Accelerating Messages by Avoiding Copies in an Asynchronous Task-based Programming Model

Nitin Bhat[†], Sam White[*], Evan Ramos[†], Laxmikant V. Kale[*†]
[†]Charmworks, Inc., Urbana, Illinois, USA
[*]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
Email: nitin@hpccharm.com, white67,@illinois.edu, evan@hpccharm.com, kale@illinois.edu

***Abstract*—Task-based programming models promise improved communication performance for irregular, fine-grained, and load imbalanced applications. They do so by relaxing some of the messaging semantics of stricter models and taking advantage of those at the lower-levels of the software stack. For example, while MPI's two-sided communication model guarantees in-order delivery, requires matching sends to receives, and has the user schedule communication, task-based models generally favor the runtime system scheduling all execution based on the dependencies and message deliveries as they happen. The messaging semantics are critical to enabling high performance.**

In this paper, we build on previous work that added zero copy semantics to Converse/LRTS. We examine the messaging semantics of Charm++ as it relates to large message buffers, identify shortcomings, and define new communication APIs to address them. Our work enables in-place communication semantics in the context of point-to-point messaging, broadcasts, transmission of read-only variables at program startup, and for migration of chares. We showcase the performance of our new communication APIs using benchmarks for Charm++ and Adaptive MPI, which result in nearly 90% latency improvement and 2x lower peak memory usage.

***Index Terms*—Charm++, AMPI, RDMA, Parallel Programming, Asynchronous Tasking, Communication Optimizations**

I. Introduction

Exascale systems are composed of increasingly powerful compute nodes composed of many-core CPUs and multiple GPUs. This in turn stresses the network to be able to keep up with the fast compute capability. Asynchronous and efficient data movement is key to overall application performance. Task-based models excel in their ability to overlap communication with computation and to intelligently schedule work so as to avoid busy waiting on the network. But the communication performance of such models does not necessarily translate across all use cases.

At the same time as task-based programming models are becoming increasingly popular, hardware systems and applications are both evolving. Remote Direct Memory Access (RDMA) support is commonplace in HPC networks, and being able to exploit it is key to communication performance. RDMA enables communication of data in-place, without the involvement of the CPU in the transfer. Applications are also becoming more complex and demand efficient support for a wide array of communication patterns and message sizes. In light of both these trends, we re-examine Charm++'s communication semantics.

II. Background

A. *Charm++*

Charm++ is an asynchronous parallel programming model and runtime system based on the idea of migratable C++ objects called *chares*, [1] that interact with each other by sending messages. Chares execute in parallel on Processing Elements (PEs, typically referring to CPU cores or nodes) as scheduled by the runtime system. The object-centric approach also enables *overdecomposition*, where the problem domain is decomposed into a larger number of chares than the number of available PEs empowering the adaptive runtime system to overlap computation and communication, and dynamically balance load. Program execution in Charm++ is driven by interactions between chares that occur through asynchronous *entry method* invocations. An entry method is a remotely invocable version of a local function call performed on an object where the caller does not wait for a reply or a return value from the function. Entry method invocations are Charm++'s equivalent of a task, they execute without interruption on the arguments passed into them and any chare-owned state. For each entry method invocation, the runtime system serializes the user passed data creating a message on the sender chare's PE and sends it to the PE of the receiver chare. On each PE, incoming messages are stored in a message queue, using which a scheduler regularly iterates over these messages and executes the appropriate entry method on the receiver chare. A chare gets execution time on the PE when its message has been picked up by the scheduler and that corresponding entry method is executed. The entire control flow of a Charm++ program happens through chares generating entry methods invocations, which in turn can create other entry method invocations until a special exit routine is invoked by any of the chares, which causes program termination.

B. *Motivation for a Zero Copy Messaging Model*

In Charm++, interactions between chares are performed using entry method invocations, which are carried out traditionally using two messaging models: a) Parameter Marshalling, and b) Custom Messages.

Parameter marshalling is used when parameters are passed in an entry method invocation and a message is created internally by the runtime system on behalf of the user. On the sender side, the marshalling of these parameters requires the runtime system to copy individual parameters passed by the user into a single contiguous buffer which is sent across the network as a message. This is done to ensure safe reuse or freeing of the passed parameters after the entry method call. On the receiver side, the parameters are unmarshalled out of the message and passed to the entry method. This allows the runtime system to use pointers directly to contiguous parameters in the message and pass them to the entry method without copying them. To ensure safe memory management of this message, i.e. to avoid a potential memory leak, the message is freed by the runtime system after the entry method completes. For this reason, these parameters have a lifetime only until the end of the scope of the entry method. In order to use the received data beyond this scope, the user must copy the data into their own data structure. Therefore, two copies are required for sending data from a chare to another remote chare: one at the sender side and another at the receiver side.

The other messaging model is to use Custom messages. Custom messages are data structures that inherit from a base message data structure and encapsulate all the parameters required by an entry method. The key difference between parameter marshalling and custom messages is with regard to ownership of the buffer. On the sender side, after invoking an entry method with a custom message, the runtime system takes ownership of the message and the user cannot access the encapsulated parameters or reuse that buffer. Additionally, unlike parameter marshalling, the semantics of not allowing buffer reuse allow the runtime system to avoid making copies on the sender side. On the receiver side, the ownership of the received message is given to the user. This allows the user to directly use the received message as if it were the user's data structure without requiring an additional copy. However, if the user forwards the received message to another chare, the message ownership is again handed back to the runtime system and the buffer is inaccessible. For iterative applications, using this model requires that the user make a new allocation and copy of the custom message for each iteration, which in turn adds to the cost of communication.

In order to transfer data from one chare's buffer to another chare's buffer, both the traditional messaging models have their limitations and neither of these allow for communicating data "in-place" generally. With large buffers, the extra allocations and copies implicit in these models result into higher latency and increased application memory footprint. In this work, we propose a new zero copy messaging model in Charm++ to address these messaging latency and memory consumption issues caused by the existing messaging semantics. This will facilitate reuse of user buffers, and eliminate the need to make additional large allocations and copies while still taking advantage of the asynchronous nature of the Charm++ model.

III. API Design and Implementation

The Converse and LRTS zero copy API [2] added support for performing RDMA operations across different networking layers and unified it under LRTS and added zero copy functionality in Converse. In this work, we build on top of that to support different user-facing zero copy API in Charm++. As discussed in [2], all the metadata information associated with a buffer is encapsulated in a Converse level class called `CmiNcpyBuffer`. Since this object only provides completion handling for converse handlers, in our Charm++ API, we add another class `CkNcpyBuffer`, which derives from `CmiNcpyBuffer`. This class adds a data member, called a `CkCallback` object, which is used for completion handling to invoke a user entry method after the completion of the zero copy operation. To effectively use zero copy semantics in applications and higher level libraries, we have designed and and implemented several higher level user-facing API in Charm++. These user level APIs primarily work with the two buffer information objects that encapsulate the metadata as discussed above: `CmiNcpyBuffer` and `CkNcpyBuffer`. To perform an RDMA operation, since the metadata information of both the local and remote buffer is required, we rely on the existing messaging API in Charm++ to transfer the metadata information from the remote target chare (on a remote PE) to the local initiator chare (on the local PE). In a zero copy transfer, the initiator chare is invoking the Get or Put call and the target chare is the remote participant of that transfer.

A. Direct API

The zero copy Direct API allows users to explicitly invoke a standard set of methods on the `CkNcpyBuffer` objects to avoid both sender and receiver side copies for point-to-point messages. It is the Charm++ equivalent of the Converse level API [2], where Charm++ entry methods are used instead of Converse message handlers. To use the Direct API, the user creates a local `CkNcpyBuffer` object and sends it to the other participating chare in a remote entry method invocation as illustrated in Figure 1.

```
// Inside an entry method...
CkCallback srcCb(CkIndex_Ping1::sourceDone(),
    thisProxy[thisIndex]);
CkNcpyBuffer source(myBuffer, size * sizeof(int),
    srcCb);

// Invoke a remote method
// passing my CkNcpyBuffer object
arrProxy[1].recvNcpySrcObj(source);
```

Fig. 1. Direct API object creation and handover

```
void recvNcpySrcObj(CkNcpyBuffer source) {
  CkCallback destCb(CkIndex_Ping1::destDone(),
    thisProxy[thisIndex]);
  CkNcpyBuffer dest(myBuffer, size * sizeof(int),
    destCb);

  // Call get on local dest object
  // passing the received source object
  dest.get(source);
}
```

Fig. 2. Direct API performing Get operation

On receiving the remote `CkNcpyBuffer` object, the other participating chare creates a local `CkNcpyBuffer` object and calls the standard get method on it by passing the remote object to perform a zero copy read operation as shown in Figure 2. After the completion of the get operation, the callbacks specified in both the objects are invoked. Inside the source callback, sourceDone, the source buffer can be safely modified or freed. Similarly, inside the destination callback, destDone, the user is guaranteed that the data transfer into the destination buffer is complete and the user can begin operating on the newly received data. These callback functions are illustrated in Figure 3.

```
void sourceDone(CkDataMsg *msg) {
  delete myBuffer;  // free the buffer
}

void destDone(CkDataMsg *msg) {
  // received data, begin computing
  computeValues();
}
```

Fig. 3. Direct API source and destination callbacks

Using this API, after the preliminary handover of one of the CkNcpyBuffer objects to the other end, the user can exploit the persistent nature of iterative applications to perform zero copy operations using the same buffer information objects across iteration boundaries. The implementation of the Charm++ Direct API is an extension of the Converse zero copy API where the same functionality is used [2] and charm callbacks are supported instead of handler functions because of the use of CkNcpyBuffer objects.

B. Entry Method API

The zero copy Entry Method API extends the capability of the existing entry methods in Charm++ with slight modifications in order to send and receive buffers without copies. It supports both point-to-point and optimized broadcast operations and allows users to send and receive previously received copy based buffers as special "zero copy" buffers.

To send a buffer using the Entry Method API, the user is required to annotate the buffer parameter as nocopypost in the entry method declaration in the .ci charm interface file as shown in Figure 4.

```
// .ci declaration
entry void recvBuffer(int size, nocopypost int
    buffer[size]);
```

Fig. 4. Entry Method API: Method Declaration

On the sender side, the user needs to wrap the buffer and an optional callback object inside a CkSendBuffer wrapper and invoke the remote entry method as shown in Figure 5. Figure 5 illustrates a point-to-point invocation. A broadcast call can be made in a similar manner by using the entire chare array proxy arrProxy instead of a chare array element proxy like arrProxy[1].

On the receiver side, the user is required to have two overloaded definitions of the same entry method. The first definition, called the Post Entry Method uses the same argument list

```
// Create a callback
CkCallback srcCb(CkIndex_Ping1::sourceDone(),
    thisProxy[thisIndex]);

// Invoke the remote method
// passing myBuffer in CkSendBuffer
arrProxy[1].recvBuffer(size, CkSendBuffer(myBuffer,
    srcCb));
```

Fig. 5. Entry Method API: Remote Invocation

with an additional CkNcpyBufferPost * parameter. The Post Entry Method is invoked first, allowing the user to match the sender/source buffers with corresponding receiver/destination buffer using tags. This is done using the CkMatchBuffer call where the user supplies the CkNcpyBufferPost * pointer along with the index of the operation and a user provided integer tag. The index corresponds to the index of the nocopypost parameter among multiple nocopypost parameters, i.e. the first nocopypost parameter will have an index of 0, the next will be 1 and so on. This is illustrated in Figure 6.

```
// post entry method
void recvBuffer(int size, int *buffer,
    CkNcpyBufferPost *post) {
  // Match 0th source buffer with tag1
  CkMatchBuffer(ncpyPost, 0, tag1);
}
```

Fig. 6. Entry Method API: CkMatchBuffer call inside Post Entry Method

For every CkMatchBuffer call with a tag, there should be corresponding CkPostBuffer call with the same tag that is used to post the receiver/destination buffer. This call can be made from any entry method: before, after, or inside the Post Entry Method. This is equivalent to an MPI_IRecv call that is made when the receiver is ready to receive a buffer. This is illustrated in Figure 7.

```
// in some other entry method...
// ready to post buffer
CkPostBuffer(myBuffer, mySize, tag1);
```

Fig. 7. Entry Method API: CkPostBuffer call

After the execution of a CkPostBuffer (occurring from any entry method) and the corresponding CkMatchBuffer (executed inside the Post Entry Method), the runtime system performs the zero copy operation. On completion of all the zero copy operations of a particular entry method, the actual entry method is invoked. The actual entry method is the other overloaded definition of the same entry method, without the CkNcpyBufferPost * parameter. Inside the actual entry method, it is guaranteed that all the posted buffers have received the data from the send buffers. This is illustrated in Figure 8. Similar to the Direct API, the source/sender callback is invoked to signal that the buffer is ready to be reused or freed.

The implementation of the Entry Method API primarily relies on the tag matching functionality and the source-to-source code generation that uses the .ci file. On the sender side, the

```
// actual entry method
void recvBuffer(int size, int *buffer) {
  computeValues();  // data ready in buffer
}
```

Fig. 8. Entry Method API: Regular Entry Method

user invoked CkSendBuffer is converted to CkNcpyBuffer using a simple macro. The implementation uses the source-to-source code generator to generate the marshalling code on the sender side and the unmarshalling code on the receiver side. On the sender side, this generated marshalling code for the Entry Method API copies the smaller CkNcpyBuffer information object into the message, as opposed to copying the entire buffer (which is much larger in size) as done in the case of the regular messaging API. On the arrival of the message, the generated unmarshalling code first executes the Post Entry Method allowing the user to match the receiver buffer with a tag. The CkMatchBuffer call uses the tag to check if the receiver has already posted a buffer with the same tag. This is done by searching a hash table postedBuffMap that is used to store any posted receiver buffer information with the tag as the key when the user calls CkPostBuffer. If the receiver buffer has already been posted, a Get operation is issued by internally calling LrtsIssueRget. If the receiver buffer is not posted, the matching source buffer information is stored in another hash table matchedBuffMap with the tag as the key. On the user calling CkPostBuffer at a later point in time, matchedBuffMap is searched to find the source buffer information and a Get operation is issued in the same manner. Thus, using two hash tables allows us to flexibly tag match and pair any CkMatchBuffer and CkPostBuffer call. After the Get operation is completed, the source callback method is invoked and the received message is enqueued again in order to execute the actual entry method to signal to the receiver that the data transfer is complete. For implementing zero copy transfers in a broadcast call, we use a spanning tree, where each node of the spanning tree represents a Charm++ process. The spanning tree is rooted at the process that contains the source buffer and all the other nodes represent the recipient processes. Get calls are performed in a top-down order, where each parent node serves as the source process for Get calls made by its immediate children. The root node's source callback is invoked when the first level of child nodes have completed their Get calls. Similarly, the entry method on each non-root parent node is invoked when its immediate child nodes have completed their Get calls. The entire broadcast call is complete when all the leaf nodes of the spanning tree have received the source buffer.

C. Pup Buffer API for Migrations

Aside from inter-task communication, another major source of communication in task-based programs is that of moving the persistent data owned by migratable objects. In Charm++, chares can migrate during execution across nodes, usually for the purposes of dynamic load balancing or checkpointing for fault resilience. Charm++ provides a Pack-UnPack (pup) API that enables users to write a single simple routine per chare class that handles both sides of the migration process. For each migration, the size of the chare's data must be assessed, a message must be allocated to that size, the chare's state copied into the message, and then the message transferred and unpacked on the destination PE. As a consequence of the existing pup API, during migrations the memory usage of a Charm++ program can transiently spike if many chares are relocating simultaneously, as is common in greedy rebalancing algorithms. Consequently, we sought to use the zero copy infrastructure we have built for the purpose of efficient migrations. The current pup API proved to be limiting in terms of not separating the allocation and transfer of data from the completion of the transfer. Thus, we added a new pup API called "pup_buffer" which operates asynchronously on the unpacking side of the protocol. This allows users to mix regular pup and pup_buffer objects in the same chare, using only pup_buffer for large arrays of data.

To use this API, the user has to call pup_buffer on the PUP::er object inside the chare's pup method. The pup method is the standard method that is written with a PUP::er object as an argument. This method is called by the runtime system when the chare is about to be migrated (for sizing and packing) or when the chare has just been migrated (for unpacking). The pup_buffer method as shown in Figure 9 takes two arguments: buffer pointer and size. Optionally, the user could also pass a custom allocator and deallocator in this call.

```
// standard pup routine
void pup(PUP::er &p) {
  p | iteration; // pup using the copy-based scheme
  p.pup_buffer(buffer, size);
}
```

Fig. 9. Pup Buffer API

Our implementation uses a similar approach to the Entry Method API, where instead of packing the entire buffer, a CmiNcpyBuffer object is created out of it and packed instead. Similarly, on the receiver side, the source buffer's CmiNcpyBuffer object is unpacked and a Get is invoked into the newly allocated buffer. CmiNcpyBuffer is used over CkNcpyBuffer because there is no use of the additional CkCallback object added to CkNcpyBuffer. Unlike the copy based pup API, since the pup_buffer API executes asynchronously, there is no guarantee of the data transfer being complete during unpacking. To avoid entry methods of a chare with an active pup_buffer call executing on incomplete data, we buffer messages targeted to this chare in the runtime system until the issued Get completes. On the completion of the Get call, all buffered messages are released to execute the appropriate entry methods. The source buffer on the previous home PE (where the chare migrated from) is deallocated.

D. Read-only Data

To avoid the duplication of read-only data, Charm++ provides a special "read-only" data abstraction. Users declare such data in the .ci interface file and set its value in the main chare's constructor before creating chare arrays whose elements will use it. Read-only data can range in size and complexity from simple integers to custom singleton objects to STL containers of objects. The only requirement is that these objects have PUP methods so that they can be communicated. Internally, Charm++ performs a broadcast of the read-only data from PE 0 to all other nodes and stores a single copy of the read-only data per node. We replaced the implementation of large, contiguous memory read-only types with a zero copy broadcast. This minimizes memory usage and speeds up startup on all nodes for applications with large read-only data. There were no changes needed for the user-facing API since the control flow of creating read-only data members on all nodes was already hidden from users. This broadcast is implemented using the same spanning tree scheme as used in the broadcast call for the Entry Method API. The runtime system decides to use the zero copy scheme to broadcast a large contiguous read-only buffer if the buffer size exceeds a predefined networking layer dependent threshold size.

IV. Results

TABLE I
BENCHMARKING MACHINES AND THEIR CONFIGURATION

Machine	Cores/Node	Memory/Node	Network	Charm Build
iForge	40	192 GB	Infiniband	ucx
Stampede2	68	96 GB	Omni-Path	ofi
Cori	32	128 GB	Aries	gni
Quartz	36	128 GB	Omni-Path	ofi

We use four HPC machines for all our performance experiments. These details including the Charm build used is summarized in Table I. We use the non-SMP version in all our Charm++ and AMPI builds, which uses one CPU core as a single PE for one process.

A. *Charm++ Benchmarks*

To evaluate the performance of the zero copy API in Charm++, we have used two benchmarks that compare the performance of the regular messaging model with the zero copy messaging model.

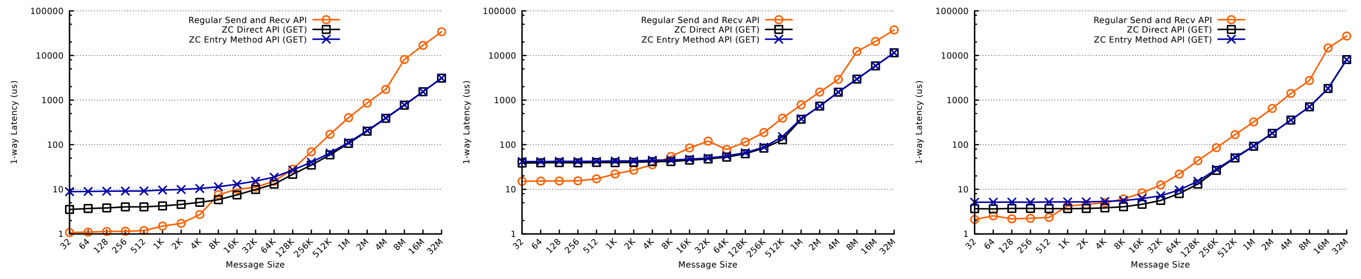
1) Point-to-Point Performance: We use a ping-pong benchmark for the evaluation of point-to-point messaging performance in Charm++. The benchmark exchanges messages for a fixed number of iterations (1000 iterations for up to 256 KB and 100 iterations for 512 KB to 32 MB) and measures the one-way messaging latency to send and receive data from user buffers of two chares on two different PEs. The one-way latency is determined by averaging out the total time across all iterations and dividing that value by 2. This entire process is repeated for different message sizes. Using this benchmark, since we aim to determine the time taken for send and receive directly from user buffers, in the Regular API we make an explicit copy from the received message into the user buffer. On the other hand, this is unnecessary for the zero copy API because this direct transfer happens implicitly.

Improvements in intra-node and inter-node latency with zero copy Direct API and zero copy Entry Method API on three different machines are illustrated in figures 10 and 11. As seen in all the latency plots, for small messages, the regular messaging API performs better than the zero copy API because of the extra memory allocations and copies being inexpensive in comparison to the time taken to send the metadata message for the zero copy API. However, we see that the zero copy API begins to outperform the regular API for medium and large messages, with the improvement increasing with message size. This is because of the metadata message latency remaining constant, whereas the cost of additional allocations and copies increases proportionally with message size.

The small performance difference seen between the Direct API and Entry Method API in all p2p latency plots can be attributed to two additional overheads in the entry method API. First, memory registration and deregistration is performed for every iteration. Second, there is some additional processing which includes tag matching and packing/unpacking. These overheads are not incurred in the Direct API because it only requires memory registration and deregistration once, and separately, there is no requirement for tag matching or packing/unpacking. Cross Memory Attach (CMA) is supported on both Stampede2 and Cori. The zero copy API executions on these machines take advantage of this for intra-node transfers and this results into a smaller performance difference between Direct API and Entry Method API as seen in 10b and 10c. For these transfers, since registrations are not required, the Entry Method API only incurs the overhead associated with the additional processing. The range of speedups, percentage improvements, and threshold message sizes over which the the two variants of point-to-point zero copy API outperform the regular API are summarized in Table II.

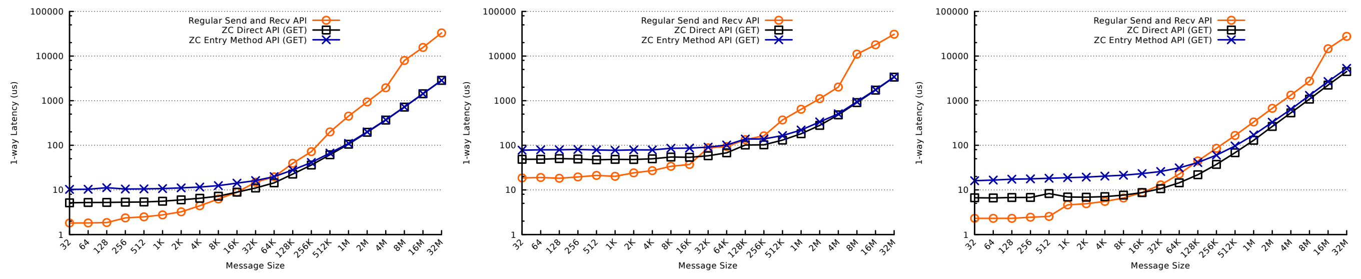
2) Broadcast Performance: To evaluate the performance of broadcast operations, we use a ping-all and reduce benchmark written in Charm++. The benchmark measures the latency for a broadcast and reduction across all PEs for different message sizes. The average time for a single broadcast and reduction operation is determined by averaging the total time across many iterations (100 iterations up to 256 KB and 10 iterations for 512 KB to 32 MB). Similar to the ping-pong benchmark, since we aim to determine the time taken for send and receive directly form user buffers, in the Regular API we make an explicit copy from the received message into the user buffer.

Figure 12 illustrates the weak scaling performance of the broadcast version of the zero copy Entry Method API over the regular API. The figure plots broadcast and reduction latency for four different message sizes on three different machines. As seen in the plots, the improvement achieved with the zero copy API increases for the larger message sizes. This can be explained with the same analysis conducted for the point-to-point ping-pong experiments i.e. as the message size increases, the cost of the extra allocation and copy increases, making the regular API perform poorly for large messages. Additionally, it is also seen that in most cases, the improvement increases proportionally to the number of



(a) iForge (b) Stampede2 (c) Cori

Fig. 10. Comparison of intra-node latency between regular and zero copy messaging API



(a) iForge (b) Stampede2 (c) Cori

Fig. 11. Comparison of inter-node latency between regular and zero copy messaging API

TABLE II
IMPROVEMENT IN POINT-TO-POINT LATENCY WITH ZERO COPY MESSAGING API.

Improvement	**Metric**		**Intra-node**			**Inter-node**	
		iForge	**Stampede2**	**Cori**	**iForge**	**Stampede2**	**Cori**
ZC Direct API	SpeedUp	1.2x – 10.9x	1.3x – 4.2x	1.15x – 8x	1.3x – 11.5x	1.5x – 9.1x	1.2x – 6.5x
	% Improvement	22% – 90%	23% – 69%	13% – 70%	25% – 9%	33% – 89%	18% – 83%
	Threshold Size	8K	8K	1K	32K	32K	16K
ZC Entry Method API	SpeedUp	1.1x – 10.9x	1.2x – 4.2x	1.1x – 8x	1.4x – 11.5x	1.2x – 9x	1.1x – 5.5x
	% Improvement	5% – 90%	19% – 69%	9% – 70%	28% – 91%	18% – 90%	8% – 81%
	Threshold Size	128K	8K	8K	128K	256K	128K

TABLE III
IMPROVEMENT IN BCAST LATENCY WITH ZERO COPY MESSAGING API.

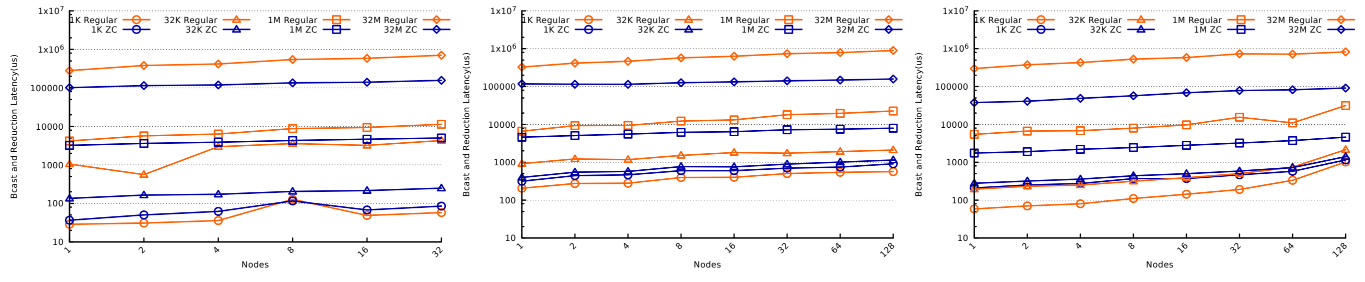
Metric	**iForge**	**Stampede2**	**Cori**
SpeedUp	1.3x – 17x	1.8x – 5.6x	2.9x – 9.2x
% Improvement	23% – 94%	30% – 82%	67% – 89%

nodes or PEs. This indicates that zero copy entry method API scales better than the regular API. On all machines, the regular API performs better for the 1K size but the zero copy API performs better for larger message sizes in most cases as seen in Figure 12. Unlike iForge and Stampede2, it can be observed that the regular API performs better than the zero copy API on Cori for a 32K message for up to 32 nodes. We believe that this is primarily due to the relatively expensive memory registration and deregistration operations on GNI, that are performed for every iteration of the zero copy API. These operations outweigh the benefits of the zero copy API at 32K message size. A similar pattern can be seen in the point-to-point case in Figure 11c. The range of speedups and percentage improvements achieved by the zero copy API over the regular API are summarized in Table III.

B. Adaptive MPI

Adaptive MPI (AMPI) is an MPI implementation developed on top of Charm++ [3]. It provides the dynamic runtime capabilities of Charm++ behind the familiar API of the MPI standard, meant as a drop-in replacement for other MPI libraries. It does this by virtualizing MPI ranks, usually assumed to be identical to Operating System processes, as User-Level Threads (ULTs). Each rank runs until it reaches a blocking communication call, at which point it yields to the scheduler who is free to schedule another virtual rank that has work to do. The Charm++ runtime system schedules AMPI ranks cooperatively as it does tasks or entry method invocations.

In its implementation, AMPI consists of a 1D chare array of all the virtual ranks in `MPI_COMM_WORLD`. Communication was previously handled via explicit Charm++ message passing (using custom messages, not parameter marshalled entry meth-



(a) iForge (b) Stampede2 (c) Cori

Fig. 12. Comparison of Broadcast and Reduction Latency between regular and zero copy messaging API

ods). This meant that, regardless of the locality of the receiver, AMPI would first serialize the message buffer into a Charm++ message, then hand that message off to Charm++ which would route it to the receiving rank's chare array element. The chare array element would take ownership of the message and when it was matched with a request, would deserialize the message into the receive buffer. In [4] AMPI was optimized for within-process transfers using direct userspace memory copies in a rendezvous protocol.

1) AMPI Communication Optimizations: In this paper we extend that previous work by providing AMPI with general zero copy or in-place communication capabilities. Internally AMPI now uses the zero copy Direct API, though it could be rewritten to use the Entry Method API just as well. We support migrations of virtual ranks during execution, so long as they do not have any pending messages. Normally load balancing is conducted at a synchronization point at the end of timestep, so this is not a problem, though handling migrations leads to complications in the rendezvous protocol.

AMPI now chooses its communication protocol based on both the message size and the expected locality of the receiver from the sender. We say expected because Charm++ uses a distributed location management protocol that does not generally guarantee knowledge of all object's places at a given time on any given PE. It does guarantee eventual delivery of messages, but the receiver may not be where the sender initially thinks it is if it has recently migrated. Consequently, we must handle the case where we expect a receiver is on our same node but has actually migrated away. In this case we choose not to pin the memory upfront for same-node transfers in order to avoid the memory registration cost, and fall back to a slower put-based protocol where the receiver pins its memory and sends back a `CkNcpyBuffer` object to the sender, who then uses it to perform a `put` of the data after pinning its own buffer. Otherwise, if the receiver is where the sender expected, the receiver does a `get` from the sender's buffer to its own. Because Charm++'s distributed location management is generally only ever out of date the first iteration after load balancing, and load balancing is usually infrequent, the slower protocol is rarely used in practice but is necessary for correctness.

Figure 13 shows the results of the OSU MPI point-to-point latency benchmark for original AMPI compared to the new one

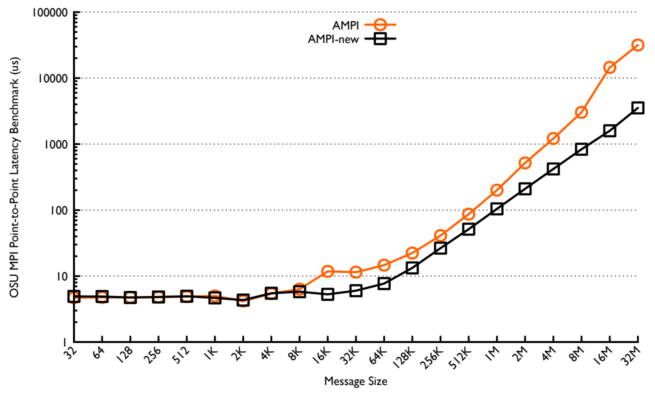


Fig. 13. OSU MPI Point-to-point latency benchmark on Quartz at LLNL.

with the zero copy API. The benefit of the zero copy API is seen when we switch from an eager protocol using Charm++ custom messages to a rendezvous one with the Direct API, avoiding unnecessary memory copies.

2) AMPI Memory Optimizations: We also modified AMPI to use the zero copy pup_buffer API for migrations. AMPI's memory allocator, Isomalloc, ensures that all stack and heap data are migratable by allocating each virtual rank's data from within unique slices of the global virtual memory address space. This ensures that after migrating a virtual rank's data, all pointers remain valid because all memory remains allocated at the same virtual address.

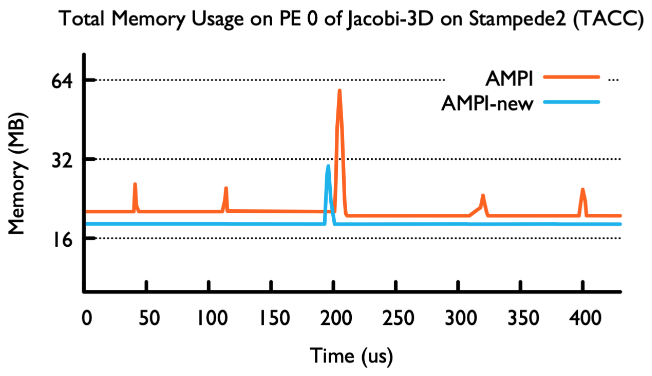


Fig. 14. Memory usage on PE 0 of a three dimensional stencil benchmark on the Skylake partition of Stampede2 (TACC).

Figure 14 shows the the memory usage over time on PE 0 of a three dimensional Jacobi solver run on AMPI with 8x overdecomposition on 8 Skylake nodes of TACC's Stampede2 machine. The plot shows four timesteps of execution, with dynamic load balancing happening between the second and third iterations. AMPI-new includes three notable memory optimizations. First, the static memory footprint– the floor of memory usage– was reduced with optimizations done to AMPI's internal storage, hoisting read-only, per-rank storage to the node-level. Second, the smaller per-timestep peaks seen in the original AMPI result were eliminated by use of in-place communication. The zero copy Direct API is used for all point-to-point messages in this application, and the small metadata messages and per-message matching queue entries inside AMPI are all pooled. Third, the largest spiked in memory usage–that due to virtual rank migrations resulting from dynamic load balancing– is decreased by use of the pup_buffer API. This reduced the peak memory usage almost in half for the same number of virtual rank migrations.

The memory optimizations, taken together, allow users to run applications with larger memory sizes without running out of memory during rebalancing. Also, when migrating within shared memory we avoid any copies by passing ownership over the buffers. Copy avoidance also means faster messaging for large buffers. The overall result is a 7% faster run time of Jacobi-3D with nearly 2x lower peak memory usage.

V. Related Work

RDMA has been incorporated into numerous programming models for HPC over the past two decades plus. MPI, with its library-based implementation, has always favored communication of user-owned memory buffers rather than first-class message structures, allowing implementors to hide eager and rendezvous protocols inside the runtime [5] [6]. FG-MPI [7] and HMPI [8] have explored true zero copy message passing semantics in MPI. PGAS models similarly hide protocol choice behind the language's abstractions, so that RDMA usage becomes an implementation issue [9] [10]. HPX [11] and Legion [12] similarly hide communication from users behind higher-level data and synchronization abstractions. GasNet [13], a communication layer for several task-based programming models, has incorporated RDMA and remote atomic operations into its design.

VI. Conclusion

Task-based programming models are gaining importance because of both increasingly complex software applications and hardware architectures. Their ability to overlap communication and computation, and balance load is key to extracting good performance and scaling on HPC machines. However, some of their simplicity in messaging semantics leads to degraded messaging latency and increased memory footprint. In this paper, we extended our previous work on enabling zero copy support in Converse and LRTS, and defined and implemented new messaging semantics in Charm++ in order to accelerate large message transfers in multiple contexts. For point-to-point messages, broadcasts, read-only data handling, and data migration, we identified the sources of data copies in Charm++'s existing APIs and designed new alternatives to avoid the copies wherever possible. This work built on top of our previous work in the lower level communication runtime [2].

For future work, we would like to extend the zero copy semantics to other collective operations in Charm++ and AMPI, like reduction, gather, scatter, and all-to-all. We plan to incorporate the zero copy entry method API inside AMPI to implement broadcast operations. We also plan to extend the zero copy support in the networking layers to incorporate and improve our lower level implementations. As a part of this effort, we plan to improve our MPI layer implementation by using the improved one-sided RMA operations. We also aim to support RDMA over Converged Ethernet (RoCE).

References

[1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[2] N. Bhat, S. White, and L. Kale, "Enabling support for zero copy semantics in an Asynchronous Task-based Programming Model," ser. AMTE, Euro-Par, 2021.

[3] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.

[4] S. White and L. V. Kale, "Optimizing point-to-point communication between adaptive mpi endpoints in shared memory," *Concurrency and Computation: Practice and Experience*, pp. n/a–n/a. [Online]. Available: http://dx.doi.org/10.1002/cpe.4467

[5] M. P. I. Forum, *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015. [Online]. Available: https://books.google.com/books?id=Fbv7jwEACAAJ

[6] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based MPI implementation over infiniband," *Int'l Journal of Parallel Programming*, 2004.

[7] H. Kamal and A. Wagner, "FG-MPI: Fine-grain MPI for multicore and clusters," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010*, April 2010, pp. 1–8.

[8] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma, "Ownership passing: Efficient distributed memory programming on multi-core systems," *SIGPLAN Not.*, vol. 48, no. 8, p. 177–186, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2517327.2442534

[9] T. S. Tarek El-Ghazawi, William Carlson and K. Yelick, *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[10] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286120.1286123

[11] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401.

[12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 66.

[13] D. Bonachea and P. H. Hargrove, "Gasnet-ex: A high-performance, portable communication library for exascale," in *Languages and Compilers for Parallel Computing*, M. Hall and H. Sundar, Eds. Cham: Springer International Publishing, 2019, pp. 138–158.