SAND2015-1946C

Evolving the message passing programming model via a fault-tolerant, object-oriented transport layer

Jeremiah Wilke
Sandia National Laboratories
Scalable Modeling & Analysis
Livermore, CA 94550
jjwilke@sandia.gov

Keita Teranishi
Sandia National Laboratories
Scalable Modeling & Analysis
Livermore, CA 94550
knteran@sandia.gov

Janine Bennett
Sandia National Laboratories
Scalable Modeling & Analysis
Livermore, CA 94550
jcbenne@sandia.gov

Hemanth Kolla
Sandia National Laboratories
Scalable Modeling & Analysis
Livermore, CA 94550
hnkolla@sandia.gov

David Hollman
Sandia National Laboratories
Scalable Modeling & Analysis
Livermore, CA 94550
dshollm@sandia.gov

Nicole Slattengren
Sandia National Laboratories
Scalable & Secure Systems
Livermore, CA 94550
nlslatt@sandia.gov

ABSTRACT

In this position paper, we argue for improved fault-tolerance of an MPI code by introducing lightweight virtualization into the MPI interface. In particular, we outline key-value store semantics for MPI send/recv calls, thereby creating a far more expressive programming model. The general message passing semantics and imperative style of MPI application codes would remain essentially unchanged. However, the additional expressiblity of the programming model 1) enables the underlying transport layer to handle fault-tolerance more transparently to the application developer, and 2) provides an evolutionary code path towards more declarative asynchronous programming models. The core contribution of this paper is an initial implementation of the DHARMA transport layer that provides the new, required functionality to support the MPI key-value store model.

1. INTRODUCTION

In high performance computing applications, message passing (MPI) has long been the predominant communication model, and checkpoint/restart has been the primary resilience strategy. However, as we look ahead to extreme-scale system architectures, traditional checkpoint/restart is no longer a viable solution due to the projected increase in system faults combined with limitations in I/O capabilities. As a result, a number of more transparent, on-line recovery mechanisms are being developed including uncoordinated checkpoint/restart techniques such as LFLR [17], Fenix [7], SCR [13], and FMI [16].

While these strategies strive towards improved resilience, key features of the current MPI model can make both transparent and user-level fault-tolerance difficult [11]:

1. Any MPI call can fail. The application must determine each time if a failure occurred and how to address it.
2. MPI matching is based on tags and message order. While tags can define unique sends, MPI is not very expressive about the logical meaning of a send.
3. MPI enforces ordered message arrival. Recovering a send/recv requires that the spare node agrees on the message order when it swaps in, likely requiring message logging for correctness.
4. Even if only a few communications are with a failed process, the live process must still roll back to ensure the messages replay.
5. Uncoordinated checkpoint restart, despite its conceptual appeal to avoid expensive global checkpoints, can create a rollback cascade [9]. This is fundamentally due to the rollback problem mentioned above cascading to other nodes and multiple checkpoint intervals.

Consequently, a controversial topic is the future of MPI in light of resilience concerns at exascale. The question is two-fold. First, is the MPI model fundamentally flawed for fault-tolerance? Would alternative programming models, such as Legion [1] or Charm++ [8], provide a path forward for addressing the fundamental fault-tolerance challenges in MPI (in addition to performance improvements via greater over-decomposition/task parallelism)? Second, if MPI remains the dominant model, what extensions are necessary for fault-tolerance both in the API and on the backend?

A pleasant simplification for MPI is maintaining a constant number of workers despite failures. Since great care is often taken to partition the problem assuming a fixed number of workers, adapting to a decreasing node count can be challenging. Both LFLR (local failure, local recovery) [17] and FMI (fault-tolerance interface) [16] have shown promise by checkpointing process state and migrating processes to spare nodes upon failure. Because of the problems outlined above, implementing fault-tolerant codes in the non-shrinking model has proven difficult [11].

In this paper, we propose key value-store semantics as a simple extension to message passing in order to facilitate fault-tolerance in the non-shrinking model employed by LFLR. To illustrate how these semantics address the underlying challenges outlined above, let us consider exactly what an MPI send/recv is meant to accomplish in most scientific codes. The data being sent/received usually have a unique physical meaning (e.g. mesh block 0,1,5). The same is true even of collectives, e.g., where you gather a resource "subset" into a resource "whole" or reduce a resource "contribution" into an aggregated "global sum." Rather than simply an integer tag and a pointer, much more can be expressed about what

the send/recv or collective is doing with a *single* additional parameter to existing function calls. For some operation, a vector A might need to be exchanged:

```
Node 0:
double* vecA = ....
MPI_Send(vecA, count,
  MPI_DOUBLE, node1,
  tag, MPI_COMM_WORLD);

Node 1:
double* vecA = ...
MPI_Recv(vecA, count,
  MPI_DOUBLE, node0,
  tag, MPI_COMM_WORLD);
```

Consider, instead, an altered version:

```
Node 0:
double* vecA = ...
MPI_Publish(vecA, "vector A");
MPI_Barrier();

Node 1:
double* vecA ...
MPI_Fetch(node0, "vector A",  vecA);
MPI_Barrier();
```

Now, Node 0 publishes a named array instead of sending, thus making the `MPI_Fetch` transaction uniquely defined. If Node 0 fails before Node 1 has fetched, the transport layer for Node 1 can automatically migrate the request to the spare node. As soon as the resource is created on either Node 0 or its replacement, it is returned to Node 1 without any user-level intervention. Because transactions are given a unique logical identifier, such MPI functions never fail - they simply get re-issued or delayed, drastically simplifying user-level code. The most logical data structure to manage fetches is a key-value store that maps strings to buffers.

Simply by having more expressive function calls, we largely avoid issues 1-3 above. Publish/fetch resembles partitioned global address space (PGAS) models [20]; however, MPI one-sided and PGAS only reference physical pointers instead of logical identifiers. If transactions directly express the *physical operation* to perform, they lack the logical context or virtualization to automatically migrate to backup nodes (and do so without rolling back non-failed processes). The approach is neither user-proof nor a black box, as indicated by the barrier; synchronization is still required to ensure proper versions are returned and resources are not prematurely garbage collected. We, therefore, do not eliminate fault-tolerance problems 4 and 5 above, but instead change them into application-level synchronization problems.

These modest modifications of the existing API require new functionality from the underlying transport layer:

1. Fault-tolerant collectives (global agreement)
2. Flexible active messages via C++ object migration
3. Well-defined reliability semantics for send/recv
4. Asynchronous progress threads

While GASNet [3] supports one-sided gets and remote procedure calls and ULFM [2] contains fault-tolerant collectives, to our knowledge no packages provide both the asynchronous and fault-tolerance support needed.

In this work we 1) introduce the DHARMA transport layer with initial Cray uGNI implementation, and 2) explore the overheads associated with a key-value store message protocol, arguing it introduces no extra network traffic. Initial results suggest transfers might be more efficient through better pinned memory registration. The present work does not implement or rigorously define a new API. Instead, we argue for and implement a new transport layer, outlining the API features it enables. We thereby hope to facilitate experimentation and prototyping of API extensions.

2. RELATED WORK

Recent fault-tolerance extensions to MPI have been added in the user-level fault mitigation (ULFM) library [2] to determine failed processes and subsequently validate or revoke MPI communicators. LFLR [17] and FMI [16] provide a well-defined recovery procedure for MPI by migrating processes from failed nodes to a pool of spares. Fault-tolerant collectives have been explored in detail elsewhere, both for collectives such as all-reduce [19] that exists in the MPI standard as well as new agreement collectives for determining which processes have failed [10].

Key-value (KV) stores are similar to coordination languages like Linda [6] and recent tuple-space programming models like concurrent collections [5]. NSSI provides flexible RDMA for KV-like data management [14]. The more declarative programming model Legion [1] uses logical identifiers to describe programs decoupled from their physical layout in memory. Charm++ operates by passing messages between distributed objects called chares [8] that also define a logical data dependency whose physical location can migrate.

Message logging has been studied in detail for fault-tolerant MPI [4], particularly for uncoordinated checkpoint/restart [9]. More general studies of uncoordinated checkpoint/restart have also been performed, analyzing the performance tradeoffs and optimal checkpoint intervals.

3. TRANSPORT LAYER

Here we outline the DHARMA transport layer (Distributed asyncHronous Adaptive Resilient Management of Applications), a name accompanying a group of related programming models and fault-tolerance work. The acronym Distributed Hash Arrays for Remote Memory Access has been used equivalently, emphasizing distributed key-value stores as fundamental to the programming model.

In the DHARMA transport layer, there are no global or `extern` functions. Instead, an object of type `transport` is created and member functions are called. The exact implementation for different platforms is encapsulated in virtual functions. The transport uses a `message` class that holds some basic, universal information:

- The sender and receiver ranks
- The message type (e.g. header, RDMA get, ACK)
- The message class (point-to-point or part of collective)
- What sort of ACK should be generated

More specific classes can inherit from `message` in general use. An internal serialization library (sprockit, Sandia Productivity C++ Toolkit) transmits C++ objects between nodes.

An asynchronous progress thread is the recommended usage for the DHARMA transport. With many-core architectures becoming the norm, MPI everywhere may no longer be an optimal model. With an excess of CPU cores, there should generally be enough CPU resources to support such a thread. The transport layer provides a basic function for querying incoming messages, which runs the internal progress engine for the transport layer until a message is received:

```
message::ptr blocking_poll();
```

3.1 Point-to-point Primitives

3.1.1 Eager Sends

Point-to-point exchanges send C++ objects inheriting from the `message` class. For small messages, the transport provides two functions that explicitly distinguish between messages sent with actual data (eager payloads) and with headers that coordinate RDMA transactions.

```
void send_smsg_payload(int dst,
  message::ptr msg, bool ack);
void send_rdma_header(int dst,
  message::ptr msg, bool ack);
```

Here `message::ptr` is a reference counted pointer typedef and `dst` is the rank of the destination (receiver) of the message. Importantly, the functions allow optional send ACKs to be generated upon delivery to the destination endpoint. Consider the following application code snippets:

```
Node 0:
send_smsg_payload(node1, msg, true/*ack*/);
message::ptr ack = blocking_poll();

Node 1:
message::ptr payload = blocking_poll();
```

Here, Node 0 sends a message with ACK request. After delivering the message, it receives the ACK notification. The exact type returned by `blocking_poll` can be queried:

```
switch(msg->type()) {
 case smsg_ack: /* ACK for a smsg send */
 case rdma_get_ack:
 case rdma_get_payload:
```

Node 0 receives an exact replica of the `message` object sent (no large buffers are copied), but the content type is changed to `smsg_ack`. All the metadata contained in the original message is, therefore, available in the send ACK and, in most usages, the original message is reused as the ACK, requiring no extra copies. Similar to Node 0, Node 1 starts polling and receives a message when the payload arrives, but now with content type `eager_payload`.

3.1.2 RDMA Sends

RDMA operations are part of the transport layer API and can have both send and receive ACKs:

```
void rdma_put(int dst, message::ptr msg,
  bool needs_send_ack, bool needs_recv_ack);
void rdma_get(int src, message::ptr msg,
  bool needs_send_ack, bool needs_recv_ack);
```

For short messages, `needs_recv_ack` must always be true to alert the receiver of an incoming message. For most RDMA operations, notifications should occur on both the receive and send side. However, certain cases may have relaxed reliability requirements. For an RDMA put, e.g., once the data arrives at its destination, an `rdma_put_ack` message is delivered to the sender and an `rdma_put_payload` message is delivered to the receiver, which will reside in a queue until retrieved by `blocking_poll`. The send ACK and receive ACK are exact replicas of the message object originally passed to the `rdma_put` function, aside from buffer copies.

The `message` object offers virtual functions for retrieving opaque `public_buffer` handles encapsulating both a `void*` and metadata for pinned memory needed for RDMA:

```
virtual public_buffer& local_buffer();
virtual public_buffer& remote_buffer();
```

While a default `rdma_message` exists, arbitrary message types are possible. The `transport` provides implementation-dependent functions for allocating or using existing allocations:

```
virtual public_buffer
allocate_public_buffer(size_t size);

virtual public_buffer
make_public_buffer(void* buffer, size_t size);
```

Equivalent functions exist for deallocation. Explicit casting can convert a `public_buffer` to its underlying `void*`.

3.2 Collectives

3.2.1 Global agreement

The main collective underlying fault tolerance is a global agreement algorithm wherein each process votes on which processes are alive [10, 19]. For recovery mechanisms such as the LFLR model, it is very important that all nodes agree exactly on who has failed. In this way, all nodes can execute process migration coherently and consistently. If nodes began the recovery process with different global views, they might assign different virtual ranks to the spare nodes.

The global collective algorithm essentially matches that published by Naughton [10]. A binary tree is constructed from the processes (Figure 1), which first vote up to their parent. Once the parent has received votes from all its children, it votes to its parent. Votes are eventually merged at the root. Once the root decides on a final outcome, the result is sent back down the tree. Even if new failures are detected during the down vote, they are temporarily ignored. The merged vote must be declared final so that all live processes exit the collective agreeing on the exact failure set.

To detect failures, the transport layer implements

```
bool ping(int dst, timeout_function* func);
```

The function immediately returns a boolean if a failure is already known; otherwise, an arbitrary timeout function is registered and the node is monitored. If the collective completes and no failure is detected, a `cancel_ping` function unregisters the timeout. If, however, a failure is detected the timeout function is invoked. In the global agreement algorithm, the timeout function reconnects the binary tree (Figure 1), re-establishing parent-child connections around the failure. This means a parent may now connect to its grandchildren, resulting in more than two connections.

The global agreement actually suffers from the Byzantine generals problem [12]. Once votes are passed down the tree, a parent node has no way of ever guaranteeing that the vote reaches the leaf nodes at the bottom. Every global agreement collective that runs is therefore given an epoch or tag. The result of the collective must be stored until the

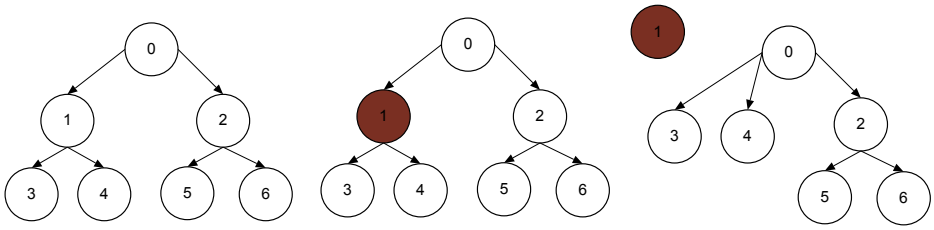

Figure 1: Demonstration of binary tree voting collective. Here Node 1 fails, forcing the tree to reconnect and reissue votes.

next collective epoch since, due to failures, grandchildren or great-grandchildren may eventually connect to the parent and require the down-vote be re-sent.

An added benefit of the DHARMA transport is a mechanism for simulating and detecting faults on current platforms that may not permit failed nodes within a job. An efficient mechanism for error detection is an RDMA get operation from a dummy array [18]. Through timeouts or RDMA NACKs, the RDMA get fails when the node fails and can no longer be reached. The DHARMA transport can emulate failures (even when systems software does not support it) by providing a `die()` member function. After calling `die()`, the transport drops all incoming messages. Even though RDMA get pings will successfully return without timing out, the value in the dummy array can indicate an emulated failure. Higher level application runtimes can therefore easily develop and test fault mitigation/recovery strategies by using the DHARMA transport layer regardless of systems software support. Large scale collective benchmarks have been run on the Cray XC30 with emulated failures, but we delay discussing them here given space constraints.

3.3 Standard collectives

Even for asynchronous execution or over-decomposed problems, optimized collective communication remains important in many algorithms. DHARMA provides collectives, but they are fundamentally non-blocking. Just as `message` notifications are delivered via `blocking_poll` for point-to-point, collectives generate a `collective_done_message`. We delay a detailed discussion of collectives. We remark, however, that MPI collectives could also be given string labels to give them a logical context.

4. IMPLEMENTING MPI EXTENSIONS VIA THE DHARMA TRANSPORT LAYER

4.1 Basic Protocol

Here we outline the code necessary to execute the `MPI_Publish` and `MPI_Fetch` functions demonstrated in the introduction. We create a new message class, `mpi_kv_message`, inheriting from `message` and implementing the RDMA interface. We demonstrate code for C++, but these could easily be extended for C bindings.

```
map<string, public_buffer> resource_map;
multimap<string, mpi_fetch_t*> fetches;

void
MPI_Publish(transport* tp, int dst,
 string name, void* buf, int size){
 public_buffer pbuf  = tp->make_public_buffer(...);
 resource_map[name] = pbuf;
}

void
MPI_Fetch_resource(transport* tp, int dst,
 string name, void* buf, int size){
 mpi_fetch_t* fetch = allocate_fetch_request();
 fetches[name] = fetch;
 mpi_kv_message::ptr header = allocate_kv_header();
 public_buffer pbuf = tp->make_public_buffer(...);
 header->remote_buffer() = pbuf;
 header->name = name;
 tp->send_rdma_header(header);
 wait_fetch_request(fetch);
}
```

The initial `MPI_Publish` call does no communication - only a local KV-store operation. To complete a send, the asynchronous progress thread on the publisher requires:

```
void process_kv_req(transport* tp,
  mpi_kv_message::ptr rdma_header){
 string name = rdma_header->name;
 public_buffer pbuf = resource_map[name];
 rdma_header->local_buffer() = pbuf;
 int src = rdma_header->sender();
 tp->rdma_put(src, rdma_header, true, true);
}

void progress(transport* tp){
  message::ptr next = tp->blocking_poll();
  /** cast or switch to determine type */
  process_kv_req(rdma_header);
}
```

The publisher receives an RDMA header requesting the resource, which is looked up and returned via a direct RDMA put into the receive buffer. For simplicity, we assume the publish happens before any requests arrive. Handling requests for data not yet published requires straightforward use of callbacks with more complicated data structures.

The progress thread on the fetching node runs:

```
void process_kv_payload(transport* tp,
  mpi_kv_message::ptr rdma_payload){
 for (fetch in fetches[rdma_payload->name]){
  fetch->complete();
 }
}

void progress(transport* tp){
  message::ptr next = tp->blocking_poll();
  /** cast or switch to determine type */
  process_kv_payload(rdma_payload);
}
```

Here, a receive ack with information for the RDMA payload is delivered. The progress thread determines any waiting fetch objects and notifies them of completion.

4.2 Transparent Fault-Tolerance

Because the RDMA request is routed through a special function, `send_rdma_header`, the DHARMA transport layer can track any pending RDMA transactions associated with a `message`, allowing it to reissue RDMA headers to the spare node. This is fundamentally different from standard message logging because only *outstanding* RDMA transactions remain logged. Completed transactions can be erased since recovery does not require message replay.

We do not get these fault-tolerance advantages for free, however. We have pushed the difficulty into an application-level synchronization problem. The application must ensure that `MPI_Fetch` is never called for resources that have been deleted or cannot be recovered from checkpoints. The problem has shifted to garbage collection, which we suggest is more intrinsically application-specific and therefore more appropriate for the user level, maintaining a clean abstraction of a fault-free transport layer. The proposed model's feasibility will depend on how difficult such garbage collection is to implement in practice for example applications.

4.3 Flexible Data Requests

In many cases, an `MPI_Send/Recv` pair fetches a data subset such as ghost cells or specific vector elements for sparse matrix-vector multiplication. An entire vector or mesh region may be passed to `MPI_Publish` when only a small region is actually needed by `MPI_Fetch`. Because arbitrary C++ objects can be sent as RDMA requests, we can create arbitrary `subset_kv_req_message` classes. The prototype for `MPI_Fetch` could be extended with subset definitions, similar to an `MPI_Type`. The same RDMA protocol described in 4.1 can be used, but instead return only a subset. This again increases the *expressiveness* of the programming model, giving sends and receives deeper logical meaning.

4.4 Declarative and Task-DAG Models

The MPI extensions do not change the imperative, sequential nature of message passing code. There is still no explicit dependency graph - the DAG is implicit in the MPI calls. However, the DHARMA transport layer offers a gateway for evolving codes towards task-DAG models. The code is more expressive, creating named dependencies. Once the code is expressed via logical rather than physical dependencies, the transition from imperative to declarative is far easier.

5. RESULTS

We demonstrate basic performance results on a Cray XC30 dragonfly testbed (Volta) using a uGNI-based version of the DHARMA transport layer [15]. We compare to the default MPICH implementation for uGNI. We do not consider this a rigorous performance comparison - only an indication of performance trends. In Figure 2 we examine the performance of a simple point-to-point exchange.

The point-to-point sends use a put protocol:

1. Destination sends RDMA header to source
2. Source receives header and issues RDMA put
3. Source sends ACK to destination when put completes

The timings have both sender and receiver waiting on a completion ACK. MPICH2 on Cray platforms employs a similar protocol, but with the receiver issuing an RDMA get.

Figure 2 demonstrates the effects of pinned memory. The DHARMA put protocol re-registers memory buffers for each send, resulting in drastically reduced throughput relative to the pre-registered put that pins all buffers beforehand. If buffers are re-used in the benchmark, MPI can cache RDMA memory registrations. Buffer re-use is staggered such that L1/L2 caches are cold, but MPI still hides pinned memory latency with a "hot cache" of RDMA buffers. Without buffer re-use, a performance drop-off is seen in Figure 2 with a "cold" RDMA registration cache. For small messages, even without caching, MPI achieves low latency. MPICH2 uses a hybrid RDMA-eager protocol. The sender copies eagerly into a pre-registered buffer, allowing the send to complete early and avoid registration overheads.

Because RDMA sends coordinate via `send_rdma_header`, DHARMA can leverage uGNI hardware ACKs instead of explicit software ACKs. Figure 2 shows a significantly decreased latency with hardware ACKs.

6. CONCLUSIONS

In this work, we suggest how minor modifications to the MPI API could make recovery mechanisms such as LFLR even more transparent to the application developer. While the current MPI standard use pointers and tags, we argue that key-value semantics create a far more expressive programming model. The additional expressiveness not only enables the transport layer to handle fault-tolerance in a manner that is more transparent to the application developer, it also provides a natural evolutionary code path towards more declarative alternative programming models. We present the portable DHARMA transport layer API based on flexible C++ objects along with an initial uGNI implementation to support these extensions. Competitive performance with Cray MPICH2 for point-to-point sends is observed, showing that the `MPI_Fetch` and `MPI_Publish` semantics still allow high performance. Lastly, the DHARMA transport layer will be made publicly available upon passing open-source review.

Acknowledgment

The authors would like to thank Craig Ulmer, Gary Templet, and Abhinav Vishnu for useful discussions. This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

7. REFERENCES

[1] M. Bauer et al. Legion: expressing locality and independence with logical regions. In *SC '12:*

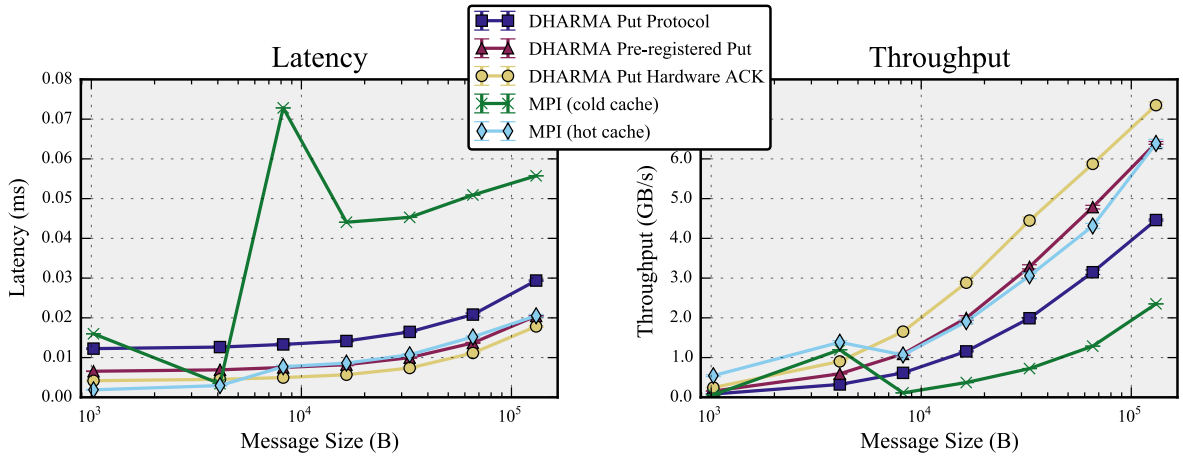

Figure 2: Total latency and throughput for varying message sizes and protocols for point-to-point sends. Protocols differ on RDMA memory registration and use of hardware/software ACKs. DHARMA uses a put protocol (see text) which either registers RDMA buffers on-the-fly or pre-registers all buffers. MPI uses a get protocol and is explored in "hot cache" mode reusing RDMA memory registrations or "cold cache" mode re-registering buffers on each send.

International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2012.

[2] W. Bland et al. An Evaluation of User-Level Failure Mitigation Support in MPI. In *Recent Advances in the Message Passing Interface*. Springer Berlin Heidelberg, 2012.

[3] D. Bonachea. GASNet Specification, v1.1 U.C. Berkeley Tech Report (UCB/CSD-02-1207). 2002.

[4] A. Bouteiller et al. Reasons for a Pessimistic Or Optimistic Message Logging Protocol in MPI Uncoordinated Failure, Recovery. In *CLUSTER*, pages 1–9, 2009.

[5] M. G. Burke et al. The Concurrent Collections Programming Model. *Technical Rep. TR 10 12*, 2010.

[6] N. J. Carriero et al. The Linda Alternative to Message-Passing Systems. *Parallel Comput.*, 20:633–655, 1994.

[7] M. Gamell et al. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *SC '14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906, 2014.

[8] Z. Gengbin, S. Lixia, and L. V. Kale. FTC-Charm++: An in-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *CLUSTER*, pages 93–103, 2004.

[9] A. Guermouche et al. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications. In *IPDPS*, pages 989–1000, 2011.

[10] J. Hursey et al. A Log-Scaling Fault Tolerant Agreement Algorithm for a Fault Tolerant MPI. In *Recent Advances in the Message Passing Interface*. Springer Berlin Heidelberg, 2011.

[11] I. Laguna et al. Evaluating User-Level Fault Tolerance for MPI Applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 57–62, 2014.

[12] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Program. Languages Syst.*, 4:382–401, 1982.

[13] A. Moody et al. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC '10: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[14] R. A. Oldfield et al. Trilinos I/O Support Trios. *Sci. Program.*, 20:181–196, 2012.

[15] H. Pritchard, I. Gorodetsky, and D. Buntinas. A uGNI-based MPICH2 Nemesis Network Module for the Cray XE. In *18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, pages 110–119, 2011.

[16] K. Sato et al. FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In *IPDPS*, pages 1225–1234, 2014.

[17] K. Teranishi and M. A. Heroux. Toward Local Failure Local Recovery Resilience Model using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 51–56, 2014.

[18] A. Vishnu et al. Fault-tolerant Communication Runtime Support for Data-Centric Programming Models. In *HiPC*, pages 1–9, 2010.

[19] J. Wilke et al. Extreme-Scale Viability of Collective Communication for Resilient Task Scheduling and Work Stealing. In *FTXS at Dependable Systems and Networks (DSN)*, pages 756–761, 2014.

[20] C. Xie, Z. Hao, and H. Chen. X10-FT: Transparent Fault Tolerance for APGAS Language and Runtime. In *PMAM 2013: International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 11–20, 2013.