Enabling support for zero copy semantics in an Asynchronous Task-based Programming Model

Nitin Bhat[1], Sam White[2] and Laxmikant V. Kale[1,2]

Charmworks, Inc., Urbana, Illinois, USA nitin@hpccharm.com
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
{white67,kale}@illinois.edu

Abstract. Communication is critical to the scalable and efficient performance of scientific simulations on extreme scale computing systems. Part of the promise of task-based programming models is that they can naturally overlap communication with computation and exploit locality between tasks. Copy-based semantics using eager communication protocols easily enable such asynchrony by alleviating the responsibility of buffer management from the user, both on the sender and the receiver. However, these semantics increase memory allocations and copies and in turn affect application memory footprint and performance, especially with large message buffers.

In this work we describe how the so-called "zero copy" messaging semantics can be supported in Converse, the message-driven parallel programming framework that is used by Charm++, by implementing support for user-owned buffer transfers in its lower level runtime system, LRTS. These semantics work on user-provided buffers and do not semantically require copies by either the user or the runtime system. We motivate our work by reviewing the existing messaging model in Converse/Charm++, identify its semantic shortcomings, and define new LRTS and Converse APIs to support zero copy communication based on RDMA capabilities. We demonstrate the utility of our new communication interfaces with benchmarks written in Converse. The result is up to 91% of message latency improvement and improved memory usage. These advances will enable future work on user-facing APIs in Charm++.

Keywords: Charm++ · Converse · RDMA · Parallel Programming · Asynchronous Tasking · Communication Optimizations

1 Introduction

With the advent of Exascale computing, the importance of efficient data movement is expected to increase greatly. In fact, the underlying technological factors that led to dramatic increase in within-node computational capacity, without a proportionate increase in communication capabilities entail that even on small clusters, communication issues present significant challenges. RDMA,

which stands for Remote Direct Memory Access, is a network capability that allows a machine to read from or write to a remote machine's memory without the involvement of the Operating System or CPUs. One sided communication with the help of RDMA supported hardware is the natural choice for large messages as it has proven to reduce latencies and increase bandwidth for large payloads in High Performance Computing (HPC) networks. RDMA also benefits from so-called "zero copy" semantics, where the data being transferred is not copied between the layers of the network stack (zero copy means no intermediate copies). The bypassing of CPU along with the elimination of copies ensure lower latencies and higher throughput for RDMA enabled networks over regular networks.

Since memory-bound operations are much slower in comparison to the CPU, it has been observed that memory intensive operations act as the primary bottleneck in numerous applications and thus reduce application performance and increase energy consumption. For this reason, reducing memory pressure by saving the cost of allocations and copies helps in improving application performance significantly.

2 Background, Motivation and Contributions

Converse [1] is a complete but low level message-driven (i.e. task based) parallel programming system. It supports a scheduler that handles user-level threads as well as stackless tasks. The latter may be created locally or from remote processors, which is similar to active messages. Each such task has a handler reference and a data payload, and possibly other metadata such as priorities. In its current usage, on the source PE (processing element, typically used to denote a CPU core), the data payload has to be copied from the user data structure to a contiguous message that includes the message metadata. Similarly, on the destination PE, the payload has to be copied from the received message into the user data structure. It is this copying on both the source and destination that we wish to optimize.

Converse is designed to be used as a substrate for implementing parallel languages, but Charm++ is the most well-known system that uses it. Charm++ is an asynchronous parallel programming model and runtime system based on the idea of overdecomposition and migratable objects [2]. A Charm++ program is expressed in the form of interacting migratable C++ objects called *chares*, which interact via asynchronous method invocations. In such a method invocation, the passed parameters are copied ("marshaled") into a *Converse* message along with required metadata to encode information such as recipient object and handler references. This copying ensures that the user passed parameters are safe to be overwritten immediately on the source chare after the entry method invocation. On the destination chare, such copying from the received Converse message into the user data structures is again required to use the received data beyond the scope of the entry method since the runtime system frees the message for safe memory management. Thus the optimization we aim at is useful (necessary but not sufficient) to optimize Charm++ and its myriad applications.

Now consider a situation in which a user of Converse (either end programmer or the Charm++ runtime) needs to send multiple large data arrays, along with other scalar data. All the large arrays must be copied into a Converse message on the sending processor and on the receiving processor they typically have to be copied into application data structures. With large buffers, these copies come at the price of increased memory footprint, higher latency, and lower bandwidth. In this work, we aim to address the limitations of the current messaging semantics in Converse and propose a new zero copy messaging model that will allow communicating data "in-place". This will allow the user to avoid additional allocations and copies, and facilitate reuse of user buffers while still benefiting as much as possible from the asynchrony that underlies Converse (and Charm++) execution model.

3 Design and Implementation

The Charm++ software stack consists of three primary software layers: Charm++, Converse, and LRTS, with support for various networking layers beneath LRTS. Charm++ is the high level layer interfacing with the user code to support processor virtualization through the idea of coarse grained task and data objects called chares. Converse is a portability layer beneath Charm++ that supports message handling and uses a scheduler to enqueue received messages and invoke message handlers by using an appropriate dequeueing strategy. The networking layer which is below Converse is called the Low Level Runtime System (LRTS). The LRTS represents a set of APIs used by Converse to perform networking operations like sending and receiving messages. Each networking machine layer implements this set of APIs using provider-specific implementations, hiding their implementation details from the upper layers of the Charm++ software stack.

3.1 LRTS API

The basic functionality for performing a zero copy transfer of a buffer is dependent on the underlying network and its capabilities. HPC specific networks like UCX, OFI, GNI, and Verbs provide native support for RDMA operations, whereas network libraries like TCP and UDP used primarily over ethernet, do not natively support RDMA operations. Since each networking layer has its own implementation for supporting zero copy transfers, we define a unified LRTS API for zero copy transfers and implement the API for each networking layer. We provide the following LRTS methods for implementing zero copy functionality:

- void LrtsSetRdmaBufferInfo(void *info, const void *ptr, int size, int mode)
- void LrtsDeregisterMem(const void *ptr, void *info, int pe, int mode)
- void LrtsIssueRget(NcpyOperationInfo *ncpyOpInfo)
- void LrtsIssueRput(NcpyOperationInfo *ncpyOpInfo)

`LrtsSetRdmaBufferInfo` is used to set the network specific metadata information for a buffer or a memory region that is intended to be used for an RDMA operation. For most RDMA supported layers, this involves registration of the memory region and storing that information in the `info` object. `LrtsDeregisterMem` is used to deregister an already registered region of memory. `LrtsIssueRget` is used to perform an RDMA Get or Read operation from a remote buffer. Similarly, `LrtsIssueRput` is used to perform an RDMA Put or Write operation to a remote buffer. Since buffer information pertaining to both the local and the remote buffer is required to perform a Get or Put operation, the wrapper object `NcpyOperationInfo` is used to store the metadata information of both the buffers, including completion handling information which is used to call the registered higher level completion function on completion of a Get or Put. These low-level LRTS APIs described above will provide the infrastructure for higher-level abstractions in Converse and Charm++. In this section, we briefly describe the implementation of the LRTS APIs for different networking layers and for the special case of transfers within a physical node.

Networking Layers Native networking layers provide explicit control to the user to design and tune the usage of the network library's API as intended. Such layers also typically require the user to explicitly manage pinned or registered memory. In our work, we have chosen to implement the basic functionality for performing zero copy operations on four popular native HPC networking layers that require explicit pinned memory management. These include Unified Communication X (UCX) [3], OpenFabrics Interfaces (OFI) [4], uGNI or GNI, and Verbs. For these networking layers, in our implementation, inside `LrtsSetRdmaBufferInfo`, we use the network library provided method to register the buffer and store the memory handle (or memory region) along with any additional information (like rkey) in the info object. Similarly, in `LrtsDeregisterMem`, we use the method to deregister the buffer using the memory handle available in the info object. Since all these networking layers natively supports RDMA operations, we directly use the Get and Put methods provided by each of the network libraries to perform RDMA Get and Put operations inside `LrtsIssueRget` and `LrtsIssueRput` respectively. Completion handling is performed by polling a completion queue and calling an appropriate higher level completion function using a heap object in the case of OFI, GNI, and Verbs. UCX supports a `ucp_send_callback_t` argument provided in the Get and Put calls, which can be set to a specific function, which is invoked on completion. Inside the ucp callback function, the common higher level completion function is invoked. In addition to native networking layers, Charm++ also provides an MPI networking layer to be used for interoperation with MPI or on new machines without reliable support yet for a native layer. Since MPI internally manages pinned memory, our implementation is simplified and simply uses matching `MPI_Isend` and `MPI_Irecv` calls to perform "zero copy" reads and writes. For networking layers that do not natively support RDMA, like TCP and UDP, we have also provided a copy based implementation in order to maintain API consistency.

Intra-Node Communication between endpoints that are on the same physical node is common on many-core nodes. We use Cross Memory Attach (CMA) [5] for performing reads and writes between processes within the same host. CMA is a mechanism that was introduced in Linux kernel version 3.2 to improve communication performance between processes of the same physical node. A `process_vm_readv` call is used in `LrtsIssueRget` and a `process_vm_writev` call is used in `LrtsIssueRput` to perform CMA read and write operations. These calls are synchronous and complete inline, allowing us to perform completion handling immediately upon returning from the CMA call. For transfers within the same process, we further optimize the communication with a user-space memcpy operation for optimal performance.

3.2 Converse API

The API for zero copy semantics in Converse is built on top of the basic functionality of the lower level, which is unified by the LRTS API. Since the metadata information of both the local and remote buffer is required to perform an RDMA operation, we support a 2-phase protocol: rely on the existing messaging API in Converse to transfer the metadata information, followed by the one-sided (say, `get`) API to execute the large data transfer. The metadata associated with a buffer includes information like the pointer, size, home PE, memory registration information (which is required for most RDMA networks), and any other data fields used for notifying the user on completion of the zero copy transfer. This metadata information in encapsulated into a class we provide called `CmiNcpyBuffer`. This class contains methods such as `get` and `put` to perform RDMA Get and Put operations respectively. Additionally, methods called `registerMem` and `deregisterMem` perform memory registration and deregistration. `registerMem` is called from the constructor of `CmiNcpyBuffer` to perform memory registration of the buffer during declaration of this object. The user is responsible for invoking `deregisterMem` after the completion of the RDMA transfer.

The above public methods of `CmiNcpyBuffer` constitute the zero copy API in Converse. The user is required to first construct and send the metadata object

```
// Inside a converse method.. Declare a CmiNcpyBuffer object
CmiNcpyBuffer srcMeta(myBuffer, buffSize, srcDoneHandler);

// Invoke a remote method passing my CmiNcpyBuffer object
buffObjMsg *msg = (buffObjMsg *)CmiAlloc(sizeof(buffObjMsg));
CmiSetHandler(msg, destMetadataHandler);
msg->buffObj = srcMeta;
CmiSyncSendAndFree(remotePe, sizeof(buffObjMsg), msg);
```

Fig. 1. CmiNcpyBuffer object creation and handover

CmiNcpyBuffer of one PE to the other participating PE, using the existing messaging API in Converse. This is illustrated in Figure 1 where destMetadataHandler is the target handler for the message. This handler on the destination constructs a local CmiNcpyBuffer and calls the get method as shown in Figure 2.

```
void destMetadataHandler(buffObjMsg *msg) {
  CmiNcpyBuffer *srcMeta = msg->buffObj;
  CmiNcpyBuffer destMeta(myBuffer, buffSize, destDoneHandler);
  destMeta.get(*srcMeta);
}
```

Fig. 2. Remote PE performing Get operation

On completion of zero copy transfers in Converse, the runtime system invokes the handlers passed by the user in the CmiNcpyBuffer object constructors. When the source handler srcDoneHandler is called, the buffer can be safely modified or freed as shown in Figure 3. Similarly, inside destDoneHandler, the user is guaranteed that the data transfer into the destination buffer is complete and the user can begin operating on the newly available data as shown in Figure 3. The runtime invocation of these handler functions on completion enables the user to be asynchronously notified about reuse of source buffer and arrival of data in the destination buffer. This is essential to integrate our protocol in a message-driven execution model and makes it more efficient in comparison to the MPI model, which requires the user to make a blocking MPI_Wait call or repeatedly call MPI_Test to determine completion. This scheme also allows one to wait modularly for multiple data transfers, even across library boundaries.

```
void srcDoneHandler(char *msg) {
  free(myBuffer);  // free the buffer
}
void destDoneHandler(char *msg) {
  // received data, begin computing
  computeValues();
}
```

Fig. 3. Source and Destination Handler function

The implementation of the Direct API is relatively straightforward. The user is responsible for explicitly sending over the remote metadata information using a CmiNcpyBuffer object as seen in Figure 1. When the get method is called on CmiNcpyBuffer by passing the source object, the Converse implementation creates a NcpyOperationInfo object from the two CmiNcpyBuffer objects and

simply makes a call to `LrtsIssueRget`. The converse completion function registered with LRTS initiates the invocation of both the source and destination handler functions passed by the user. PUT based one sided operations are supported in a similar manner.

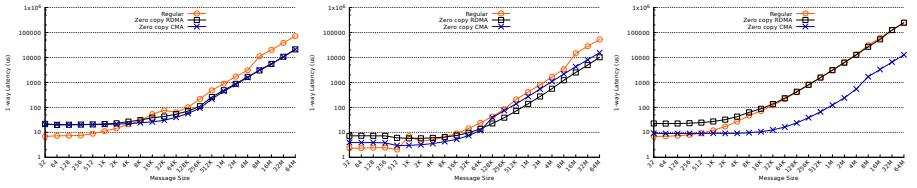
4 Results

Table 1. Benchmarking machines and their configuration

Machine	Cores/Node	Memory/Node	Network	Charm Build
iForge	40	192 GB	Infiniband	ucx, verbs
Stampede2	68	96 GB	Omni-Path	ofi
Cori	32	128 GB	Aries	gni, mpi
Linux Workstation	4	8GB GB	Ethernet	netlrts(udp)

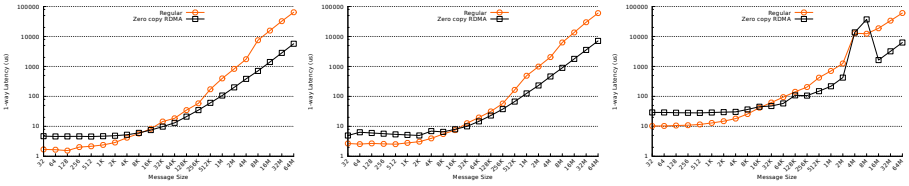
We conducted our performance experiments on three HPC machines and one general purpose linux machine as summarized in Table 1. All our converse builds are configured to use the non-SMP version, which uses one CPU core as a single PE for one process. For benchmarking, we use 2 PEs on 1 node for intra-node messaging and 1 PE each on 2 nodes for inter-node messaging. In our experiments, we use a point-to-point ping-pong benchmark written in Converse. This measures the one-way messaging latency for different message sizes between two processes that exchange messages using their user buffers for a fixed number of iterations. Since our ping-pong benchmark requires exchange of data directly from the user buffers, in the Regular API we explicitly make a copy from the received message into the user owned buffer. This is not required in the zero copy API because the data transfer always happens directly from the sender owned buffer to the receiver owned buffer. Across iterations, since the same buffers are used for a particular message size, it is only required to register the buffer at the beginning of all the iterations corresponding to that size. This allows the zero copy API to perform the `get` operation persistently using the same buffer information objects.

Figures 4 and 5 illustrate the improvements in intra-node and inter-node latency with zero copy API on four different machines. As seen in all the latency plots, the regular messaging API performs better than the zero copy API for smaller messages. This is because of the time taken for the extra memory allocation and copy performed in the regular API being small in comparison to the additional latency incurred in sending the metadata information message for the zero copy API. Starting from medium to large messages, we see that the zero copy API begins to perform better than the regular API and the improvement increases with message size. This can be again explained by the cost of performing the additional allocation and copy, which begins to proportionally increase

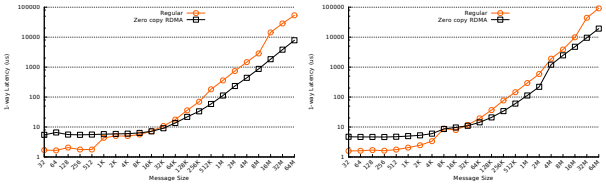


(a) ofi on Stampede2 (b) gni on Cori (c) netlrts on a Workstation

Fig. 4. Comparison of intra-node latency between regular and zero copy API



(a) ucx on iForge (b) verbs on iForge (c) ofi on Stampede2



(d) gni on Cori (e) mpi on Cori

Fig. 5. Comparison of inter-node latency between regular and zero copy API

with message size, whereas the metadata message latency remains constant. The range of performance improvements in latency and the threshold message sizes above which the zero copy API begins to perform better than the regular API is summarized in Table 2 for intra-node transfers and Table 3 for inter-node transfers.

Cross Memory Attach (CMA) is supported on Stampede2, Cori and the commodity linux workstation. Figure 4 highlights the performance between the regular API and the zero copy API with both CMA and RDMA on intra-host transfers between 2 PEs. On Stampede2, CMA performs better than RDMA (using ofi) for most message sizes, esp in the medium message size range because of the expensive network operations in comparison to using shared memory. However, on Cori, CMA outperforms RDMA (using gni) only upto a threshold size. Beyond this, the advantage of bypassing the CPU as done in the case of RDMA outweighs the benefit of using shared memory, which still requires kernel intervention. On the linux workstation as seen in 4c, because of no support for RDMA, we use the copy based implementation underneath to maintain API consistency. The additional overhead of sending the metadata message incurred in the zero copy API, leads to the poorer performance as compared to the regu-

Table 2. Improvement in intra-node latency with zero copy messaging API.

Metric	Stampede2		Cori		Workstation
	CMA	ofi	CMA	gni	CMA
SpeedUp	1.2x – 3.72x	1.4x – 3.7x	1.13x – 3.5x	1.2x – 5.8x	1.3x – 26x
Threshold Size	8K	16K	1K	16K	512

Table 3. Improvement in inter-node latency with zero copy messaging API.

Metric	iForge		Stampede2	Cori	
	ucx	verbs	ofi	gni	mpi
SpeedUp	1.1x – 11.4x	1.2x – 8.4x	1.3x – 11.5x	1.2x – 7.8x	1.07x – 4.7x
Threshold Size	16K	32K	32K	16K	32K

lar API for smaller message sizes. As the message size increases, this additional overhead becomes minuscule in comparison to the cost of allocations and copying, leading to similar performance between regular and the copy-based zero copy API, and much better performance for CMA based zero copy API.

5 Related Work

RDMA has been well studied and applied to numerous parallel programming models over time. MPI's library model meant it has always operated on user-owned memory rather than explicit message objects. This has allowed library implementors to hide eager and rendezvous protocols behind two-sided send/recv operations [6]. PGAS models, such as UPC [7] and Chapel [8], aim to hide communication from users, so incorporating RDMA into those models has mostly been done in the lower levels of the runtime system and not in the user-facing API. GasNet serves as a lower level communication substrate for various task-based programming systems and has had RDMA incorporated into its design. Legion [9], which builds on top of GasNet, strives to hide communication from users and manage data movement automatically based on task dependencies. HPX [10] is another example of a tasking model that hides communication behind higher-level abstractions such as futures and executors.

6 Conclusion

With the growing complexity of exascale software applications and hardware architectures, task-based programming models appear promising. Asynchrony and the ability to migrate tasks and data around the system to balance computational load will be important for overall performance and scalability. In this work, we identified the shortcomings with the current messaging API in Charm++ for sending and receiving large buffers. We also added support for zero copy messaging in Converse and LRTS to enable the development of zero copy user APIs in Charm++.

Future work includes implementing Charm++ user-facing APIs on top of this work and improving the performance of our new APIs. We plan to implement two key optimizations. First, by adding a registration cache that will intelligently handle memory registrations and deregistrations. Second, by developing a generic memory pool for allocating all the small sized heap objects that we use in our implementation. We believe these optimizations will allow us to extract better performance from our new APIs.

References

1. L. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An Interoperable Framework for Parallel Programming," in *International Parallel Processing Symposium 1996*, 1996.
2. 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.
3. P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "Ucx: An open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.
4. P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 34–39.
5. J. Vienne, "Benefits of cross memory attach for mpi libraries on hpc clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, ser. XSEDE '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2616498.2616532
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. T. S. Tarek El-Ghazawi, William Carlson and K. Yelick, *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.
8. 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
9. 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.
10. 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.