \quad 0.005373 \\
0.00522494 & \quad 0.00528998 & \quad 0.00517297 & \quad 0.00516987 & \quad 0.00628185 & \quad 0.00476503 & \quad 0.00579095 \\
0.00595689 & \quad 0.00734091 & \quad 0.00867701 & \quad 0.00788689 & \quad 0.00652289 & \quad 0.007761 & \quad 0.0065999 \\
0.00709295 & \quad 0.00650287 & \quad 0.00456691 & \quad 0.00613999 & \quad 0.00640488 & \quad 0.00889182 & \quad 0.00590682 \\
0.00658798 & \quad 0.00640798 & \quad 0.00795388 & \quad 0.0063928 & -0.00266504 & \quad 0.00589395 & \quad 0.00580287 \\
0.00546384 & \quad 0.00594282 & \quad 0.00605488 & \quad 0.00599003 & \quad 0.00617099 & \quad 0.00738382 & \quad 0.00620484 \\
0.00728703 & \quad & \quad & \quad & \quad & \quad & \quad 
\end{align*}
\]

Times after NBC_Wait:

\[ p = 64 \]

Size of the Ibcast = 32768 double’s w/ initialization & rank 0 write-over, dragger test

Blocking NBC Walltime = 0.0210428  
Blocking NBC Walltime = 0.00132704  
Blocking NBC Walltime = 0.0014081  
Walltime = 0.0912139  
Finish times for procs:

\[
\begin{align*}
8.29697e-05 & \quad 0.089983 & \quad 0.00374794 & \quad 0.0936749 & \quad 0.00381684 & \quad 0.0937381 & \quad 0.00481486 \\
0.0947399 & \quad 0.003319958 & \quad 0.0895591 & \quad 0.00469398 & \quad 0.0946069 & \quad 0.00426102 & \quad 0.094177 \\
0.00434995 & \quad 0.0942719 & \quad 0.00444794 & \quad 0.0943429 & \quad 0.00380301 & \quad 0.093729 & \quad 0.00437307 & \quad 0.094331 \\
0.00462604 & \quad 0.0945721 & \quad 0.00390005 & \quad 0.0937729 & \quad 0.00457287 & \quad 0.094492 & \quad 0.00443506 & \quad 0.094363 \\
0.00440001 & \quad 0.0943179 & \quad 0.00509286 & \quad 0.0947769 & \quad 0.00482488 & \quad 0.0960629 & \quad 0.00743794 & \quad 0.09673 \\
0.00531697 & \quad 0.0966721 & \quad 0.00534987 & \quad 0.09567 & \quad 0.00525904 & \quad 0.0936379 & \quad 0.00514507 & \quad 0.095046 \\
0.00546288 & \quad 0.0976999 & \quad 0.00510001 & \quad 0.095278 & \quad 0.00539589 & \quad 0.096776 & \quad 0.00538588 & \quad 0.075633 
\end{align*}
\]
0.00511408 0.0948949 0.00375605 0.094316 0.00514603 0.0949779 0.00504994
0.0949271 0.0065999 0.09516

Times after MPI_Bcast:

\( p = 64 \)
Size of the Bcast = 32768 double's w/ initialization & rank 0 write-over, dragger test
...
Walltime = 0.0915451
Finish times for procs:
6.00815e-05 0.0904081 0.00412822 0.0944362 0.00430799 0.0946102 0.00584316
0.0960541 0.00463414 0.0949631 0.0056982 0.0959191 0.00551915
0.00527406 0.0954561 0.00545406 0.095777 0.00480604 0.0950632 0.00551915
0.0957201 0.00566316 0.095879 0.00629711 0.0966311 0.00560021
0.0956781 0.00542617 0.0955961 0.00595522 0.09542307
0.0956781 0.00542617 0.0955961 0.00635505 0.0951781 0.00595522 0.0964012
0.00750399 0.0990372 0.00810409 0.0968921 0.00785398 0.097043 0.00730801
0.0969591 0.00474215 0.096585 0.00666118 0.0993781 0.00606108 0.0970311 0.0065701
0.098341 0.00663805 0.0877762 0.00610709 0.096271 0.00559521 0.096411 0.00628114
0.0964601 0.00640011 0.0978112 0.00648022 0.0977392

The times after NBC_Ibcast indicate proper non-blocking behavior: only the time reported by processor with rank 1 is delayed. The times after NBC_Wait and the times after MPI_Bcast clearly show fan-out (binary tree) behaviors: every other processor is delayed by the unavailability of processor 1. These results are expected, but such behavior may be too fragile if we want to maximize asynchronuous execution with non-blocking collective communications as one single slow processor will delay half of the nodes. Given the nature of the problem, adaptive/fault-tolerent non-blocking broadcast may be desired.

NBC_Iallreduce:

This is the non-blocking version of MPI_Allreduce. Surprisingly, its performance when called with blocking (NBC_Iallreduce directly followed by NBC_Wait) is significantly better than MPI_Allreduce, at least until \( p=128 \):
The latencies of MPI calls are the averages of three calls. The latencies of NBC calls were tested under the same conditions, but the data clearly showed there were additional overheads for the first NBC_Iallreduce calls of the program. Therefore, the NBC latencies were the averages of the second and third calls, and the differences between the latency of the first calls and the average of the second and third ones were plotted as NBC initialization costs. The trends seem to indicate that the latency of the MPI_Allreduce scales $\sim \log(p)$ with a big constant, while that of NBC_Iallreduce scales $\sim p$ with a small constant (in fact, the trendlines were plotted accordingly). Even if that’s the case, however, the NBC version will probably still be faster than the MPI version before $p$ is on the order of tens of thousands.

That said, the implementation of NBC_Iallreduce doesn’t fulfill its original purpose: that is, overlapping the communication latency with computation. Flop’s placed between NBC_Iallreduce and NBC_Wait are still unacceptably slow, just like the case with NBC_Ibcast. Flop’s placed after NBC_Wait did get executed with normal speed, however, and the overall walltimes are better than the MPI counterparts:
The sequence of barriers, MPI_Wtime, and the operation (MPI_Allreduce or NBC_Iallreduce) are the same as before. For the NBC_Test data points, each floating-point operation is followed by NBC_Test (without NBC_Test the performance is slightly better, data omitted for clarity). For the NBC_Wait data points, all the Flop’s are placed after NBC_Wait. For the MPI data points all the Flop’s are placed after MPI_Allreduce.

In short, NBC_Wait properly frees the CPU from the collective operation, but not NBC_Test. In theory, the best way to use NBC_Iallreduce seems to be squeezing out as much overlap as possible while calling NBC_Wait right after the collective operation is done:

```c
NBC_Handle req;
NBC_Iallreduce(MPI_IN_PLACE, arr, size, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD, &req);

int l;
for(l=0; l<factor; ++l)
{
    sum += arr[0];
    if(NBC_Test(&req))
        break;
}

int round = l+1;
```
NBC_Wait(&req);
for(; l<factor; ++l) {
    sum += arr[0];
}

Where factor is the number of flop’s required for each processor. In actual tests, however, the round by which NBC_Iallreduce is finished is always reported to be 1, by all 64 processors. There are two possibilities at this point:

1. Flop’s overlapped with NBC_Iallreduce are so slow such that Allreduce is completed in one round
2. There is a bug in NBC_Test itself.

The first possibility follows seductively easily from above, but it turns out that even the drastically slowed Flop’s are orders of magnitude faster than Allreduce. The second possibility is confirmed by the next collective communication investigated.

NBC_Igather:

Although NBC_Test still fails to function properly for this operation (it always returns true, even though subsequent measurements of the wait time of NBC_Wait clearly show that the operation is not finished), NBC_Igather is functional for its original intention: CPUs can carry out Flop’s while waiting for the collective communications to finish, without reduction in performance. However, it is not trivial to apply. As clarified in the developers’ publications, these non-blocking collective communication operations require the user’s program to call NBC_Test in order to progress the communication. If NBC_Igather is called without subsequent NBC_Test, it’s equivalent to calling it right before NBC_Wait, as confirmed in the tests. Further tests showed that 2 calls of NBC_Test are sufficient for p=64, if they are spread out in time. On the other hand, NBC_Test calls are also sufficiently cheap (it is faster than flop) such that any reasonable number of calls won’t have any detectable effect on the performance. Take simplicity and tests with smaller number of flop’s into consideration, the performance tests simply made 10 NBC_Test calls, evenly spread out in time:
int waits = 10;
int segment = factor/(waits+1);

NBC_Handle req;

MPI_Barrier(MPI_COMM_WORLD);

if(rank == 0)
    start_time = MPI_Wtime();

NBC_Igather(arr, size, MPI_DOUBLE, rbuf, size, MPI_DOUBLE, 0, MPI_COMM_WORLD, &req);

int l=0;
int goal=0;

for(int k=0; k<waits; ++k)
{
    goal += segment;
    for(; l<goal; ++l)
        sum += arr[0];
    NBC_Test(&req);
}

for(; l<factor; ++l)
{
    sum += arr[0];
}

double before_wait = MPI_Wtime();

NBC_Wait(&req);
MPI_Barrier(MPI_COMM_WORLD);

if(rank == 0)
{
    finish_time = MPI_Wtime();
    wait_time = finish_time - before_wait;
    cout<<factor<<'\t'<<finish_time - start_time<<'\t'<<wait_time<<endl;
}

As before, factor is the number of flop’s required. The results vs. the MPI equivalent are plotted below.
As before, the results plotted above are the averages of three runs. The wait times in the NBC plot are the time by which the processor (specifically the root) has to stall for the communication after it finishes its flop’s. For unknown reasons, the latency of NBC_Igather is very erratic compared to its MPI counterpart, which is the likely cause of the anomaly of blocking call (zero flop) walltime. Aside from that, these results are expected. As more and more flop’s are overlapped with the communication latency, the wait time decreases linearly while the walltime stays virtually constant, so the walltime starts to increase only after the wait time drops.
to the fixed costs of NBC_Wait and MPI_Barrier. For the blocking counterpart of MPI, not such overlap is allowed, so the walltime simply grows linearly with the number of flop’s. For sufficiently large number of flop’s, the NBC_Igather consistently outperforms MPI_Gather. That said, the blocking performance of NBC_Igather does not match that of MPI_Gather:

![Blocking Gather Latency (s) vs. Number of Proc.](image)

In circumstances where large number of calculations can be overlapped with communication latency, though, such shortcoming is irrelevant. NBC_Igather can be used to improve the application’s efficiency in such situation, especially when MPI_Gather becomes the bottleneck.

NBC_Iallgather:

Not unexpected, the situation with NBC_Iallgather is similar to that of NBC_Igather, except that even its blocking calls outperform its MPI counterpart:
Walltime (s) vs. # of Flop’s following NBC_Iallgather Operation

Walltime (s) vs. # of Flop’s following MPI_Allgather Operation

(Notice the difference in the units of the axes)
Just like the case of Allreduce, additional overheads were involved for the first NBC_Iallgather calls of the program, so the NBC latencies were the averages of the second and third calls, and NBC initialization costs were plotted. Although the initialization costs are much higher here, real-world applications most likely involve at least hundreds of iterations, so the amortized costs are most likely negligible.

What’s most surprising here is the scaling of MPI_Allgather latency: it seems to be linear for small \( p \) and quadratic for large \( p \). These results suggest that naïve Alltoall-like communication patterns are used as the underlying mechanism: the latency scales linearly for small \( p \) because of the number of messages each processor sends and receives, but for large \( p \) the network is flooded by the number of messages such that only a fixed number of them can pass at the same time, so the scaling becomes quadratic. The NBC counterpart seems to scale logarithmically, indicating fan-in and fan-out communication pattern (binary tree gather followed by binary tree bcast).

NBC_Ialltoall:

This operation is also properly implemented in libNBC 1.0. As before, 10 NBC_Test calls spread out in time were used to progress the operation.
Interestingly, the wait time doesn’t drop to the baseline cost here. Further tests showed that it doesn’t happen until even larger number of NBC_Test calls are spread out in the process (somewhere between 1000~10000), and the actual performance (walltime) worsens. Most likely, substantial amount of CPU time is required on the receiving end for complete exchange regardless of latency, so trying to overlap these operations with flop’s will only add overhead like context
switching and reduce the performance. Nevertheless, overlapping allows NBC_alltoall to outperform MPI_Alltoall for large number of flop’s as expected.

The scaling of latency with respect to the number of processors is as follows. Since both the NBC version and MPI version involve initialization cost in the first call of the program for large p, the latencies are the averages of two runs and initialization costs were also plotted.
The NBC version seems to be slower by a modest factor, and both Openmpi 1.0.2 and libNBC 1.0 seem to switch the underlying algorithm from $p=16$ to $p=32$. Interestingly, both got the cutoff wrong: the algorithm for larger $p$ will probably run faster for $p=16$ and maybe even $p=8$. Openmpi 1.0.2 didn’t fail to surprise again by turning in performance better than that of Allgather – I won’t attempt to comment any further.

**NBC_Ireduce:**

This operation is also properly implemented. Since reduction is a relatively fast operation, however, the advantage of the non-blocking version is marginal even for large number of flop’s:

![Walltime (s) vs. # of Flop’s following Reduce Operation](chart.png)

As before, 10 NBC_Test calls spread out in time were used. The scaling of latency with respect to the number of processors is plotted below. In this case, both the NBC version and MPI version seem to involve initialization cost in the first call of the program for all $p$. 
The NBC version seems to be slower by a small, constant factor.

In summary,

Bad overlap: Unusable: NBC_Ibcast  
            Shorter Latency: NBC_Iallreduce

Good overlap: longer latency: NBC_Ireduce, NBC_Ialltoall, NBC_Igather  
              Shorter Latency: NBC_Iallgather
Application

Given that Homework 4 is the original motivation of the final project, it is very frustrating to find that LibNBC 1.0 is of limited applicability to my original MPI code. For the three major collective operations used – MPI_Bcast (broadcasting the locations of the vertices of the $\sqrt{N}$ hub region to all the non-root processors), MPI_Gatherv (collecting the locations of the vertices connected to the ones handled by the root), and MPI_Allreduce (Finding the longest edge), only the counterpart of MPI_Gatherv can be used. NBC_IBcast doesn’t properly release CPU time as mentioned before, and NBC_Iallreduce does not support custom MPI_Op yet. Since MPI_Allreduce is used in combination with custom type to provide the identity of the longest edge to all the processors, it can’t be converted to the NBC call. The performance of the code using NBC_Igatherv instead of MPI_Gatherv is plotted below.

![Graph showing performance comparison]

Obviously, NBC_Igatherv doesn’t improve the performance. Moreover, the plateauing behavior of the program with $p>32$ remains identical, indicating that the performance bottleneck exists somewhere else.
Conclusion

Given the presence of MPI_Isend/Irecv and the whole range of collective operations, the absence of the non-blocking collective operations is a peculiar omission. LibNBC seems to be a promising remedy, but caution must be exercised for the current iteration: flaws render certain operations unusable, and most of them are slower in the blocking setting. In cases where the original blocking operations are not the bottleneck, non-blocking version won’t improve the performance.

Personally, I would like to see non-blocking collective operations included in the next version of the MPI standard. This will not only provide vastly more developer-time to improve the implementation, but also make it easier to optimize using lower-level resources. In particular, the requirement of calling NBC_Test to progress the communication makes the non-blocking operations harder to use and generates additional overheads like context switch, while offloading the task to network controller or even dedicated core should eliminate the problem. Fortunately given the information I found online, non-blocking collective operations have a good chance of making it into MPI-3 standard.

The website of libNBC:

http://www.unixer.de/research/nbcoll/libnbc/

(Friendly reminder: I am looking forward to doing authentic work on the scaling issues of parallel scientific programming for my thesis research.)
Appendix: NBCBench 1.0 Results on Nyx

These are the benchmark results generated by NBCBench 1.0. “Microbenchmarking” was used for these results: that is, it measures the individual CPU time used in these operations instead of the collective walltime. As multiple tests were performed, the median of the maximal CPU time used among all the processors is plotted. The rationale is that when sufficient amount of computation can be overlapped with the communication, such measure reflects the performance gain without complications. As before, the “data size” referred in the x-axis means the size of the individual messages involved, and the rightmost data points correspond to the walltime benchmarks above.

We can see that the flaws that consume the CPU time well after the operation are not captured in the Bcast and Allreduce microbenchmarks. NBC performs better in terms of walltime in complete exchange compared to MPI, but was reported to be about the same by NBCBench 1.0. NBC_Ireduce was reported to be slightly worse by NBCBench 1.0, but the walltime is the other way around. The walltime and microbenchmarks only agree on the performance of NBC_Iallgather(v).
MPI_Reduce Time (p=64)

The graph shows the relationship between the data size (in bytes) and the time (in t) taken to perform the MPI Reduce operation for different processors (p=64). The graph compares the performance of MPI and NBC methods, with MPI generally showing a lower time for the same data size compared to NBC.