Design, Implementation, and Usage of LibNBC

Torsten Hoefler[1,2] and Andrew Lumsdaine[1]

[1]Open Systems Laboratory
Indiana University
501 N. Morton Street
Bloomington, IN 47404 USA
{htor,lums}@cs.indiana.edu

[2]Dept. of Computer Science
Technical University of Chemnitz
Strasse der Nationen 62
Chemnitz, 09107 GERMANY
htor@cs.tu-chemnitz.de

August 17, 2006

LibNBC Version 0.8.1

Abstract

We describe the design and implementation of LibNBC a library that implements non-blocking collective operations. Its main goals are high portability and high performance. The library is written in ANSI C on top of MPI-1. This document describes the internal design, implementation and various internal and external programming interfaces of LibNBC.

1 Introduction

The NBC (Non-Blocking Collective) library (short: LibNBC) is a portable implementation of the non-blocking collective operations proposed as an addition to MPI-2 [7]. The main advantage non-blocking collective operations is that they offer a high flexibility to enable the programmer to use non-blocking semantics to overlap computation and communication, and to avoid pseudo-synchronization of the user application. Non-blocking collective operations combine the advantages of non-blocking point-to-point communication [1] and collective communication [3].

LibNBC is based on MPI-1 and written in ANSI C to enable high portability to many different parallel systems. Some MPI implementations offer the possibility of overlapping communication and computation with non-blocking point-to-point communication (e.g. Open MPI [2]). Using LibNBC with such a library enables overlap for collective communications. However, most MPI implementations can be used to avoid pseudo-synchronization [8].

The next section defines the main goals of LibNBC. Section 2 provides details about the current implementation, followed by a description of the internal and external APIs in Section 3. The last section provides some examples of the internal implementation of collective algorithms, which indicate the extensibility of the library, and some examples for the usage of non-blocking collective operations.

1.1 Goals

The main goal of LibNBC is to provide a portable and stable implementation of non-blocking collective operations on top of MPI-1. LibNBC should support all collective operations defined in MPI-1 and should be easily extensible with new operations. The overhead added by the library should be minimal so that a blocking execution (e.g. MPI_IBCAST immediately followed by a MPI_WAIT) should have the same performance as the blocking collective operation (MPI_BCAST). The potential to overlap communication and computation should be as high as the lower communication layers support and support for special hardware operations should be easily addable.

2 Implementation

LibNBC is written in ANSI C (C90) to compile with `gcc -W -Wall -ansi -pedantic` without warnings or errors. The library uses so called collective schedules to save the necessary operations to complete a MPI collective communication. These schedules are built with helper functions and executed by the scheduler to perform the operation.

2.1 The Collective Schedule

A collective schedule is a process specific "execution plan" for a collective operation. It consists of all necessary information to perform the operation. Collective schedules are designed to support any collective communication scheme with arbitrary data dependencies. A schedule can consist of multiple rounds to model the data dependencies. Operations in round r may depend on operations in rounds $i \leq r$ and are not executed by the scheduler before all operations in rounds $j < r$ have been finished. Each collective operation can be represented as a series of point-to-point operations with one operation per round. Some operations are independent of each other (e.g. the data sent in a simple linear MPI_BCAST), and some depend on previous ones (e.g. a non-root, and non-leaf process in a tree implementation of MPI_BCAST has to wait for the data before it can send to its children). Independent operations can be in the same round and dependent operations have to be in the right order in different rounds.

Definition 2.1: *A collective schedule is a process specific plan to execute a collective operation. It consists of $r \geq 1$ rounds and $o \geq 0$ operations.*

Definition 2.2: *A round is a building block of a collective schedule which may consist of $o \geq 0$ operations. Rounds are interdependent, round r will only be started if all operations in round $r-1$ are finished. Round 0 can be started immediately.*

Definition 2.3: *An operation is the basic building block of a collective schedule. It is used to progress collective operations. Operations are grouped in rounds and executed by the scheduler. Possible data dependencies between operations are expressed in their grouping into rounds.*

The schedule design enables coarse-grained dependencies which allow some degree of parallelism and an optimized implementation of the schedule. It is possible to implement automatic and transparent segmentation and pipelining.

```
schedule = size, {round, delimiter}, round, end | size, end;
round = num, {type, type-args};

size = 'size of the schedule (int)';
num = 'number of operations in round (int)';
type = 'operation type (enum NBC_Fn_type)';
type-args = 'operation specific arguments (struct NBC_Args_<type>)';
delimiter = '1';  (char), indicates next round;
end = '0';  (char), indicates next round;
```

Listing 1: EBNF of the Schedule Array

2.1.1 Memory Layout of a Schedule

The easy round-based design of a schedule allows the schedule to be stored contiguously in memory. This design is very cache-friendly; the fetch of the first operation of the round will most likely also load the following operations. This special design has been used previously to optimize MPI_BARRIER synchronization for InfiniBand™ where the calculation of the communication peers during the execution was too costly and had been done in advance [6].

The first element of each schedule is the size in bytes, stored as an int. At least one round follows the size element. A round consists of a number of operations, o, stored as int, and o operation argument structures. The operation argument structures consist of all operation-specific arguments and vary in size. A delimiter, stored as char, follows the last operation of each round. A delimiter may be followed by another round or indicates the end of the schedule. An EBNF of the schedule array is given in Listing 1.

2.2 Identifying a Running Collective Operation

A handle (NBC_HANDLE) is used to identify running collective operations, which are called instances in the following.

Definition 2.4: *An instance of a collective operation is a currently running operation which has not been completed by Test or Wait.*

The handle identifies the current state, the schedule and all necessary information to progress the collective operation. Each handle is linked to a user communicator. Each user communicator is duplicated at the first use into a so called shadow communicator. All communication done by LibNBC is performed on the shadow communicator to prevent tag collisions on the user comm. A tag, specific to each communicator-instance pair, is also saved at the handle to distinguish between different ongoing collective operations on the shadow communicator. The handle holds all information related to outstanding requests (in the current implementation MPI_REQUESTS, but other (e.g., hardware specific) request types are also possible) and a void pointer to store arbitrary information needed by the collective routines (e.g., temporary buffers for reductions).

The current state of an instance is determined by the pending requests and the current round in the schedule. The open requests are attached to the handle and the current round is saved as an integer offset (bytes) in the schedule array. This means that a single schedule could be used by many instances at the same time (i.e., can be attached to multiple handles) and may remain cached at the communicator for future usage (this is not implemented yet).

2.3 The Non-Blocking Schedule Execution

The execution of a schedule is implemented in the internal function NBC_START. It checks if a shadow communicator exists for the passed user communicator and creates one if necessary. MPI_ATTRIBUTES attached to the user communicator are used to store communicator specific data (shadow comm and tag). Tags are reset to "1" if they reach "32767" or if a new shadow communicator is constructed. The NBC_START function sets the schedule offset at the handle to the first round and starts the round with a call to NBC_START_ROUND.

2.3.1 Starting the Execution of a Round

The function NBC_START_ROUND issues all operations for the next round (indicated by the handles offset). All requests returned by non-blocking operations (like Isend, Irecv) are attached to the handle and all blocking (local) operations (like Copy, Operation) are executed at this point. The progress function is called to check if the round can already be finished (e.g., if it only had local operations).

2.4 Progressing a Non-Blocking Instance

We differentiate two kinds of progress. The first kind of progress is done inside the library (e.g., to send the next fragments). The MPI progress may be asynchronous (e.g., separate thread), or synchronous (user has to call MPI_TEST to achieve progress) depending on the MPI implementation. The second progress is the transition from one round to another. This is currently only implemented in a synchronous fashion, which means that users should call NBC_TEST if they want the operation to progress in the background.

However, the current version of LibNBC implements the synchronous fashion and performs the progress in NBC_PROGRESS or NBC_PROGRESS_BLOCK, which call MPI progress functions. These functions test all outstanding requests for completion. A successful completion of all requests means that the active round is finished and the instance can be moved to the next round. The whole operation is finished if the current round is the last round and NBC_OK is returned. If a next round exists, NBC_START_ROUND is called after adjusting the handles offset to the new round. The function returns NBC_CONTINUE if there are still outstanding requests in this round. NBC_PROGRESS_BLOCK behaves identically but blocks (uses MPI_WAITALL instead of MPI_TESTALL) until the instance is finished.

3 API Definition

This section defines the internal API to add new non-blocking collective algorithms and new collective functionality (e.g., new operations) to LibNBC and the external API for end-users to use the non-blocking operations. All functions return NBC error codes, defined in `nbc.h`.

3.1 Internal API - Building a Schedule

This section describes all the functions that can be used to build a schedule for a collective operation.

`NBC_Sched_create(NBC_Schedule* schedule)` allocates a new empty schedule array for later usage.

`NBC_Free(NBC_Handle *handle)` deallocates a schedule and all related allocated memory.

`NBC_Sched_send(void* buf, int count, MPI_Datatype datatype, int dest, NBC_Schedule *schedule)` adds a non-blocking send operation to the schedule

`NBC_Sched_recv(void* buf, int count, MPI_Datatype datatype, int source, NBC_Schedule *schedule)` adds a non-blocking receive operation to the schedule

`NBC_Sched_op(void* tgt, void* src1, void* src2, int count, MPI_Datatype datatype, MPI_Op op, NBC_Schedule *schedule)` adds a blocking local reduction operation of the buffers `*src1` and `*src2` into `*tgt` to the schedule.

`NBC_Sched_copy(void *src, int srccount, MPI_Datatype srctype, void *tgt, int tgtcount, MPI_Datatype tgttype, NBC_Schedule *schedule)` adds a blocking local copy operation from `*src` to `*tgt` to the schedule.

`NBC_Sched_barrier(NBC_Schedule *schedule)` ends the current round in the schedule and adds a new round.

`NBC_Sched_commit(NBC_Schedule *schedule)` ends the schedule. The commit function could be used to apply further optimization to the schedule, e.g., to add automatic segmentation and pipelining (currently not done).

3.2 Internal API - Executing a Schedule

This section describes all internal functions that are used to execute a schedule.

`NBC_Start(NBC_Handle *handle, MPI_Comm comm, NBC_Schedule *schedule)` starts the execution of a schedule, see Section 2.3. NOTE: NBC_START_ROUND should never be called directly, it is only called by NBC_START and NBC_PROGRESS!

`NBC_Progress(NBC_Handle *handle)` progresses the instance identified by the handle. Returns NBC_OK if the instance (collective operation) is finished or NBC_CONTINUE if there are still open requests.

3.3 Internal API - Performing Local Operations

This section describes all internal functions that are used to perform local (blocking) operations.

`NBC_Copy(void *src, int srccount, MPI_Datatype srctype, void *tgt, int tgtcount, MPI_Datatype tgttype, MPI_Comm comm)` copies a message from a source to a destination buffer (from `*src` to `*tgt`).

`NBC_Operation(void *tgt, void *src1, void *src2, MPI_Op op, MPI_Datatype type, int count)` performs a reduction operation from `*src1` and `*src2` into `*tgt`.

3.4 External API

The external API is similar to the standard proposal in [7]. Differences are:

1. Names are prefixed with `NBC_` instead of `MPI_` to avoid confusion with standardized MPI calls.
2. Environmental attributes are not defined, the maximum number of running instances is 32767.
3. Testall, Waitall, Testsome, Waitsome, Testany, and Waitany are not implemented yet.
4. NBC_TEST and NBC_WAIT take only a single argument (NBC_HANDLE - no status or flag is returned, the return-value indicates completion)
5. Fortran bindings are not implemented yet.
6. User defined operations are not supported yet.
7. NBC collective operations cannot be mixed with MPI collective operations

4 Examples

This section presents two examples. Section 4.1 is for implementers who want to implement new non-blocking collective algorithms using the NBC scheduler. Section 4.2 is for MPI programmers which want to use the non-blocking collective operations to optimize their program.

4.1 Implementing Collective Algorithms in LibNBC

New collective algorithms can easily be added to LibNBC. We present an example of the implementation of the dissemination barrier, that has been proposed in [4] and a pseudo-code is given in [5]. The implementation with the NBC scheduler can be found in `nbc_ibarrier.c` in the LibNBC sources. An excerpt of the code is show in Listing 2

Line 13 creates a new schedule. The algorithm consists of $log_2 P$ rounds and processes synchronize pairwise in each round. This means that each process sends and receives to/from other processes each round and it has to wait until the messages have been received. The rank-specific schedule

```
1  int NBC_Ibarrier(MPI_Comm comm, NBC_Handle* handle) {
     int round, rank, p, maxround, res, recvpeer, sendpeer;
     NBC_Schedule *schedule;

     res = MPI_Comm_rank(comm, &rank);
6    res = MPI_Comm_size(comm, &p);

     schedule = malloc(sizeof(NBC_Schedule));

     round = -1;
11   handle->tmpbuf=NULL;

     res = NBC_Sched_create(schedule);

     maxround = (int)ceil((log(p)/LOG2)-1);
16
     do {
       round++;
       sendpeer = (rank + (1<<round)) % p;
       /* add p because modulo does not work with negative values */
21     recvpeer = ((rank - (1<<round))+p) % p;

       /* send msg to sendpeer */
       res = NBC_Sched_send(NULL, 0, MPI_BYTE, sendpeer, schedule);

26     /* recv msg from recvpeer */
       res = NBC_Sched_recv(NULL, 0, MPI_BYTE, recvpeer, schedule);

       /* end communication round */
       if(round < maxround){
31       res = NBC_Sched_barrier(schedule);
       }
     } while (round < maxround);

     res = NBC_Sched_commit(schedule);
36
     res = NBC_Start(handle, comm, schedule);

     return NBC_OK;
   }
```

Listing 2: Dissemination Barrier in LibNBC (error checks removed)

```
int function() {
  buffer comm, comp;

  do {
    /* do some computation on comm buffer */

    MPI_Bcast(comm, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* do some computation on comp buffer */

  while (problem_not_solved);
}
```

Listing 3: Code example with blocking MPI_BCAST

is built in the do-loop (Line 17-32). The schedule is committed in Line 36 and ready for use afterwards. Line 27 starts the scheduler to execute the schedule (non-blocking).

A schedule of rank 0 in a 4-process communicator would consist of two rounds. Each round consists of a send and a receive operation, and its representation in memory is shown in Figure 1.

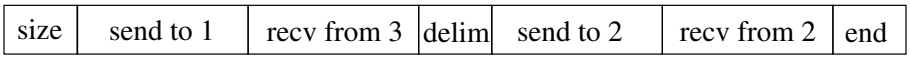

Figure 1: Schedule for Rank 0 of 4 for a Dissemination Barrier

4.2 Using Non-Blocking Collective Operations from a MPI Program

The transition from blocking MPI collective operations to non-blocking NBC collective operations is simple. The NBC interface is similar to the blocking MPI collective operations and adds only a NBC_HANDLE as last parameter. The transition for a code using MPI_BCAST to a code using NBC_IBCAST is shown in Listing 3 and Listing 4.

Listing 3 uses a blocking MPI_BCAST, but the computation in the comp buffer is independent of the communication of the comm buffer. This enables the user to use overlap techniques with non-blocking collective operations. Listing 4 shows a possible way to leverage overlapping.

To ensure progress in the background, the user should call NBC_TEST on active handles as shown in Listing 5.

Acknowledgements

The authors want to thank Laura Hopkins from Indiana University for editorial comments and writing support.

```
int function() {
  buffer comm, comp;
  NBC_Handle handle;

  do {
    /* do some computation on comm buffer */

    NBC_Ibcast(comm, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD, &handle);

    /* do some computation on comp buffer */

    NBC_Wait(&handle);
  while (problem_not_solved);
}
```

Listing 4: Code example with non-blocking NBC_IBCAST

```
int function() {
  buffer comm, comp;
  NBC_Handle handle;

  do {
    /* do some computation on comm buffer */

    NBC_Ibcast(comm, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD, &handle);

    /* do some computation on comp buffer */

    NBC_Test(&handle);

    /* do some computation on comp buffer */

    NBC_Wait(&handle);
  while (problem_not_solved);
}
```

Listing 5: Code example with non-blocking NBC_IBCAST with NBC_TEST

References

[1] Ron Brightwell and Keith D. Underwood. An analysis of the impact of mpi overlap and independent progress. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 298–305, New York, NY, USA, 2004. ACM Press.

[2] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[3] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[4] Debra Hengsen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[5] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte - CSR-04-03*, 2004. url: http://archiv.tu-chemnitz.de/pub/2005/0074/data/CSR-04-03.pdf.

[6] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. Fast Barrier Synchronization for InfiniBand. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06)*, April 2006.

[7] Torsten Hoefler, Jeffrey M. Squyres, George Bosilca, and Graham Fagg. Non Blocking Collective Operations for MPI-2. Technical report, Indiana University, 2006.

[8] J.B. White III and S.W. Bova. Where's the Overlap? - An Analysis of Popular MPI Implementations, 1999.