The Case for Collective Pattern Specification

Torsten Hoefler, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine

Advances in Message Passing, Toronto, ON, June 2010
Motivation and Main Theses

- Message Passing (MP) is a useful programming concept
  - Reasoning is simple and (often) deterministic
  - Message Passing Interface (MPI) is a proven interface definition
- MPI often cited as “assembly language of parallel computing”
  - Not quite true as MPI offers collective communication
  - But: Many relevant patterns are not covered
    - e.g., nearest neighbor halo exchange
- Bulk Synchronous Parallelism is a useful programming model for MP programs
  - Easy to reason about the state of the program
    - cf. structured programming vs. goto
Valiant’s BSP Model

- Envisioned as hardware and software model
  - SPMD program execution is split into $k$ supersteps
  - All instances are in the same superstep
    - Implies synchronization / synchronous execution
  - Messages can be sent and received during superstep $i$
    - Received messages can be accessed in superstep $i + 1$

- Our claim:
  - Many algorithm communication patterns are constant or exhibit temporal locality
    - Should be defined as such!
    - Allows various optimizations
    - Takes the MPI abstractions to a new (higher) level
We classify applications (or algorithms) into five main classes of communication patterns:

1. Compile-time static
2. Run-time static
3. Run-time flexible
4. Dynamic
5. (Massively parallel) mostly for completeness and not discussed further
Compile-time static

- Communication pattern is completely described in source code
  - Shape is independent of all input parameters
- Implementation in MPI
  - Either collectives or bunch of send/recvs
  - Proposal for “Sparse collectives” allows definition of arbitrary collectives (MPI 3?)
- Examples:
  - MIMD Lattice Computation (MILC) – 4d grid
  - Weather Research and Forecasting (WRF) – 2d grid
  - ABINIT – collectives only (Alltoall for 3d FFT)
Run-time static

- Communication pattern depends on input but is fixed during execution
  - Can be compiled once at the beginning
- Implementation in MPI
  - Use graph partitioner (ParMetis, Scotch, …)
  - Send/recv communication for halo zones
  - Will be supported by “Sparse Collectives”
- Examples:
  - TDDFT/Octopus – finite difference stencil on real domain
  - Cactus framework
  - MTL-4 (sparse matrix computations)
Run-time flexible

- Communication pattern depends on input but changes over time
  - However, there is still some locality
- Implementation in MPI
  - Graph partitioning and load balancing
  - Typically send/recv communication (often request/reply)
  - Static optimization might be of little help if pattern changes too frequently
- Examples:
  - Enzo – cosmology simulation - 3d AMR
  - Cactus framework - Berger-Oliger AMR
Dynamic

- Communication pattern only depends on input and has no locality
  - Little can be done: BSP might not be the ideal model

- Implementation in MPI:
  - Typically send/recv request/reply
    - Active message style
  - Often employ “manual” termination detection with collectives (Allreduce)
  - Not a good fit to MPI 2.2 (MPI 3?)

- Examples:
  - Parallel Boost Graph Library (PBGL) – implements various graph algorithms on distributed memory
Our Proposal

- Specify collective operations explicitly
  - MPI has collectives
    - … but they are inadequate
  - Want to express sparse collectives easily

- A *declarative* approach to specifying communication patterns
- Describe the *what*, not the *how*, of communications
- An abstract specification that is implemented efficiently
  - Don’t talk about individual messages
Benefits

- Abstract specification
  - Easier for programmers to understand
- Easier for compilers to optimize
  - Overlap communication and computation
  - Message coalescing, pipelining, etc.
  - Does not need to be implemented as BSP (weak sync.)
- An efficient runtime
  - That can choose an implementation approach based on memory/network tradeoffs
  - Use one-sided or two-sided based on hardware
Compile-time static

- Communication patterns expressed as a set of individual communication operations
- Built by quantifying over processors, array rows, etc.
- Dense and sparse collectives are supported directly
- Compiler optimizations apply readily

For all nodes p in grid:
  send A[0] on p to B[n] on up(p)
  and A[n] on p to B[0] on down(p)
Run-time static and flexible

- Collective communication pattern can be generated at run-time, and regenerated as necessary
  - Communication operations can use array references, etc.
- Compiler analyses are more difficult in these cases
  - Run-time optimization must sometimes be used
- Communication patterns may not be known globally
  - Not scalable for large systems
  - Conversion to multicast/… trees may be impossible

For all nodes p in grid:

send A[0] on p to B[n] on next[p]
Summary

- Communications in BSP-style programs should be expressed as *collective operations*
- We suggest using a declarative specification of the communication operations
  - Better ease of development
  - Enables compiler optimizations (e.g., removing strict synchronization)
- Our approach can be embedded into an existing programming language as a library
  - Can be added incrementally to existing applications
Thank you for your attention!

Discussion

Algorithm 2: NBX—Nonblocking Consensus.

Input: List \( l \) of destinations and data
Output: List \( O \) of received data and sources

1. \( \text{done}=\text{false}; \)
2. \( \text{barr}_{\text{act}}=\text{false}; \)
3. foreach \( i \in l \) do
   4. start nonblocking synchronous send to process \( \text{dest}(i) \);
5. while \( \text{not done} \) do
   6. msg = nonblocking probe for incoming message;
   7. if \( \text{msg found} \) then
      8. allocate buffer, receive message, add buffer to \( O \);
   9. if \( \text{barr}_{\text{act}} \) then
      10. \( \text{comp} = \text{test barrier for completion}; \)
      11. if \( \text{comp} \) then \( \text{done}=\text{true}; \)
   12. else if all sends are finished then
      13. start nonblocking barrier;
      14. \( \text{barr}_{\text{act}}=\text{true}; \)