New and Improved MPI Abstractions - Performance, Portability, Planning, Productivity

Anthony Skjellum

Professor

SimCenter: Center of Excellence for Applied Computational Science and Engineering

The University of Tennessee at Chattanooga

MPI Abstractions

MPI is nearly 30 years old:

1. Cost of performance-portability has gone up
   1. Architectures are more complex and heterogeneous
   2. Scales are much larger
2. We need to cope with i) GPUs, ii) manycore, iii) concurrency inside MPI; iv) multi-model apps
3. We need to overlap communications, communications, computations, I/O
4. Move CPU code out of critical path
5. Higher level abstractions would enable multiple strategies inside MPI implementations – vs. one done by the application itself

MPI Abstractions - Extrapolations

MPI applications and frameworks benefit immediately from

1. New abstractions – are powerful for performance
   1. Extrapolations (e.g., persistent collective communication)
   2. Partitioned point-to-point
2. More new abstractions
   1. Partitioned/persistent, neighborhood collectives
   2. Triggered operations (new, scalable inter-model triggering and completion)
3. Using temporal locality allows planning and reduced cost of portability
4. Not all operations are 100% persistent, or 100% variable… we have to explore “slow changing” patterns too—open topic

MPI Abstractions - Extrapolations

MPI is applications benefit immediately from

1. Both aggregating and disaggregating messages can help with “halo codes”
2. SOME MOTIVATING IDEAS:

   $N_{1/2}$ ~ 2,500-25,000 range for modern networks… don’ need megabyte messages for 90% efficiency … plus concurrent injection possible with modern NICs

   Modern networks are/will be adaptive, out-of-order, offer offload ops, primitive QoS… MPI forces ordering semantics, has no differentiated service, no predictability or admission controls

MPI Abstractions – Raise abstraction level – Example data reorganization

Provide APIs, that are MPI supersets, that reflect the goals of the application

1. Data transpose
2. Data reorganization

3. MPI_Alltoall* is not the best level of abstraction
   -> What vs. how issue ; bundles of send/receives too low-level
4. Describing these operations again and enabling these abstractions supports i) efficient implementations, ii) potential for network offload, iii) profile-guided optimization,…
5. Tie exploration to multi-physics applications' needs for sharing data – not just 3D FFT

MPI Abstractions – Memory Kinds/Layouts

Modern CPU/GPU/NIC architectures have memory kinds

1. Just like there is caching in the CPU, there are memory kinds and MPI is oblivious to this (NIC memory, HBM banks, nonvolatile,….)
2. Opportunities
    1. Manage (or help manage) special, limited memory
    2. Potential to support constructive semantics and reduce copies
3. Moving more memory management associated with transfers to MPI (and other models) can produce serious headroom
4. Space-filling orderings also deserve special support/attention
5. Dual concept of offload

MPI Abstractions – Sessions/Topo Ops

MPI-4 has sessions; MPI-Next "can" have direct topological collectives/comms

1. We can refactor applications to scale better than requiring MPI_COMM_WORLD
2. Most apps don't need all-to-all virtual topology of the default communicator
3. Groups -> Topological and Neighbor Communicators and Collectives may be an important source of better scalability and performance
4. Better to build up, rather than build-down, legacy from MPI-1
5. Aspects of QoS and planning will couple with this approach

MPI Abstractions – Language Interfaces

Many C++ and Fortran apps can benefit from new, modern interfaces

1. Starting with better C++ interfaces to MPI can
    1. Reduce middle level code
    2. Replace derived datatypes – more natural gathers and scatters possible
    3. Support better hardware offload
    4. Drive MPI implementations to be refactored to inline and optimize
2. Lots of the new NNSA/DOE applications are C++, so a good choice
3. Modern Fortran would follow; we need to figure out what's possible just with the language
4. Source-to-source translation will also be considered in reducing overheads, if appropriate (e.g., Rose, LLVM)

Some sources of Headroom

New abstractions can really match to underlying hardware with small changes to legacy DOE/NNSA MPI apps

1. Persistent collectives can be implemented without big receive queues (e.g., RDMA inside)
2. Partitioned operations
   1. Thread interface to MPI – but offloadable to GPUs/FPGAs
   2. Isolates parts of MPI that need to be highly concurrent
   3. Reflects the need to have parallel buffer production and consumption
3. Triggers, schedules and completions - planning
   1. Enables multi-model integration without forcing MPI or 3rd party libraries to co-standardize or co-implement their products. DAGs in CUDA, DAGs needed in MPI, etc.
   2. Reflects the need to support overlap of communication and communication

ExaMPI Practical, Modern C++ MPI

Experiment, Demonstrate, Drive Production MPIs…

1. Fully progressive MPI with modularity
2. Supports quick prototyping of new ideas
3. Modern C++ source base allows tractable experimentation
4. Use as vehicle to demonstrate new performance, new abstractions, and, later
    1. Propose as MPI-5+ additions
    2. Show path for production MPI implementations

System and User-level Schedules, I

Unique extension API of ExaMPI

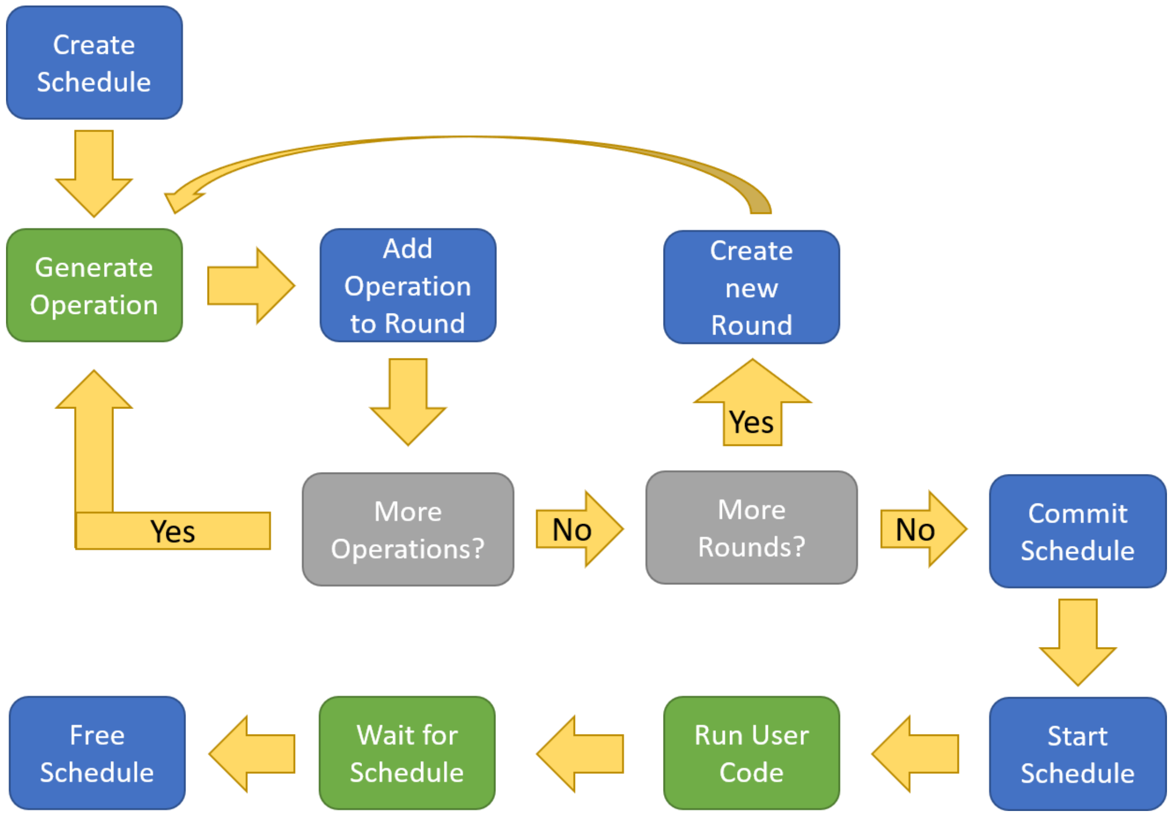
1. Builds on best-practices internal to OpenMPI
2. Supports Turing-complete compositions of schedules
    1. New collectives
    2. Full persistence
    3. Integration with triggered operations
        1. Kernel kicks messages off (send) or whole schedules
        2. Messages arriving kick kernels off
        3. Kernels can be i) CPU, ii) GPU
    4. Progress engine(s) advance the schedules (strong progress)

System and User-level Schedules, II

Unique extension API of ExaMPI

1. We get MPI+X integration with performance
2. We get user extensibility without recompiling MPI
3. We avoid having GPU-specific extensions for heterogeneous programming in MPI and vice versa
4. System-level schedules backport User-level -> May allow “unsafer” and “lower-level” operations (2-level security model)
5. Could integrate with Task runtimes and unified resource backend managers

Concept of User-level schedule



Portable Implementation of Partitioned Point-to-Point Communication Primitives - MPIPCL

- a layered library built on top of MPI persistent point-to-point functionality
- enable exploration of the partitioned communication model in applications and libraries
- allows early adopters of partitioned communication to use the library now to refactor their threaded MPI point-to-point applications with the new partitioned communication semantics while main-stream MPI implementations work to integrate the new APIs in future releases
- Integration with ExaMPI underway

Path to Upgraded Production, I

Should we do these things?

1. Improve Trilinos and other frameworks with better MPI usage?
2. What frameworks will have most leverage for NNSA applications – which to target?
3. How much value is attached to getting “spaghetti code” out of applications by providing higher-level primitives?
4. Should we provide our own “MPI+” library that’s available with any MPI but faster with ExaMPI to begin with?

Path to Upgraded Production, II

Introducing new primitives

1. Break abstraction? Build new ones?
2. Introduce partitioned, persistent directly in apps?
3. Target a framework?
4. If app doesn't use a framework, retarget to framework?
5. Support "both old and new" modes?
6. Support "easy" revalidation…
7. How to justify "large changes"?

Example: EMPIRE Suite of applications