MPI-3 Coll Workgroup

Status Report to the MPI Forum

presented by: T. Hoefler

edited by: J. L. Traeff, C. Siebert and A. Lumsdaine

July 1st 2008

Menlo Park, CA

Overview of our Efforts

0) clarify threading issues

1) sparse collective operations

2) non-blocking collectives

3) persistent collectives

4) communication plans

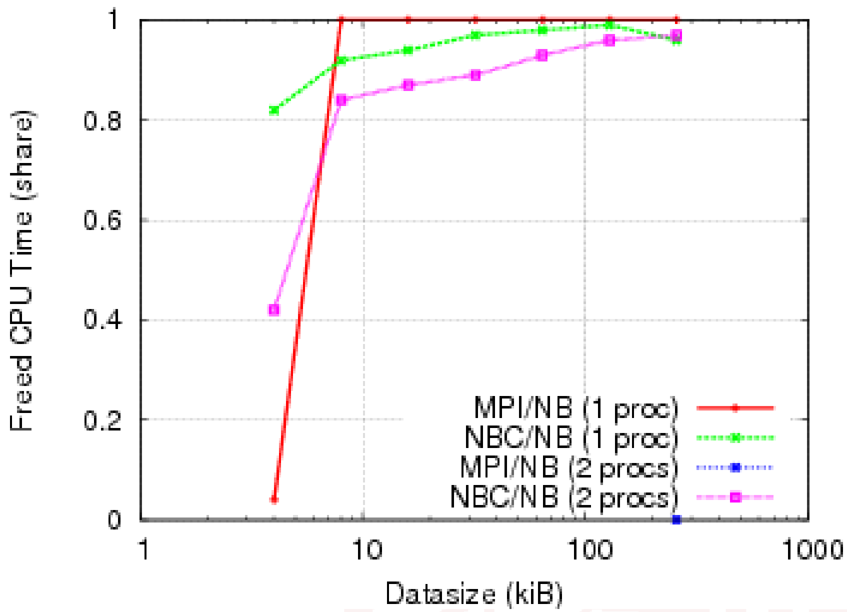
5) some smaller MPI-2.2 issues

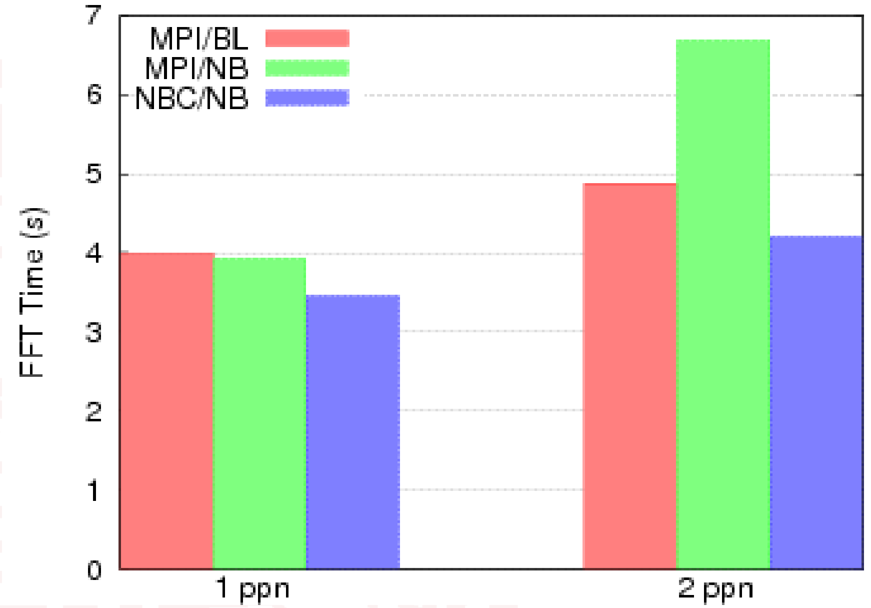
Can threads replace non-blocking colls?

"If you got plenty of threads, you don't need asynch. collectives"

- we don't talk about asynch collectives (there is not much asynchronity in MPI)
- some systems don't support threads
- do we expect the user to implement a thread pool (high effort)? Should he spawn a new thread for every collective (slow)?
- some languages don't support threads well
- polling vs. interrupts? All high-performance networks use polling today – this would hopelessly overload any system.
- is threading still an option then?

Threads vs. Colls - Experiments





used system: Coyote@LANL, Dual Socket, 1 Core

- EuroPVM'07: *"A case for standard non-blocking collective operations"*
- Cluster'08: *"Message progression in parallel computing – to thread or not to thread?"*

High-level Interface Decisions

Option 1: ”One call fits all”

- 16 additional function calls
- all information (sparse, non-blocking, persistent) encoded in parameters

Option 2: ”Calls for everything”

- 16 * 2 (non-blocking) * 2 (persistent) * 2 (sparse) = 128 additional function calls
- all information (sparse, non-blocking, persistent) encoded in symbols

Differences?

- implementation costs are similar (branches vs. calls to backend functions)
- Option 2 would enable better support for subsetting
- pro/con? – see next slides

1) One call fits all

Pro:

- less function calls to standardize
- matching is clearly defined

Con:

- users expect the similar calls to match (prevents different algorithms)
- against MPI philosophy (there are n different send calls)
- higher complexity for beginners
- many branches and parameter checks necessary

2) Calls for everything

Pro:

- easier for beginners (just ignore parts if not needed)
- enables easy definition of matching rules (e.g., none)
- less branches and parameter checks in the functions

Con:

- many (128) function calls

Example for Option 1

MPI_Bcast_init(buffer, count, datatype

root, group, info, comm, request)

New Arguments:

- group – the sparse group to broadcast to
- info – an Info object (see next slide)
- request – the request for the persistent communication

The Info Object

hints/assertions to the implementation (preliminary):

- enforce (init call is collective, enforce schedule optimization)
- nonblocking (optimize for overlap)
- blocking (collective is used in blocking mode)
- reuse (similar arguments will be reused later – cache hint)
- previous (look for similar arguments in the cache)

Examples for Option 2

- MPI_Bcast(<bcast-args>)
- MPI_Bcast_init(<bcast-args>, request)
- MPI_Nbcast(<bcast-args>, request)
- MPI_Nbcast_init(<bcast-args>, request)
- MPI_Bcast_sparse(<bcast-args>, group-or-comm)
- MPI_Nbcast_sparse(<bcast-args>, group-or-comm)
- MPI_Bcast_sparse_init(<bcast-args>, group-or-comm, request)
- MPI_Nbcast_sparse_init(<bcast-args>, group-or-comm, request)

(<bcast-args> ::= buffer, count, datatype, root, comm)

Isn't that all fun?

- obviously, this is all too much
- we need only things that are useful, why not:
- omit some combinations, e.g., Nbcast_sparse (user would *have* to use persistent to get non-blocking sparse colls)?
  (-> reduction by a constant)
- abandon a parameter completely, e.g., don't do persistent colls
  (-> reduction by a factor of two)
- abandon a parameter and replace it with a more generic technique?   (see MPI plans on next slides)
  (-> reduction by factor of two)

MPI Plans

- represent arbitrary communication schedules
- a similar technique is used in LibNBC and has been proven to work (fast and easy to use)
- MPI_Plan_{send,recv,init,reduce,serialize,free} to build process-local communication schedules
- MPI_Start() to start them (similar to persistent requests)
- -> could replace all (non-blocking) collectives, but ...

MPI Plans - Pro/Con

Pro:

- less function calls to standardize
- highest flexibility
- easy to implement

Con:

- no (easy) collective hardware optimization possible
- less knowledge/abstraction for MPI implementors
- complicated for users (need to build own algorithms)

But Plans have Potential

- could be used to implement libraries (LibNBC is the best example)
- can replace part of the collective (and reduce the implementation space), e.g.:
- sparse collectives could be expressed as plans
- persistent collectives (?)
- homework needs to be done ...

Sparse/Topological Collectives

- Option 1: use information attached to topological communicator
- MPI_Neighbor_xchg(<buffer-args>, topocomm)

- Option 2: use process groups for sparse collectives
- MPI_Bcast_sparse(<bcast-args>, group)
- MPI_Exchange(<buffer-args>, sendgroup, recvgroup)

  (each process sends to sendgroup and receives from recvgroup)

Option 1: Topological Collectives

Pro:

- works with arbitrary neighbor relations and has optimization potential (cf. ”Sparse Non-Blocking Collectives in Quantum Mechanical Calculations” to appear in EuroPVM/MPI'08)
- enables schedule optimization during comm creation
- encourages process remapping

Con:

- more complicated to use (need to create graph communicator)
- dense graphs would be not scalable (are they needed?)

Option 2: Sparse Collectives

Pro:

- simple to use
- groups can be derived from topocomms (via helper functions)

Con:

- need to create/store/evaluate groups for/in every call
- not scalable for dense (large) communications

Some MPI-2.2 Issues

1) Local reduction operations:

- MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op)
- reduces inbuf and inoutbuf locally into inoutbuf as if both buffers were contributions to MPI_Reduce() from two different processes in a communicator
- useful for library implementation (libraries can not access user-defined operations registered with MPI_Op_create())
- LibNBC needs it right now
- implementation/testing effort is low

Some MPI-2.2 Issues

2) Local progression function:

- MPI_Progress()
- gives control to the MPI library to make progress
- is commonly emulated ”dirty” with MPI_Iprobe() (e.g., in LibNBC)
- makes (pseudo) asynchronous progress possible
- implementation/testing effort is low

Some MPI-2.2 Issues

3) Request completion callback

- MPI_register_cb(req, event, fn, userdata)
- event = {START, QUERY, COMPLETE, FREE}
- used for all MPI_Requests
- easy to implement (at least in OMPI ;))
- gives more progression options to the user
- would enable efficient LibNBC progression

Some MPI-2.2 Issues

4) Partial pack/unpack:

- modify MPI_{Pack,Unpack} to allow (un)packing parts of buffers
- simplifies library implementations (e.g., LibNBC can run out of resources if large 1-element data is sent because it packs it)
- necessary to deal with very large datatypes

More Comments/Input?

Any items from the floor?

General comments to the WG?

Directional decisions?

How's the MPI-3 process? Should we go off and write formal proposals?