Can There Be a Common Communication Runtime System?

William Gropp and Darius Buntinas
Mathematics and Computer Science Division
Overview of Pros and Cons

- **Pro**
  - Share development work
  - Encourage interoperability of programming models
  - Provide portability for HPCS languages (ubiquity)

- **Con**
  - Match to programming model (duplicate the “MPI effect” — constrain models into CCS semantics)
  - Match to hardware (particularly hardware that is expensive to emulate in software, e.g., full/empty bits or remote atomic updates)
  - Runtime overhead may be unsuitable for load-store operations
  - RISC vs CISC (small and simple vs large and rich)

Why develop a common runtime?

Short answer:
- Maybe …
Some Issues

- What memory may be used in zero-copy mode?
  - Special memory? Statically allocated memory? Stack?
  - Alternately, which classes of RMA memory does the programming model require:
    - RMA memory defined collectively at init time
    - RMA memory defined collectively at any time
    - RMA memory defined non-collectively at any time
    - All of process memory

- How are remote addresses specified?
  - Require “symmetric allocation”?
  - Prior initialization?

- Are stores ordered? What is the consistency model?

- Is the model scalable? Is it scalable to subsets of processes (teams)?

- What data alignments are supported efficiently?

- Are there remote atomic operations? Fetch and increment? Compare and swap? Load-link/store conditional? Queue insert and extract?

- How is progress managed (polling verses interrupt/non-polling/thread/separate hardware)?

- We examined these issues and others for MPI in the context of some existing runtime systems
  - These other systems are well-optimized for their programming models
  - This illustrates some of the challenges in a common model — the devil is in the details
Motivation

- We worked on implementing a hybrid MPI-UPC programming environment
  - Port MPICH2 over the GASNet communication subsystem
  - GASNet couldn’t efficiently support all that was needed by MPICH2
    - *And MPI can’t efficiently support what is needed by UPC*
- While there are many common features
  - E.g., RMA operations, bootstrapping
- Communication subsystems are typically designed to support a specific middleware library or runtime system
- Previously analyzed the requirements of various programming model middleware and the communication subsystems that support them
  - There are no existing communication subsystems that efficiently support all middleware
  - There are no mutually exclusive requirements
Software Layers of a High-Performance Computing System

- **Application**
- **Middleware**
  - (MPICH2, GA Toolkit, UPC Runtime)
- **Communication Subsystem**
  - (ARMCI, GASNet, Portals)
  - Shared Memory
  - GM
  - IBA
  - QSNet
Design Issues for Communication Subsystems for MPI

Required features (for the MPI programming model)
- Remote Memory Access operations
- MPI-2 RMA support
- GAS language and remote-memory model support
- Efficient transfer of large MPI two-sided messages

Desired features
- Active messages
- In-order message delivery (to simplify support for MPI “envelope” ordering)
- Noncontiguous data (not just contiguous or strided)
### Summary of Features Supported by Current Communication Subsystems

<table>
<thead>
<tr>
<th>Feature</th>
<th>ARMCI</th>
<th>GASNet</th>
<th>LAPI</th>
<th>Portals</th>
<th>MPI-2</th>
<th>Noncontiguous data</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMA operations</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>V,S</td>
<td>●</td>
</tr>
<tr>
<td>MPI-2 active-mode RMA</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>MPI-2 passive-mode RMA</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>GAS language support</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Transfer of large MPI messages</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>In-order message delivery</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>

* V = I/O vector; S = strided; B = block-indexed
An Example Communication Subsystem – CCS

- CCS (Common Communication Subsystem) is based on
  - Nonblocking RMA operations
    - For efficient data transfer
  - Active messages
    - For small messages, control and invocation of remote operations

- Outline
  - Active messages
  - Remote memory access operations
  - Efficient transfer of large MPI two-sided messages
  - In-order message delivery
  - Noncontiguous data
Active Messages

- CCS provides active messages
  - Sender specifies handler function with parameters
    - Handler is executed on receiver when message is received
  - Provide flexibility to upper layer developers
  - Intended for small messages, so should be optimized for latency

- Depending on implementation, handlers will be called from within a CCS function, or asynchronously
  - CCS provides locks which can safely be called from within handlers
  - CCS provides a mechanism to prevent a handler from interrupting the current thread

- CCS allows multiple upper layer libraries to use CCS at the same time
  - Each library allocates a context
    - Uniquely identifies a set of handler functions
Remote Memory Access Operations

- CCS provides nonblocking RMA operations
  - Use the interconnect’s native RMA operations to maximize performance
  - If native RMA operations are not available, use active messages
    - E.g., Get: active message + put
  - Support for GAS language and remote-memory models
    - Concurrent accesses are allowed

- CCS uses callback functions for completion notification
  - A callback function pointer and parameter are specified in the call to the RMA operation
  - The callback is called when the RMA operation completes remotely
  - This can be used to implement fence operations

- Lower-level interconnect libraries have different requirements for RMA memory
  - CCS provides different functions to meet these requirements
    - Registration of existing memory to be used for RMA
    - RMA memory allocation
RMA Memory

- Registration of existing memory to be used for RMA
  - Most user-level communication libraries require registration
  - CCS will manage which pages to register with communication library
    - *Communication library may limit the number of registered pages*
    - *Not all pages registered with CCS need be registered with the communication library*
  - Note that there are many well-known problems with user-mode registration caches (if user/OS/middleware releases memory)

- Allocation of RMA memory
  - Some architectures don’t support registration of existing pages
    - *E.g., Solaris can’t pin existing pages*
  - What if the implementation communicates using shared memory?
    - *Can’t make existing memory shared memory*
  - CCS provides methods to allocate RMA memory
    - *E.g., allocate a shared memory region to which others can attach*
Noncontiguous Data

- CCS supports noncontiguous data using *datadescs*
  - Similar to MPI Datatypes
  - Defined recursively
    - *But unrolled into component loops rather than use recursive procedure calls*
  - Basic datadescs
    - *Contiguous*
    - *Vector – blocks of data at regular intervals*
    - *Struct – like a C struct of different datadescs*
    - *Indexed – similar to I/O vector*
    - *Block-indexed – like indexed, but each segment is the same length*

- Datadescs
  - Along with native datatype info (e.g., int, double) can be used to implement MPI Datatypes
  - LAPI I/O vectors can be implemented with *Indexed datadesc*
  - ARMCI
    - “*Strided” can be implemented with Vector datadesc*
    - “*Vector” can be implemented with Indexed datadesc*
Preliminary Performance Results (over GM2)

Latency

- CCS
- GASNet
- ARMCI

4-Byte latencies:
- GASNet 8.8 µs
- CCS 9.6 µs
- ARMCI 10.8 µs

Max bandwidth:
- GASNet 242 MBps
- CCS 244 MBps
- ARMCI 238 MBps

Bandwidth
Implications for a Common Runtime System

- A “classic” runtime library is unlikely to satisfy all needs
  - There may be too many differences at both the hardware and programming model level to bridge while maintaining performance
  - We have an example in the BLAS and sparse BLAS
    - BLAS for small matrices slower than simple Fortran code
      - Overhead dominates for latency-sensitive sizes
    - Sparse BLAS have had little impact
      - Rich but still a mismatch to hardware and/or “programming model” (application data structures)

- What can we do?
  - After all, BLAS are useful in the right place …
Some Steps Toward a Common Communication Runtime

- Like the beginnings of MPI, there are a number of high-quality systems targeting different parts of the general space
- Methods could be shared for specific operations
- Initialization of runtime systems could be arranged to allow different systems to interoperate
- Source “templates” could be used as “executable documentation” of best practice and used as input in creating custom runtimes
- An extensible common core could be defined
  - Define required architectural abilities
    - Part of MPI RMA model complexity results from accommodating non-cache-coherent systems; other complexity from weak consistency model
  - Consider allowing several “progress” alternatives
    - Picking one model is guaranteed to drive away some systems
  - Consider following the graphics engine model (basic ops plus optional special features)
  - Start from scratch (don’t start from anyone’s existing system)
- No matter what you do, by definition it will be a Greatest Common Denominator system