ACCELERATING MESSAGES BY AVOIDING COPIES USING RDMA IN AN ASYNCHRONOUS PARALLEL RUNTIME SYSTEM

BY

NITIN KUNDAPUR BHAT

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor L V Kale

ABSTRACT

With the advent of Exascale computing, the number and size of messages is expected to increase greatly. One sided communication with the help of Remote Direct Memory Access (RDMA) supported hardware is the natural choice for large messages as it has proven to provide reduced latencies and increased bandwidth for large payloads in High Performance Computing (HPC) networks. Using RDMA technology enables the network to bypass the Operating System and perform data transfers without the involvement of the Central Processing Unit (CPU). In addition to not consuming CPU cycles, using RDMA also benefits from zero copy networking where the data being transferred is not copied between the layers of the network stack.

Since memory performance is significantly lesser than the CPU performance, it has been observed that memory intensive operations reduce application performance and increase energy consumption. For this reason, reducing memory pressure by saving the cost of allocation and copy helps in improving application performance significantly.

The asynchronous message sending paradigm in Charm++ makes a copy of the payload at the sender side. It also requires copying the data from the message into the user's data structure at the receiver side. As the payload gets larger, the cost of these allocations and copies also increase proportionally. In this thesis, we show the benefits of avoiding the copies at both the sender and receiver side using RDMA on different applications. We also discuss the design of the zero copy user level Application Programming Interface (API) in Charm++ along with the underlying RDMA implementations for different networks in today's supercomputers.

To Manasa, for all her love and support

ACKNOWLEDGMENTS

I am forever indebted to Prof. Kale for being a very supportive guide, mentor and employer. He has always pushed me to persevere and strive for perfection. I thank all fellow students and co-workers at the Parallel Programming Laboratory and Charmworks, Inc, especially those whom I closely worked with for this project. I have learnt a great deal from Vipul about computer science and programming and his contribution in brainstorming and developmental efforts for this project was instrumental in shaping it. Jaemin and Justin worked on the Verbs and GNI layers respectively for the Entry Method API. I am grateful to Sam and Phil for their quick Gerrit reviews and suggestions during the development process. This project would not have been successful if not for them. I also thank all my fellow graduate students and Alumni at the University of Illinois for being great friends and constantly helping me learn and grow.

My parents have played a big role in shaping my personality and character. My brother, Pavan, is one of my best friends and he has always been by my side during all moments of life. I thank him and my family back in India for making me the person I am today.

Finally, I would like to thank a very special person, my fiance, Manasa Rao, who has changed my outlook towards life. I dedicate this thesis to you and I look forward to our partnership in life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMPI Adaptive MPI.

API Application Programming Interface.

CMA Cross Memory Attach.

CPU Central Processing Unit.

DAPL Direct Access Programming Library.

EM Entry Method.

GNI Generic Network Interface.

HPC High Performance Computing.

NIC Network Interface Card.

OFA Open Fabrics Alliance.

OFI Open Fabrics Interface.

PAMI Parallel Active Messaging Interface.

RCoE RDMA over Converged Ethernet.

RDMA Remote Direct Memory Access.

RTS Run Time System.

TCP Transmission Control Protocol.

TMI Tag Matching Interface.

UDP User Datagram Protocol.

CHAPTER 1

INTRODUCTION

The last few decades have seen a tremendous rise in processor performance. This exponential increase in processor performance has been beneficial not only to the scientific community but all the users of computing. However, memory performance has been observed to be improving at a much slower rate resulting into a larger gap between CPU and memory performance. Inspite of growing CPU performance, the slower memory operations continue to be one of the biggest bottlenecks for application performance and scaling.

Message passing in parallel computers typically involves memory allocation and copying. As the effects of cache flushing and page swapping occur more often with increasing data sizes, the performance of memory allocation and copy drastically reduces with increasing message sizes. We will now introduce the networking capability called RDMA, which helps improve performance by kernel bypass and supporting zerocopy networking. Then we discuss the Charm++ programming model and runtime system and set the ground for the thesis topic to exploit opportunities to use zerocopy networking in Charm++.

1.1 RDMA

Traditionally, in the case of a non-RDMA enabled network like User Datagram Protocol (UDP) or Transmission Control Protocol (TCP), when a machine has to send a buffer in its memory location to another machine, depending on the network, the data may get copied several times in the network software stack before it reaches the host machine network adapter. Similarly, on the remote machine, as the packets are received, they are copied across the network software stack before reaching the application memory on the receiver. These copies slow down the performance of the CPU by causing potential page swaps on the machine. Additionally, as the CPUs handle the data transfer between the machines, CPU cycles that could've been used for other tasks are used up in data transfer.

RDMA stands for Remote Direct Memory Access and is a network capability that allows a machine to read from or write to a remote machine without the involvement of the CPUs. In an RDMA enabled network, the the responsibility of data transfer is offloaded to the network adapter. This allows the CPUs on both the sender and receiver machine to be bypassed and these cycles could be used for other useful application work.

The RDMA enabled network adapter is also capable of zero-copy networking. This allows the network adapter to directly read from or write to the original buffer by eliminating all the copies in the software stack. Therefore, the network adapter of the host machine directly reads from the original buffer and sends the data. Similarly, on the remote machine, the network adapter directly writes into the application buffer.

The bypassing of CPU along with the elimination of copies ensure lower latencies and higher throughput for RDMA enabled networks over regular networks.

There are two types of RDMA data transfer mechanisms: 1. PUT The PUT call is used when a host machine can write to or put the data into the memory location of a remote machine. 2. GET The GET call is used when a host machine can read from or get the data from the memory location of a remote machine.

Because of the low-latency and high-throughput messaging required in HPC, almost all the networks used in today's supercomputers are RDMA enabled. The RDMA enabled networks (software interfaces) are:

1. Parallel Active Messaging Interface (PAMI) by IBM
2. Generic Network Interface (GNI) by Cray
3. Infiniband Verbs
4. Open Fabrics Interface (OFI) by Intel
5. RDMA over Converged Ethernet (RCoE)

1.2 Charm++

The Asynchronous message driven programming model has been found to be beneficial for a number of parallel applications. In this model, the application flow is driven by message passing between the interacting entities. An entity here could be a thread, a process or any other form of virtualization used. When an entity receives a message, the corresponding handler method is invoked and the message is provided as an argument to the handler method. With such a scheme, the user code can send a message without the receiver having to explicitly know about the arrival of the message. Very often, a message driven model is implemented with a scheduler on each processor to maintain a queue of received messages. It is the scheduler that continuously polls for messages and on receiving messages, invokes their corresponding handler. This message driven model of parallel programming makes it possible to adaptively overlap communication and computation.

Charm++[1] is an asynchronous programming model and runtime system based on object based virtualization. With object based virtualization, users view the system as a collection of interacting objects spanning across the set of physical processors. The Charm++ Run Time System (RTS) maps the objects to physical processors. Each of this interacting object is called a chare and contains a set of user defined handler methods called entry methods. A chare interacts with a remote chare by calling its entry method with the desired data. It is the responsibility of the Charm++ RTS to locate the receiving chare and invoke its entry method with the data passed by the sender chare.

A remote entry method invocation in Charm++ is very similar to a local function call performed on an object in C++. The sender chare invokes the remote method on the proxy of the remote chare by passing all the parameters to it. The remote method invocation by the sender is non-blocking and returns immediately without a return value. The programming model also guarantees that overwriting the data passed in the remote method right after the remote method invocation does not affect the data passed to the entry method. The scheduler picks up the message on the processor that the remote chare lives on and calls the remote method of the chare by passing the data passed to it by the sender chare.

This user view of invoking entry methods is provided by the programming model along with the Charm++ RTS. The RTS is responsible for the buffer management at both the sending and the receiving side. At the sender side, the parameters passed by the user are marshalled into a contiguous message and sent across the network by the RTS. At the receiver side, after the scheduler picks the message, the message is unmarshalled into individual parameters and then passed to the remote chare's method.

The marshalling of parameters involves copying the parameters into a single contiguous buffer which is sent across the network as a message. This buffer allocation and copying ensures that the user passed parameters are safe to be overwritten immediately after the remote method call. At the receiving side, the parameters are unmarshalled out of the message and passed to the entry method. To ensure the safe memory management of this message i.e avoid a potential memory leak, the RTS frees the message at the end of the remote method invocation. For this reason, the parameters of the entry method have a lifetime only until the end of the scope of the remote method. In order to store the data received as a parameter, the user has to copy the data into a user data structure and use that after the execution of the remote method.

Therefore, two copies are required for sending data from a chare to another remote chare : one at the sender side and one at the receiver side. Even if the Charm++ RTS were to use RDMA to send the message, that cannot avoid the copying costs during marshalling and unmarshalling of messages. In this thesis, we design a zero copy API in Charm++ to avoid both the copies. We implement the API using the RDMA capability of the underlying network on different network platforms.

CHAPTER 2

ZERO COPY API IN CHARM++

2.1 Motivation

Invoking an entry method in Charm++ to send a buffer of data from a user data structure on the sender process to another user data structure on the receiver process requires two copies.

The first copy is made inside the sending process, while marshalling non-contiguous parameters into a large contiguous marshalled message. This scheme allows different parameters stored at non-contiguous memory locations to be copied into a single contiguous message, which will be sent across to the receiver.

The second copy is made inside the receiving process by the user code to copy the data from the received message into the user data structure. As the marshalled messages are managed by the Charm++ runtime system, they are deallocated after the completion of execution of the entry method. For this reason, if the user needs to use the data even after the entry method execution, the data needs to be copied from the received message into the user's data structure.

For large sized buffers that are sized around a few 100 KBs and more, these allocations and copies bloat up the memory footprint of the application causing slowdowns because of data cache misses and page swaps.

Avoiding these additional copies to reduce the memory footprint and accelerate sending and receiving large messages motivated the design for a Zerocopy API in Charm++.

The Zero Copy API in Charm++ is designed in two flavors. These are:

1. Entry Method API :

2. Direct API :

The Entry Method API offloads the user from the responsibility of making additional calls to support zero copy semantics. It extends the capability of the existing entry methods with slight modifications in order to send and receive large buffers without a copy. In cases where the received data needs to obtained in a user specified buffer, the entry method API can be used to avoid both the sender and receiver side copies. In other cases where there is no requirement of receiving the data in a user specified buffer, the user can save only the sender side copy and access the received data using an unmarshalled pointer parameter of the entry method. However, the pointer is valid only in the entry method's scope as the message can be deallocated by the Charm++ RTS after the entry method completes executing.

It can be used to avoid both the sender and receiver side copies or avoid just the sender side copy in cases where there is no requirement to use the received data after entry method execution.

The Direct API allows users to explicitly invoke a standard set of methods on predefined buffer information objects to avoid copies. Unlike the Entry Method API which calls the zero copy operation for every zero copy entry method invocation, the direct API provides more flexibility to the user by allowing the user to invoke the nocopy operations as required by the application. Additionally, the Direct API allows the user to exploit the persistent nature of iterative applications to perform nocopy operations using the same buffer information objects across iterations boundaries.

The following sections describes the usage and semantics of both these APIs in detail.

2.2 Zero Copy Entry Method API

To avoid a sender side copy and send an array using the Zero Copy Entry Method API, the array should be specified in the .ci file with the 'nocopy' keyword to indicate that the array is to be sent using the Zero copy entry method API.

The following code snippet shows the declaration of an entry method containing an rdma parameter:

```
entry void foo(int size, nocopy int arr[size]);
```

In the .C file, while sending the array through the remote method invocation, the array or the pointer should be wrapped in a CkSendBuffer object. The following code snippet shows the remote method invocation call where the pointer is wrapped inside a CkSendBuffer.

```
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr));
```

As there is no copy and the user buffer is directly read from, it is unsafe to modify or free the buffer after invoking an entry method with a nocopy parameter. For this reason, the user can pass a CkCallback object along with the CkSendBuffer wrapper to have the callback invoked on the completion of the zero copy transfer. The following code snipped illustrates the passing of a callback object along with a nocopy buffer.

```
CkCallback Cb(CkIndex_Foo::zerocopySent(NULL),
                                thisProxy[thisIndex];
arrayProxy[0].foo(500000, CkSendBuffer(arrPtr, cb));
```

The callback will be invoked on the completion of the RDMA operation associated with the corresponding array. Inside the callback or any time after the callback has been invoked, it is safe for the sender to write over the buffer sent using the zero copy API. The buffer sent can be accessed inside the completion callback by dereferencing the CkDataMsg received as the callback parameter.

The following code snippet shows an example callback method and illustrates the approach to extract the pointer from the received CkDataMsg.

```
// called when the RDMA operation is completed
void zerocopySent(CkDataMsg *m) {
// get access to the pointer and free the allocation
        void *ptr = *((void **)m->data);
        free(ptr);
        delete m;
}
```

In addition to avoiding the sender side copy, the entry method API also allows to avoid the receiver side copy. In order to avoid the receiver side copy, the user should define an additional function to post the receiver side user buffers where the user expects to receive the data.

The following code snippet shows the .ci file entry method declaration and its corresponding C++ definition.

```
// In .ci file
entry void foo(int size, nocopy int arr[size]);
```

```
// In .cpp file
void foo(int size, int *arr) {
        // function declaration
}
```

The additional function to post the receiver buffers has a similar name and signature to the original function. The name of this function should be "RdmaPost_" prepended to the actual entry method name. The signature is also slightly changed such that each pointer is replaced with a CkRdmaPost struct pointer and the last parameter to the function is a CkRdmaPostHandler pointer.

```
void RdmaPost_foo(int size, CkRdmaPostStruct * postStruct,
                                CkRdmaPostHandle *handle) {


}
```

The function name and signature should confine to these rules in order to successfully overload a runtime defined function of the same signature. Inside the user defined RdmaPost function, the user is allowed to specify pointers for each postStruct object like:

```
postStruct1->ptr = myArr;
```

After specifying the pointer, the user should call the CkRdmaPost function passing the handle object. This operation initiates the RDMA data transfers into the user specified buffers directly, avoiding a receiver side copy. After the completion of the data transfer, the runtime invokes the regular entry method function with the user parameters.

The following example illustrates the definition of the RdmaPost function for the already discussed entry method foo, to avoid receiver side copies.

```
// .ci declaration
entry void foo(int size, nocopy int arr[size]);

// .C RdmaPost function definition
void RdmaPost_foo(int size, CkRdmaPostStruct * postStruct,
                            CkRdmaPostHandle *handle) {
    // Control reaches here first
    postStruct1->ptr = myArr;
    CkRdmaPost(handle);
}
```

```
// .C regular function definition
void foo(int size, int *arr) {
    // Control reaches here after RdmaPost execution
    // arr is the same as user posted myArr

    // use myArr/arr to compute
    computeValues(arr);
    free(myArr);
}
```

In the example discussed, the RTS first calls the method `RdmaPost_foo` and performs the RDMA transfers when the user code invokes `CkRdmaPost`. When the RTS completes all RDMA get operations under the hood, it invokes the regular entry method definition `foo` indicating that the RDMA transfers have been received in the user's posted buffers.

2.3 Zero Copy Direct API

To send an array using the zero copy Direct API, the user should define a CkNcpySource object on the sender chare specifying the pointer, size and a CkCallback object.

The following code snippet shows the declaration of a CkNcpySource object.

```
CkCallback srcCb(CkIndex_Ping1::sourceDone,
                        thisProxy[thisIndex]);
CkNcpySource source(arr1, arr1Size * sizeof(int), srcCb);
```

The callback is specified to notify the user on the completion of a get or put operation.

Similarly, to receive an array using the zero copy Direct API, the user should define a CkNcpyDestination object on the receiver chare object specifying the pointer, the size and a CkCallback object.

The following code snippet shows the declaration of a CkNcpyDestination object.

```
CkCallback destCb(CkIndex_Ping1::destinationDone,
                      thisProxy[thisIndex]);
CkNcpyDestination dest(arr2, arr2Size * sizeof(int), destCb);
```

Similar to the CkNcpySource, the callback is specified to notify the user on completion of a GET or PUT operation.

Once the CkNcpySource and CkNcpyDestination objects have been defined on the sender and receiver chares, to perform a GET operation, the user should send the CkNcpySource to the receiver chare. This can be done using a regular entry method invocation as shown in the code snippet where the sender, arrProxy[0] sends its source object to the receiver chare, arrProxy[1].

```
// On Index 0 of arrProxy chare array
arrProxy[1].recvNcpySrcObj(source);
```

After receiving the sender's CkNcpySource object, the receiver can perform a GET operation on its CkNcpyDestination object by passing the CkNcpySource object as an argument to the runtime defined rget method as shown in the code snippet.

```
// On Index 1 of arrProxy chare array
// Call rget on the CkNcpyDestination object passing the
// CkNcpySource object
dest.rget(source);
```

This call performs an rget reading the remote source buffer into the local destination buffer.

Similarly, a receiver's CkNcpyDestination object can be sent to the sender for the sender to perform a PUT on its source object by passing the CkNcpyDestination object as an argument to the runtime defined rput method as shown in the code snippet.

```
// On Index 1 of arrProxy chare array
arrProxy[0].recvNcpyDestObj(dest);
```

```
// On Index 0 of arrProxy chare array
// Call rput on the CkNcpySource object passing the
// CkNcpyDestination object
source.rput(dest);
```

After the completion of either an rget or an rput, the callbacks specified in the CkNcpySource and CkNcpyDestination are both invoked. Within the callback, the buffer can be safely modified or freed "“"”"' as shown in the code snippet.

```
// Invoked by the runtime on source (Index 0)
void sourceDone() {
    // update the buffer to the next pointer
    updateBuffer();
}
```

```
// Invoked by the runtime on destination (Index 1)
void destinationDone() {
    // received data, begin computing
    computeValues();
}
```

CHAPTER 3

IMPLEMENTATION

The Charm++ software stack consists of three independent software layers

- Charm++
- Converse
- 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 treats the system as a set of processes and is agnostic to chares. It supports message handling and uses a scheduler to queue 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 API used by Converse to perform networking operations like sending and receiving messages. Each networking machine layer implements this set of API using provider specific implementations and semantics. The key design principle for the LRTS layer is to unify the underlying network fabric dependent layers to implement functionality using a common API used by the upper layers.

The Zerocopy API is implemented across different network fabrics using the Low Level Runtime System of Charm++. All the methods used for the Zerocopy Entry Method API and the Zerocopy Direct API are designed as LRTS methods which are implemented by each networking layer.

3.1 PAMI

PAMI stands for Parallel Active Messaging Interface and is the low level messaging API developed for IBM supercomputers. It comprises of improvements over the previously developed LAPI (Low level API) and Blue Gene/P DCMF framework. PAMI offers active messaging semantics where a PAMI message handler function can be set in the message header. On receiving the message, the message handler function is invoked. The PAMI port of Charm++ [2] is designed such that it uses `PAMI_Immediate_Send` and `PAMI_Send` for small and medium sized messages respectively. On the receiver, corresponding callbacks are called to receive the message and appropriately call the converse message handler, which then calls the charm message handler, which eventually invokes the entry method. For large messages, the rendezvous protocol is used to send a small message to the receiver describing the buffer information. On receiving the buffer information, the receiver uses it to perform a `PAMI_Rget` call to remotely read using the RDMA capability of the network.

Since the PAMI communication API natively supports RDMA capabilities and provides calls to perform RDMA gets and puts, the PAMI based Zero copy API implementation uses the rendezvous protocol to transfer large buffers. With this protocol, for performing a `PAMI_Rget`, the process that contains the source buffer sends the buffer information to the destination process to perform a `PAMI_Rget` to remotely read the source buffer using RDMA. To perform a `PAMI_Rput` operation, the process that contains the destination buffer sends the buffer information to the source process to perform a `PAMI_Rput` to remotely write into the destination buffer using RDMA.

PAMI supports creation of overlapping memory regions that represent regions of pinned memory for performing onesided operations. In the PAMI port for Charm++, a kernel memory region is created for the entire available process memory using a call to `Kernel_CreateMemoryRegion`. PAMI uses communication contexts to drive network communication between interacting processes. Using the created kernel memory region, the PAMI port for Charm++ creates overlapping memory regions for each of the contexts configured to be used for a process. The remote buffer information that is required by a process to perform an RGET or RPUT operation includes the remote memory region and the offset of the buffer from the base address in the memory region.

Since an `PAMI_Rget` or `PAMI_Rput` operation is asynchronous and non-blocking, the completion management is handled by the PAMI runtime system invoking a user passed `done_fn`. For the Zerocopy API, we use a method that is different from the `done_fn` used for regular messages. Inside the `done_fn` for the Zerocopy API's GET or PUT calls, the method handler is called to invoke the entry method or callback to indicate the completion of the operation on the local process that performed the RDMA operation. Additionally, a small acknowledgment message is sent to the remote process to signal the completion of the operation.

3.2 Infiniband Verbs

The Infiniband architecture is one of the most commonly used network architectures in HPC machines. It was created to provide direct access to the network interface like the Network Interface Card (NIC) from the application space and for applications to exchange data directly between their respective virtual buffers over a network without the intervention of the Operation System in address translation and networking. `Infiniband Verbs` is a software API developed to facilitate networking operations on the Infiniband network architecture. The Verbs port of Charm++ uses two different schemes for sending messages. For small messages, it uses the eager protocol where the data is tagged with the tag `INFIPACKETCODE_DATA` and is sent using the `ibv_post_send` function. On the receiver side, there are pre-posted buffers which receive the small messages and repost the buffer after invoking the message handler. For large messages beyond a threshold size, the buffer information without the data is sent to the receiver as a short message which is tagged with the tag `INFIRDMA_START` to identify it as an RDMA buffer information message. On receiving messages with this tag, the receiver process performs an RDMA GET operation using `ibv_post_send` function with the `op_code` parameter of the input data structure set to `IBV_WR_RDMA_READ` in order to remotely read using the RDMA capability of the network. Completion queues are used in the Verbs port in order to determine completion of both send and receive queues.

Since the Infiniband network architecture natively supports RDMA capabilities and the Verbs API provides calls to perform RDMA gets and puts, the Verbs based Zero Copy API implementation uses the RDMA calls directly to transfer large buffers. To perform an RDMA Get operation, the process that contains the source buffer sends the buffer information to the destination process to perform an RDMA GET using the method `ibv_post_send` with the `IBV_WR_RDMA_READ` opcode and `INFI_ONESIDED` packet type. Similarly, to perform an RDMA Put operation, the process that contains the target buffer sends the buffer information to the source process to perform an RDMA PUT using the method `ibv_post_send` with the `IBV_WR_RDMA_WRITE` opcode and `INFI_ONESIDED` packet type.

To perform an RDMA operation on Infiniband fabric, both the source and target buffers needs to be pinned. Unlike the PAMI port of Charm++, which pins all the available memory, the Infiniband architecture has a limit on the amount of memory that can be pinned or registered. It allows explicit pinning from the user code using the `ibv_reg_mr` method. The result of an `ibv_reg_mr` operation is a registered memory region represented by the `ibv_mr` type. In the Zerocopy API, the buffer is explicitly pinned before sending the buffer information across to the remote process. After pinning, the obtained `ibv_mr` variable is used to obtain an 8 byte integer value called `key` that is representative of the registered buffer. Both the key and the pointer to the registered memory region are sent as a part of the buffer information.

Since the `ibv_post_send` call to perform a GET or PUT operation is asynchronous and non-blocking, the completion management of RDMA operations is handled by repeatedly polling the completion queue using the method `ibv_poll_cq`. The completion queue returns a list of completed operations and this list is searched for RDMA operations using the opcode variable. After determining the completed RDMA operations, among those, the Zerocopy API operations are determined by searching for `INFI_ONESIDED` packets. At this point, the entry method or callback is invoked to indicate the completion of the operation on the local process that performed the RDMA operation. An acknowledgmenet message is sent to the remote process to signal the completion of the operation.

3.3 GNI

uGNI (or simply GNI) stands for User Level Generic Network Interface and is the low level messaging API developed for the Cray supercomputers. They provide a software API to interact with the `Gemini` interconnect used in Cray XK supercomputers and the newer `Aries` interconnect used in Cray XC supercomputers. The GNI port of Charm++ [3] is designed such that short messages and large messages are sent and received differently. The GNI port uses specialized short message or SMSG methods that are optimized for short message communication. As a requirement for using the short message channel, each peer-to-peer connection requires to create `GNI mailboxes` using a portion of allocated and registered memory. As GNI provides a lightweight event notification mechanism called Completion Queues (CQ), the GNI port also uses completion queues for tracking sends and receives. Specifically, it uses the `GNI_SmsgSendWTag` method with the `SMALL_DATA_TAG` for sending short messages and uses completion queues to poll messages from the mailbox. On receiving each message, appropriate action can be taken depending on the received message's tag. For large messages beyond a threshold size, the buffer information without the data is sent to the receiver as a short message with the tag `LMSG_INIT_TAG`. On receiving this buffer information, the receiver process uses a memory region from an already created allocated memory pool and calls `GNI_PostRdma` to perform the RDMA GET operation.

Since both `Gemini` and `Aries` interconnect natively support RDMA capabilities and the GNI API provides calls to perform RDMA operations, the GNI based Zero Copy API implementation uses RDMA calls directly in order to perform large buffer transfers. To perform an

RDMA Get operation, the process that contains the source buffer sends the buffer information to the destination process to perform an RDMA GET using the method `GNI_PostRdma` with the post descriptor type set to `GNI_POST_RDMA_GET`. Similarly, to perform an RDMA Put operation, the process that contains the destination buffer sends the buffer information to the source process to perform an RDMA PUT using the method `GNI_PostRdma` with the post descriptor type set to `GNI_POST_RDMA_PUT`.

To perform an RDMA operation using the GNI network API on the Aries and Gemini interconnect, both the source and destination buffers need to be pinned or registered. Similar to the Infiniband fabric, these interconnects also have a limit on the amount of memory that can be registered. It allows explicit pinning from user code using the `GNI_MemRegister` method. An `GNI_MemRegister` method call writes into an input `gni_mem_handle_t` object that represents a registered region of memory. After registering, the `gni_mem_handle_t` object is sent to the remote process as the buffer information.

The GNI API method `GNI_PostRdma` when called to perform an RDMA GET operation requires an alignment condition to be satisfied. This condition is to have 4-byte alignment for the source address, destination address and the length of the buffer that is being read from the source address and written into the destination address. However, such a restriction does not exist for the RDMA PUT operation offered by the GNI API. For this reason, in those cases where either one of : source address, destination address or buffer length is not 4-byte aligned, the RDMA GET operation is internally implemented by performing an RDMA PUT operation. This leads to the destination buffer information being sent back to the source process and that process performing an RDMA PUT on the destination process using `GNI_PostRdma` with the post descriptor type set to `GNI_POST_RDMA_PUT`.

The completion management for asynchronous RDMA calls in the Zerocopy API for the GNI port is handled in a similar manner as the Verbs port. A separate completion queue is used for handling RDMA transfers using the Zerocopy API. This completion queue is polled regularly using the method `GNI_CqGetEvent`. The entry method or callback is invoked to indicate the completion of the operation on the local process that performed the RDMA operation. An acknowledgement message is sent to the remote process to signal the completion of the RDMA transfer.

3.4 Intel OFI

Intel Omni-Path is a newer network architecture developed as a part of Intel's Scalable System Framework. It's a successor to Intel True Scale Fabric and aims to provide numerous optimizations in congestion control and routing. There are multiple software layers that can be used to benefit from the Omni-Path network architecture. These include Direct Access Programming Library (DAPL), Tag Matching Interface (TMI), Open Fabrics Alliance (OFA) and OFI. Out of these, since OFI or the libfabric library aims to unify multiple backend providers like GNI, Verbs, PAMI, PSM2 etc, we decided to implement an OFI port for Charm++. It was newly contributed to Charm++ by developers from Intel.

OFI[4] or libfabric is a low level generic networking API that abstracts underlying network specific implementation details and allows users to design networking layers such that it supports a subset of network specific providers based on user based feature requirements. For example, on a machine with multiple providers, a user can decide to use only those providers that have RDMA support. In addition to choosing network providers in the code, the user can also set environment variables at runtime to define the subset of available providers.

The OFI port of Charm++ is designed such that it uses the eager protocol for small and medium sized messages and the rendezvous based RDMA operations for transfer of large messages beyond a threshold. For small messages, the payload is sent as a single message using `fi_inject`. This scheme guarantees immediate completion and does not require to wait or use a callback for completion management. For sizes greater than `context.inject_maxsize`, the regular send using tagged messaging is used through the method `fi_tsend`. Both small and medium sized messages are tagged with `OFI_OP_SHORT`. On the receiver, there are pre-posted buffers which receive these messages and repost the buffer after invoking the message handler. For large messages, the buffer information without the data is sent to the receiver as a short or medium sized message which is tagged with `OFI_OP_LONG` to identify it as an RDMA buffer information message. On receiving messages with this tag, the receiver process allocates a buffer and performs an RDMA GET operation using `fi_read` in order to remotely read using the RDMA capability of the network. Very similar to Verbs and GNI, completion queues are used to determine completion of both send and receive operations.

The OFI or libfabric based Zero Copy API implementation uses the RDMA calls directly made available by the libfabric library. To perform an RDMA Get operation, the process that contains the source buffer sends the buffer information to the destination process to perform an RDMA GET using the `fi_read` method. Similarly, to perform an RDMA Put operation, the process that contains the destination buffer sends the buffer information to the source process to perform an RDMA Put using the `fi_write` method.

Similar to the Verbs and GNI layers, the OFI layer too requires explicit memory registration. It allows pinning from the user code using the `fi_mr_reg` method. The result of this call is a registered memory region represented by an `fid_mr` object. This object is sent as a part of the buffer information to either read from or write into the registered memory region.

The completion management in the OFI layer is performed using completion queues similar to Verbs and GNI. Completion queues are used and can be queried or polled regularly using `fi_cq_read`. When completion queues return entries, their respective callback function pointer is invoked. This function pointer is appropriately configured depending on the type of the message. Inside the function used as a Zerocopy API RDMA transfer callback, the entry method or callback is invoked locally and an acknowledgement message is sent to the remote process to signal the RDMA transfer completion.

3.5 MPI

MPI[5] is a Message Passing Library standard based on the consensus of the MPI Forum. It is the most commonly used middleware used for writing HPC applications across the world. It defines the standard for moving data from one process to another process through cooperative operations on each process. The most widely used MPI implementations are: MPICH, Open MPI, MVAPICH, Intel MPI and Cray MPI. The MPI port of Charm++ uses the MPI library as the networking layer. The motivation for developing an MPI port for Charm++ was allowing Charm++ applications to use the MPI networking layer where native networking implementations were not developed.

The MPI port of Charm++ uses `MPI_Isend` with a constant tag value for sending messages. As it a non-blocking call that doesn't wait for completion, the MPI port uses `MPI_Test` to wait for the completion of the sending. On the receiving side, `MPI_IProbe` is called repeatedly with the same constant tag value to check for any incoming messages. Getting a positive value on the input flag indicates that there are messages being sent to my process. With a call to `MPI_Get_Count`, the size of the incoming message is determined and this allows the receiver process to allocate a buffer of that size and post a matching `MPI_Recv` to receive the posted message. As the message is ready to be received, there is no additional waiting that occurs at `MPI_Recv`. The blocking call ensures that the buffer is entirely received.

As the MPI provided RMA methods require every participating process to create a window and then interact using participating windows, it doesn't fit well with Charm++'s point to point messaging that does not require every process to participate in messaging. For this reason, the Zerocopy API implementation of the MPI layer also uses `MPI_Isend` and matching `MPI_Irecv`, but without using a constant tag. To ensure that the user's source buffer gets received in the user's receive buffer, the matching send and receive should have a unique tag to not interfere with other posted sends and receives. For this reason, the tag used is a unique integer that is incremented for every `MPI_Isend` or `MPI_Irecv` posted and reset when the value reaches the allowed upper bound.

To implement a Get operation using MPI, a unique tag is obtained by incrementing the existing tag value. Then the destination process sends a small buffer information message consisting of the unique tag to the source process. Then, it posts an `MPI_Irecv` with the destination buffer pointer and the unique tag. On the sender process, on receiving the buffer information message, it posts an `MPI_Isend` with the source buffer pointer and the unique tag to match the receive posted by the destination process. This results in completion of the matching with the unique tag and transfer of the large buffer. Similarly, to implement a Put operation using MPI, a unique tag is obtained by incrementing the existing tag value. Then the source process sends a small buffer information message consisting of the unique tag to the destination process. Following that, it posts an `MPI_Isend` with the source buffer pointer and the unique tag. On the destination process, on receiving the buffer information message, it posts an `MPI_Irecv` with the destination buffer pointer and the unique tag to match the send posted by the source process. This results in completion of the matching with the unique tag and transfer of the large buffer.

For both the Get and Put operations, the completion management is performed by calling `MPI_Test` on the request objects. Appropriate callbacks or handlers are called when it is detected that `MPI_Isend` or `MPI_Irecv` has completed as that indicates the completion of the Zerocopy API operation.

CHAPTER 4

RESULTS

In the following section, we compare and analyze the performance of the pingpong micro-benchmark implemented with the regular charm++ messaging API, the Zerocopy Entry Method API and the Zerocopy Direct API. This performance evaluation is conducted on a wide range of supercomputers using different network architectures and interconnects. The pingpong micro-benchmark is a simple program that uses two computing entities like chares or processes that send and receive the same message. The program captures the one-way time taken for sending a message of a particular size by timing the duration between sending and receiving back the same message and using half the duration as the one-way time.

In our experiments, we have run the pingpong benchmark on two processes spread across two physical nodes such that one process resides on each node. This was done to exercise the the network adapters and links. The pingpong results compare the following timings:

- Regular Send API - Time taken by the regular messaging API for sending a message to a remote host. The message is received in the entry method only and not in the user's data structure.

- Zerocopy Entry Method Send API - Time taken by the Zerocopy Entry Method API for sending a message to a remote host. The message is received in the entry method only and not in the user's data structure.

- Regular Send and Receive API - Time taken by the regular messaging API for sending a message to a remote host and that host receiving it into a user specified buffer. Since the regular messaging API in Charm++ does not provide the functionality of receiving a message into a user specified buffer, we have explicitly copied the message into the user buffer using a memcpy instruction.

- Zerocopy Entry Method Send and Receive API - Time taken by the Zerocopy Entry Method API for sending a message to a remote host and that host receiving it into a user specified buffer.

- Zerocopy Direct API (GET) - Time taken by the Zerocopy Direct API for sending the buffer information to a remote host and that host performing an RDMA Get to receive that source buffer into a user specified buffer.

- Zerocopy Direct API (PUT) - Time taken by the Zerocopy Direct API for requesting the source buffer information from a remote host, getting that buffer information back and then performing an RDMA Put into the user specified buffer.

  As the Regular Send API and Zerocopy Entry Method Send API perform only the send operation and do not receive the data into the user's data structure or posted buffer, they are compared with each other for evaluating the performance of the Zerocopy Entry Method Send API. Similarly, as the Regular Send and Receive API, the Zerocopy Entry Method Send and Receive API and both forms of Zerocopy Direct API perform both sending and receiving from the user's data structure, they are compared with each other for evaluating their performance.

  The plots have Message Size in KiloBytes(KB) or MegaBytes (MB) plotted on the X-axis and the one-way time in microseconds (us) plotted on the Y-axis. Both these values are plotted in logarithmic scale.

4.1 PAMI

Table 4.1 shows the result of running the pingpong benchmark on two processes across two nodes of Vesta, a BG/Q machine at ALCF. The plot 4.1 shows the one way time for the Regular Send API and the Zerocopy Entry Method (EM) Send API plotted against the message size. As seen in the table and the plot 4.1, starting from 512 KB, the Zerocopy EM Send API performs better than the regular send API. We achieve a performance improvement of 5% at 512 KB and up to 23% at 16 MB. Similarly, the plot 4.2 shows the one way time for the Regular Send and Receive API, the Zerocopy EM Send and Receive API and the Zerocopy Direct API using GET. As seen in the table and the plot, the Zerocopy Entry Method Send and Receive API starts performing better than the Regular API at 128 KB and the Zerocopy Direct API at 64 KB. We achieve a performance improvement of 1.5% and 3% at 128 KB and 64 KB and go up to 38% improvement at 64 MB.

Table 4.1: Performance of the Zerocopy API with Pingpong Benchmark on the PAMI layer

One way Pingpong time for the PAMI port on Vesta (BG/Q)					
Message Size	Regular Send API	Zerocopy EM Send API	Regular Send and Receive API	Zerocopy EM Send and Receive API	Zerocopy Direct API (GET)
2 KB	34.10	67.85	35.57	61.99	47.82
4 KB	37.00	68.02	38.27	62.00	49.02
8 KB	40.28	70.31	42.03	64.52	51.43
16 KB	46.58	74.12	48.74	69.04	56.07
32 KB	57.11	83.43	61.49	78.49	64.66
64 KB	78.80	101.76	86.15	96.77	83.48
128 KB	122.08	138.55	135.77	133.56	121.14
256 KB	208.94	212.58	235.53	207.52	195.19
512 KB	381.90	359.59	434.52	354.99	341.57
1 MB	728.81	655.91	831.49	650.21	636.78
2 MB	1484.07	1245.52	1755.24	1239.89	1228.63
4 MB	3307.57	2676.49	3718.02	2419.50	2407.34
8 MB	6569.11	5282.12	7465.67	4779.38	4767.12
16 MB	13771.92	10565.15	15539.09	9561.57	9560.51
32 MB	29246.51	23730.00	33700.23	21597.18	21573.57
64 MB	57096.48	44976.32	65988.34	40660.49	40644.15

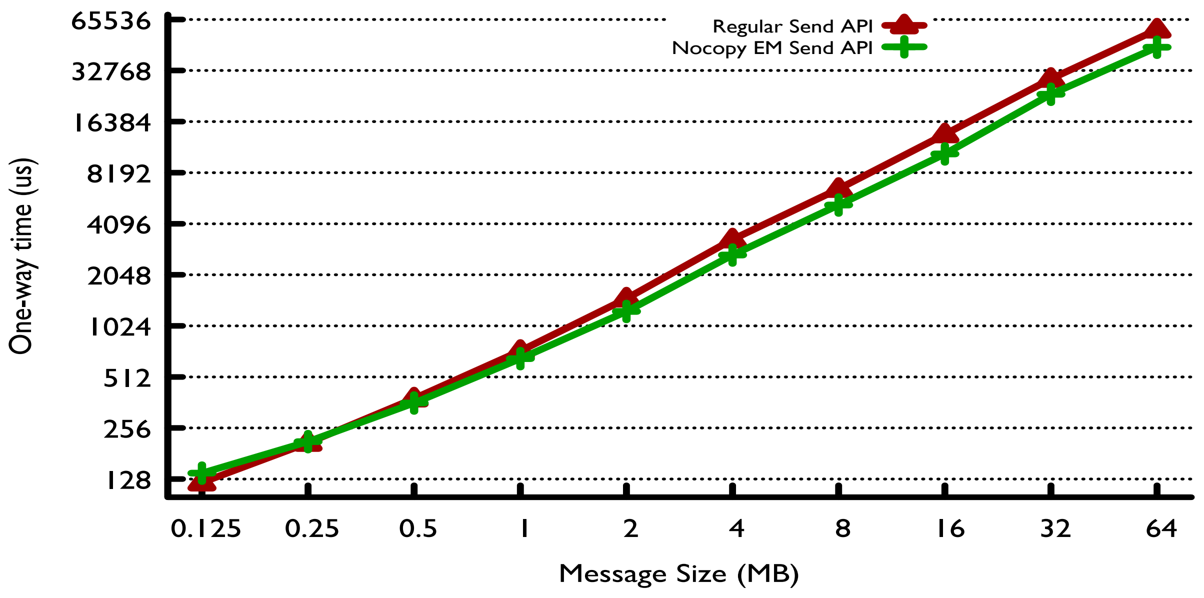

Figure 4.1: Comparing Regular Send API with Zerocopy EM API on the PAMI layer

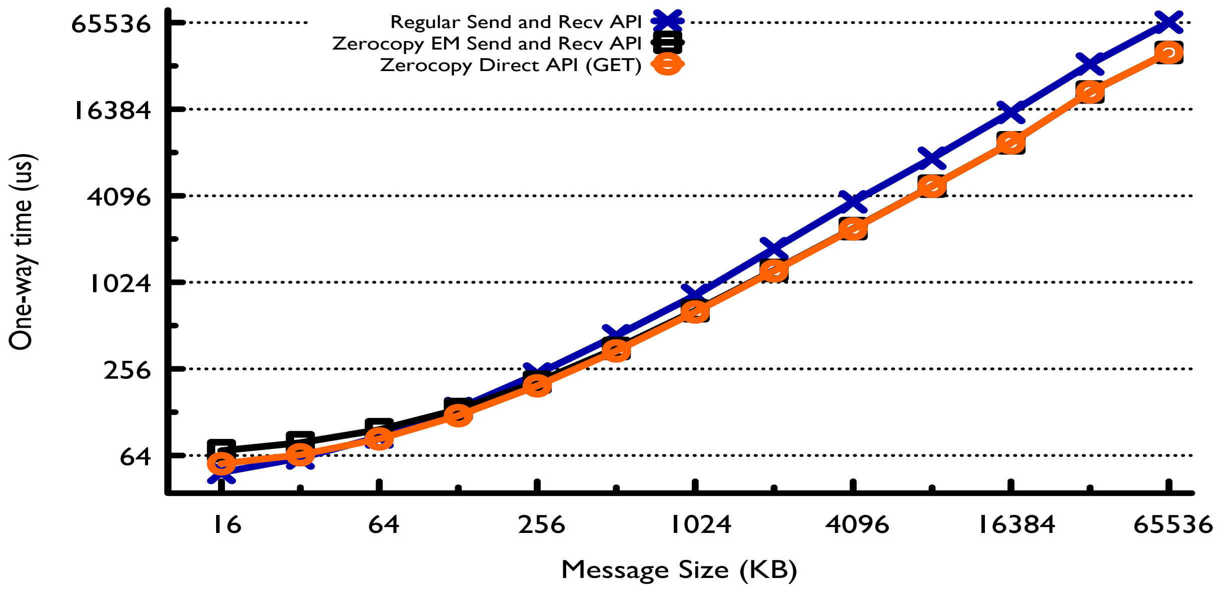

Figure 4.2: Comparing Regular Send-Recv API with Zerocopy Send-Recv APIs on the PAMI layer

4.2 Infiniband Verbs

Table 4.2: Performance of the Zerocopy API with Pingpong Benchmark on the Verbs layer

One way Pingpong time for the Verbs port on Golub (Dell/Intel cluster)						
Message Size	Regular Send API	Zerocopy EM Send API	Regular Send and Receive API	Zerocopy EM Send and Receive API	Zerocopy Direct API (GET)	Zerocopy Direct API (PUT)
2 KB	4.15	114.34	3.99	113.57	6.15	9.85
4 KB	4.98	115.56	4.77	115.37	6.38	10.13
8 KB	6.52	115.48	6.15	115.60	7.32	10.80
16 KB	9.30	120.05	8.92	119.10	8.95	12.50
32 KB	15.64	124.64	15.76	124.63	12.04	15.48
64 KB	24.67	133.03	27.82	133.49	18.13	21.62
128 KB	43.53	150.97	53.55	151.75	30.20	33.76
256 KB	81.57	179.27	103.41	178.61	54.65	58.17
512 KB	159.56	244.22	202.44	242.51	103.98	107.20
1 MB	397.62	421.63	528.40	365.41	201.00	204.48
2 MB	760.64	726.41	970.92	662.34	396.63	401.04
4 MB	1456.88	1348.92	1878.60	1262.71	794.44	839.03
8 MB	6428.19	3835.77	7154.38	2448.19	1658.74	1823.56
16 MB	13891.67	6287.78	15631.23	4846.88	3305.39	3581.27
32 MB	24835.79	17905.08	28174.30	10169.21	6654.32	7141.50
64 MB	50290.13	35370.92	56955.59	19055.91	13259.62	14215.27

Table 4.2 shows the result of running the pingpong benchmark on two processes across two nodes of Golub, a Dell Cluster using Infiniband network at the Univeristy of Illinois. The plot 4.3 shows the one way time for the Regular Send API and the Zerocopy EM Send API plotted against the message size. As seen in the table and the plot 4.3, starting from 2 MB, the Zerocopy EM Send API performs better than the regular send API. We achieve a performance improvement of 4.5% at 512 KB and up to 54% at 16 MB.

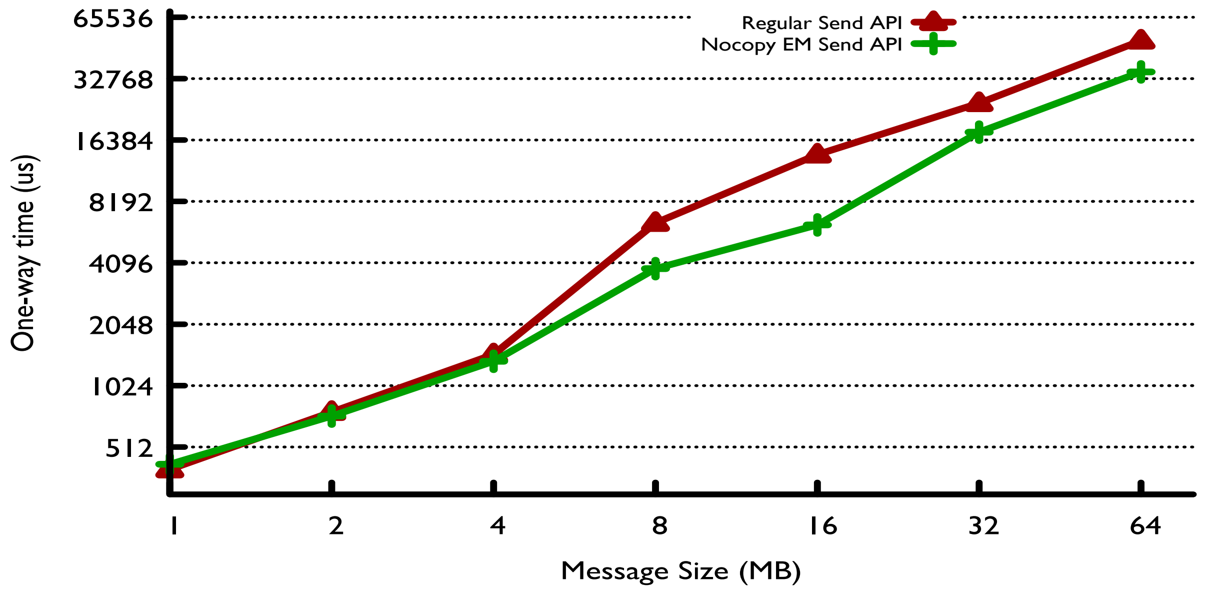

Figure 4.3: Comparing Regular Send API with Zerocopy EM API on the Verbs layer

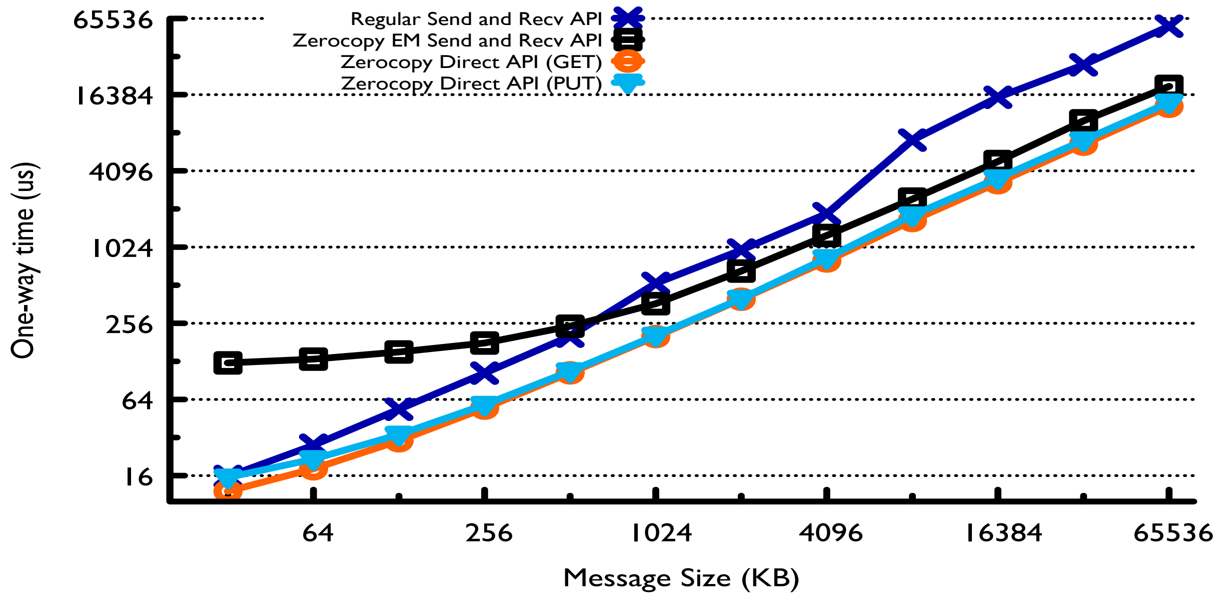

Figure 4.4: Comparing Regular Send-Recv API with Zerocopy Send-Recv APIs on the Verbs layer

Similarly, the plot 4.4 shows the one way time for the Regular Send and Receive API, the Zerocopy EM Send and Receive API and the Zerocopy Direct API using GET. As seen in the table and the plot, the Zerocopy Entry Method Send and Receive API starts performing better than the Regular API at 1 MB and the Zerocopy Direct API at 32 KB. With the EM API, we achieve a performance improvement of 31% at 1MB and go up to 68% at 16 MB.

With the Direct API, we achieve a performance improvement of 23% at 32 KB and go up to 76% at 64 MB.

4.3 GNI

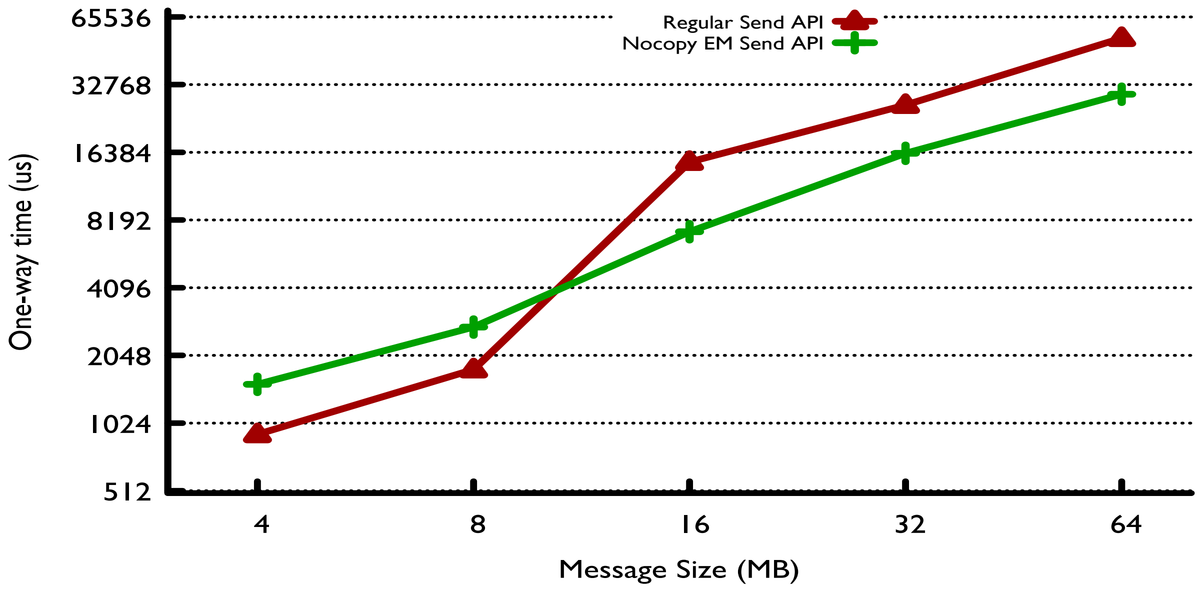

Figure 4.5: Comparing Regular Send API with Zerocopy EM API on the GNI layer

Table 4.3 shows the result of running the pingpong benchmark on two processes across two nodes of Edison, a Cray XC30 supercomputer at NERSC using the Aries interconnect. The plot 4.5 shows the one way time for the Regular Send API and the Zerocopy EM Send API plotted against the message size. It is only at 16 MB and above that the Zerocopy EM API performs better than the Regular API. We only see a slight performance improvement of 1.5% to 2%. The Zerocopy Entry Method API with both Send and Recv too does not perform too well until 8 MB as seen from plot 4.6. However, the Zerocopy Direct API beats the Regular API around 16KB - 32 KB. It performs 1% to 9% better than the Regular Send and Recv API starting from 16 KB to 64 MB.

Table 4.3: Performance of the Zerocopy API with Pingpong Benchmark on the GNI layer

One way Pingpong time for the GNI port on Edison (Cray XC30)						
Message Size	Regular Send API	Zerocopy EM Send API	Regular Send and Receive API	Zerocopy EM Send and Receive API	Zerocopy Direct API (GET)	Zerocopy Direct API (PUT)
2 KB	4.38	589.26	4.50	614.71	5.91	9.02
4 KB	5.54	576.78	5.51	593.00	5.90	9.32
8 KB	5.86	560.79	5.99	589.60	6.71	9.74
16 KB	6.76	587.23	7.82	569.98	7.70	10.69
32 KB	10.95	568.08	13.16	566.73	9.43	12.13
64 KB	15.72	602.76	26.59	623.87	13.28	15.74
128 KB	30.08	626.32	49.09	636.59	21.20	22.48
256 KB	56.59	649.52	95.08	694.77	36.40	35.87
512 KB	108.59	698.80	205.05	791.78	67.68	63.42
1 MB	226.92	759.19	372.59	955.09	157.04	116.94
2 MB	475.25	915.49	828.88	1398.00	307.46	246.20
4 MB	913.32	1523.03	1475.04	2651.53	517.97	458.40
8 MB	1773.81	2738.94	3342.99	3192.85	1025.65	949.98
16 MB	14835.30	7263.10	18455.18	11861.01	2245.69	2031.27
32 MB	26601.09	16218.50	38212.89	28430.86	4589.43	4113.82
64 MB	52790.98	29718.78	81922.82	59242.16	9388.09	8252.03

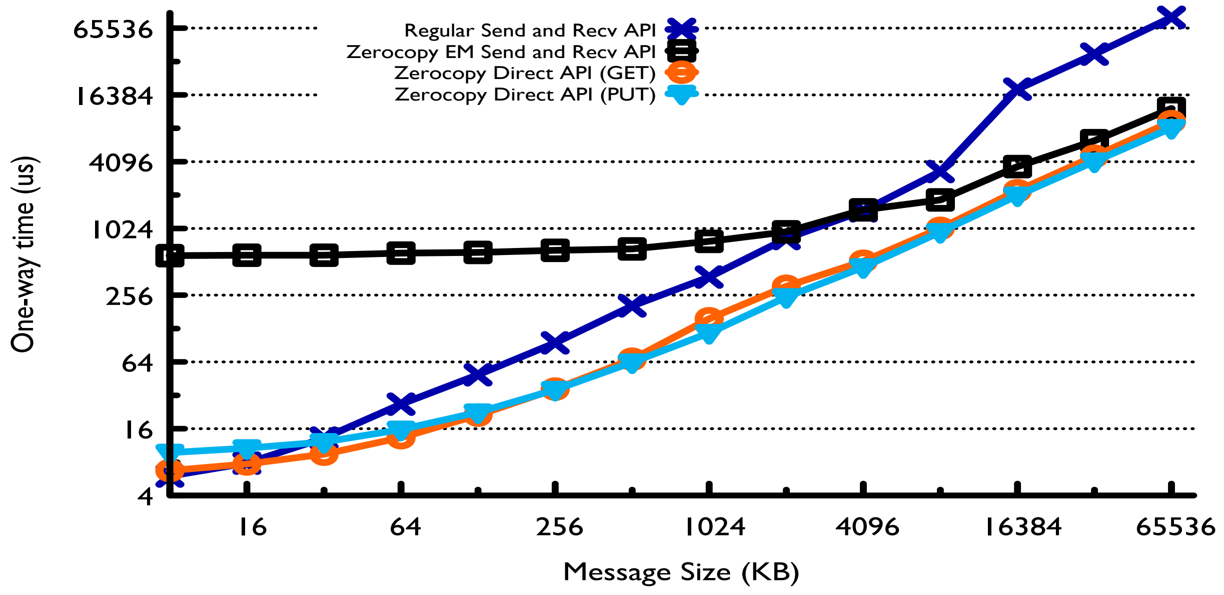

Figure 4.6: Comparing Regular Send-Recv API with Zerocopy Send-Recv APIs on the GNI layer

4.4 Intel OFI

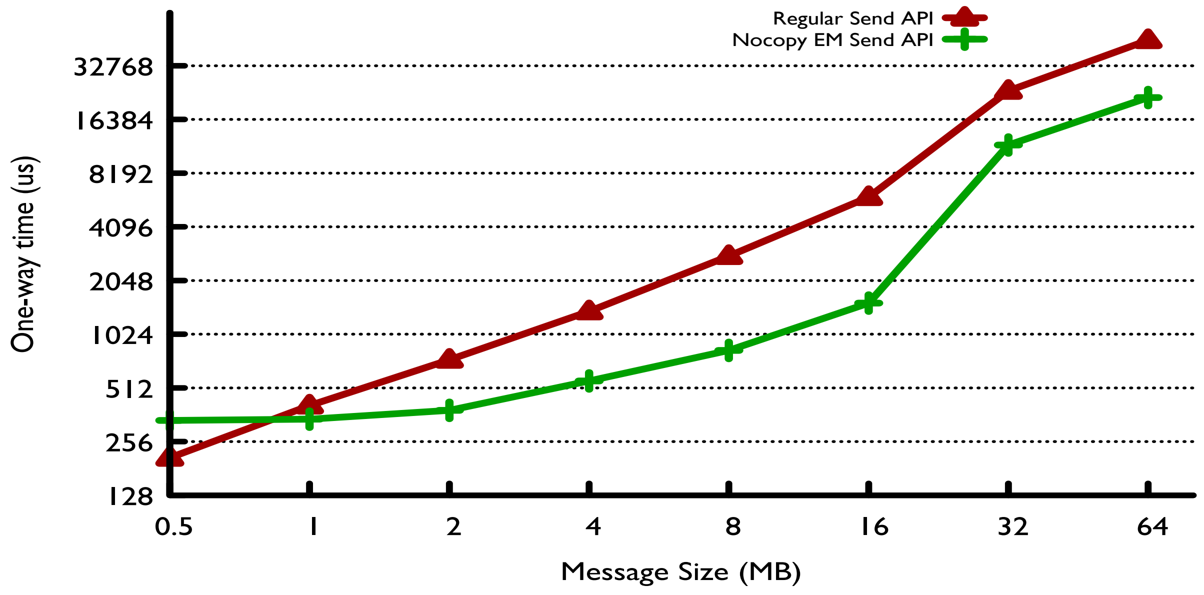

Figure 4.7: Comparing Regular Send API with Zerocopy EM API on the OFI layer

Table 4.4 shows the result of running the pingpong benchmark on two processes across two nodes of Stampede 2, a Intel KNL supercomputer at TACC using the Intel Omni-path network architecture. The plot 4.7 shows the one way time for the Regular Send API and the Zerocopy EM Send API plotted against the message size. As seen in the plot 4.7 and the table, at the size of 1 MB, the Zerocopy EM Send API performs better than the regular Send API by 16% and it improves further up to almost 70% at 16 MB. Additionally, as seen in the plot 4.8 with the Zerocopy Send and Recv API and the Zerocopy Direct API, the improvement is seen to be 40% - 60% at 64 KB and increases up to 90% for the largest messages.

Table 4.4: Performance of the Zerocopy API with Pingpong Benchmark on the OFI layer

One way Pingpong time for the OFI port on Stampede2 (Intel KNL cluster)						
Message Size	Regular Send API	Zerocopy EM Send API	Regular Send and Receive API	Zerocopy EM Send and Receive API	Zerocopy Direct API (GET)	Zerocopy Direct API (PUT)
2 KB	16.79	55.91	16.96	55.28	36.18	74.89
4 KB	18.06	59.45	18.61	56.38	37.95	74.58
8 KB	21.23	68.65	21.46	73.75	42.05	78.41
16 KB	24.69	74.33	25.80	63.67	46.19	89.72
32 KB	30.39	75.55	33.41	68.66	49.97	93.48
64 KB	137.88	147.84	154.33	76.95	57.28	96.81
128 KB	179.06	205.41	191.62	179.36	155.07	205.89
256 KB	215.92	319.49	195.90	181.79	162.64	228.59
512 KB	207.66	336.76	323.97	198.43	154.20	199.82
1 MB	407.83	342.27	605.58	231.88	194.84	255.75
2 MB	736.41	383.23	1060.35	299.68	248.68	298.90
4 MB	1376.30	560.89	1901.06	478.30	453.40	514.70
8 MB	2811.16	831.74	6805.73	814.89	781.65	855.98
16 MB	6008.41	1531.04	16454.11	1499.96	1498.92	1548.02
32 MB	23693.12	11775.96	29109.18	2891.20	2888.36	2916.72
64 MB	45585.29	21727.71	55920.52	5676.88	5666.87	5677.70

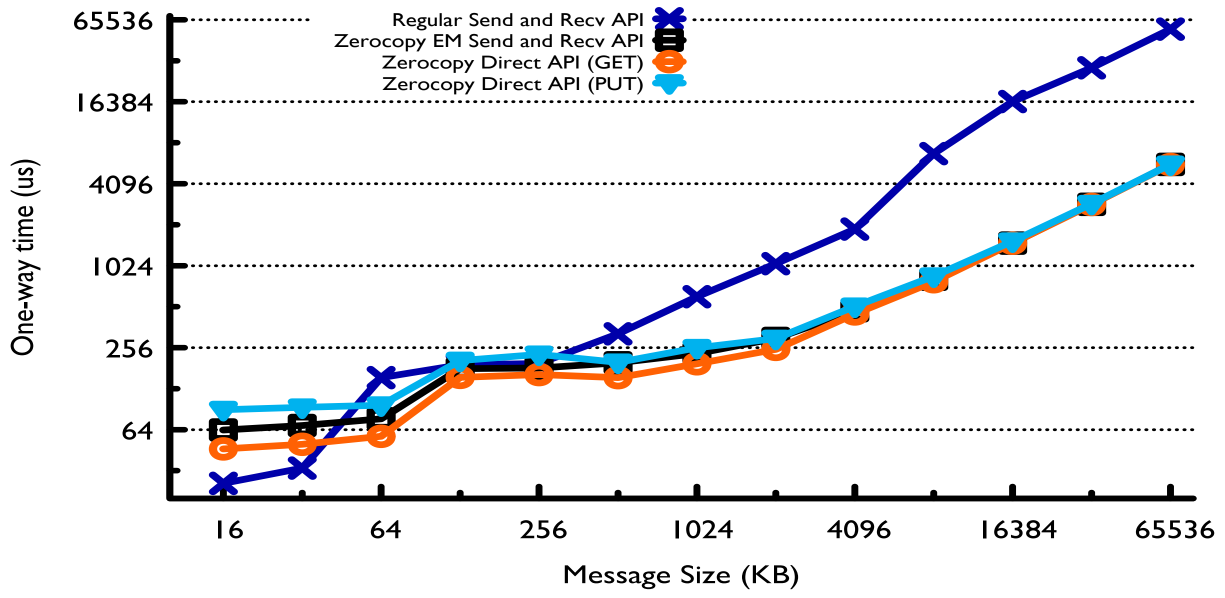

Figure 4.8: Comparing Regular Send-Recv API with Zerocopy Send-Recv APIs on the PAMI layer

4.5 MPI

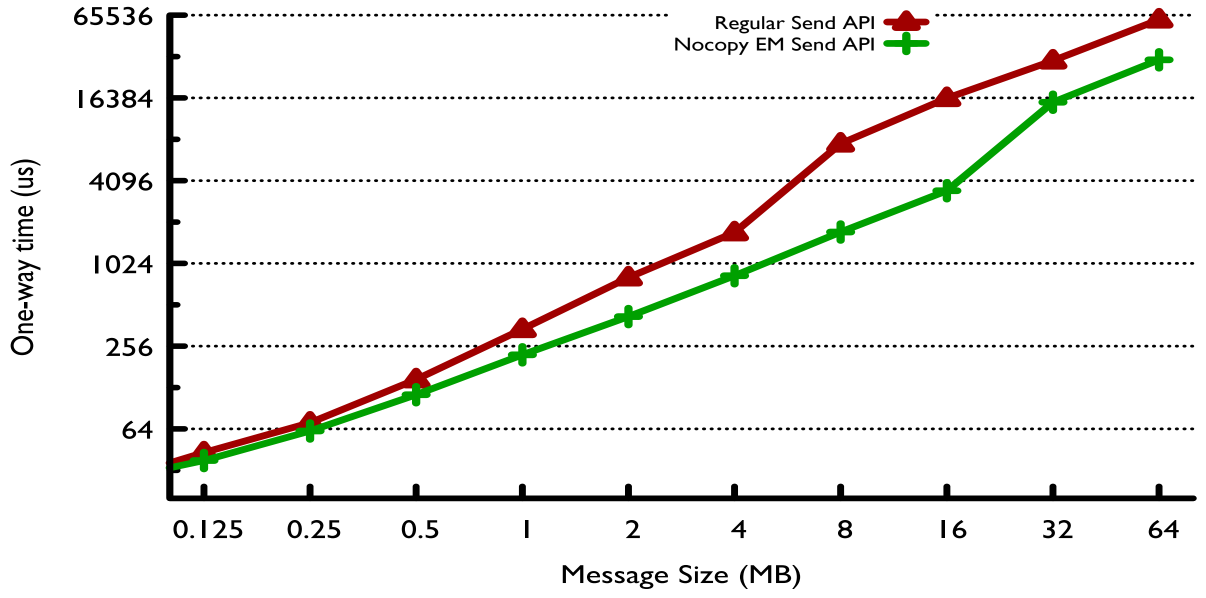

Figure 4.9: Comparing Regular Send API with Zerocopy EM API on the MPI layer

Table 4.5 shows the result of running the pingpong benchmark on two processes across two nodes of golub, a Dell cluster at the University of Illinois. It is the same cluster that the verbs port was tested on. However, this experiment uses the MPI layer as the networking layer.

Table 4.5: Performance of the Zerocopy API with Pingpong Benchmark on the MPI layer

One way Pingpong time for the MPI port on Golub (Intel/Dell Cluster)						
Message Size	Regular Send API	Zerocopy EM Send API	Regular Send and Receive API	Zerocopy EM Send and Receive API	Zerocopy Direct API (GET)	Zerocopy Direct API (PUT)
2 KB	6.52	9.28	6.55	8.91	15.41	15.68
4 KB	7.51	10.19	7.54	9.69	17.47	17.81
8 KB	9.46	12.05	9.59	11.59	21.52	21.70
16 KB	12.34	17.70	14.04	15.54	142.51	143.50
32 KB	17.30	19.83	20.89	18.63	161.83	161.48
64 KB	25.01	25.79	34.00	25.14	191.21	190.87
128 KB	43.09	37.64	60.47	38.30	253.26	253.96
256 KB	71.03	61.77	104.42	64.68	364.52	372.73
512 KB	146.47	113.59	223.30	117.15	601.82	593.40
1 MB	340.50	221.70	571.44	223.03	1136.34	1168.72
2 MB	808.51	419.93	1291.23	434.06	2336.35	2350.12
4 MB	1720.50	833.82	2799.81	860.32	4326.41	4357.69
8 MB	7609.11	1733.76	9192.62	1708.50	8365.46	8365.91
16 MB	16315.04	3461.23	19368.35	3420.91	16673.26	16670.08
32 MB	30628.62	15232.59	39901.88	6807.33	39780.72	39339.44
64 MB	60930.98	31072.79	80223.07	13588.72	77792.86	86901.92

The plot 4.9 shows the one way time for the Regular Send API and the Zerocopy EM Send API plotted against the message size. As seen in the plot 4.9 and the table, the Zerocopy EM Send API performs better than the regular Send API by 12% at 128 KB and it improves further up to almost 78% at 16 MB. Similarly, as seen from the send and recv side plot 4.10, the Zerocopy EM Send API beats the regular API at 32 KB with 10% improvement and it increases up to 83% at 64 MB. However, the Zerocopy Direct API performs poorly in the case of MPI as seen from the plot. Unlike the other layers, the Direct API performs better than the regular API only at 8 MB.

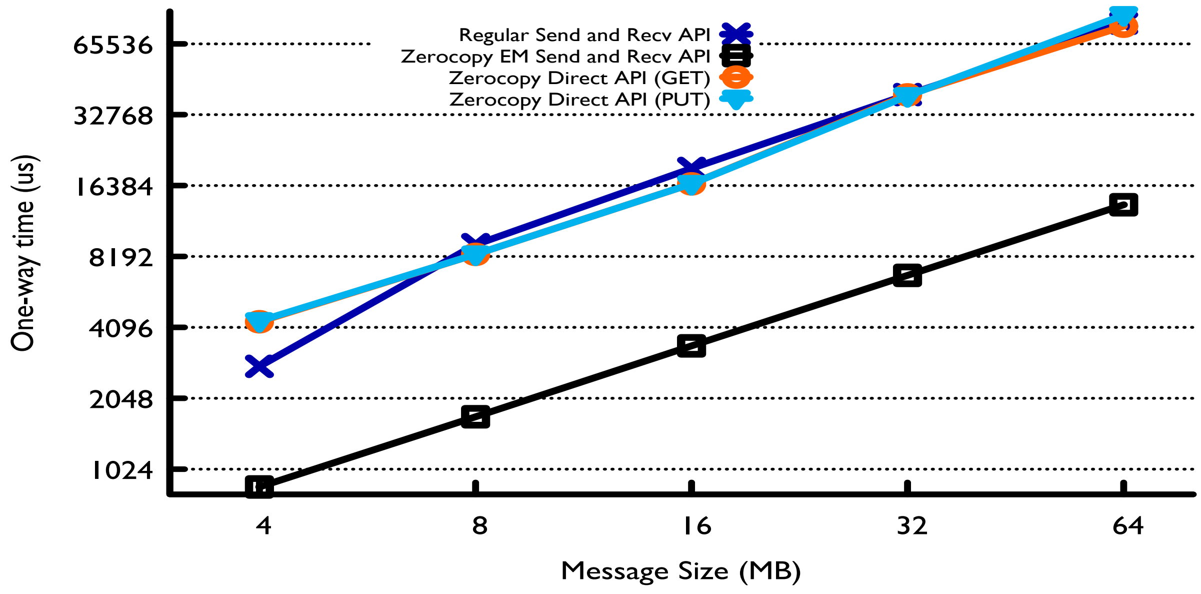

Figure 4.10: Comparing Regular Send-Recv API with Zerocopy Send-Recv APIs on the MPI layer

In all the above network architectures and their implementations, the Zerocopy API performs better than the Regular API for large messages as it saves additional memory allocations and copies. Since the Regular API also uses RDMA Get operations for large messages, the performance benefit for the Zerocopy API is solely due to the avoidance of memory operations i.e the malloc and memcpy operation used to allocate and copy the large message. Therefore, the performance gain is mostly dependent on the processor's memory performance and not so much on the network performance as the network is equally utilized by both the APIs.

Additionally, along with this improved performance, the Zerocopy API also benefits in the application having a reduced memory footprint.

The eager protocol is used in the regular API for small messages up to a threshold size, following which the rendezvous protocol is used. Since the rendezvous protocol is used for all message sizes in the Zerocopy API, the regular API performs better than the Zerocopy API for smaller message sizes. This is because of the time taken for memory registration and the additional round trip used by the rendezvous protocol being large in proportion to the time taken for message allocation, copy and the eager protocol send and receive.

Comparing the different network implementations, we observe that the Zerocopy EM API for PAMI, OFI and MPI implementations perform better than the Verbs and GNI layer implementations. This is because of the expensive memory registration operations that are incurred by the Zerocopy EM API to explicitly register the user buffer for each iteration.

These costs are not incurred by the Regular API for the GNI and Verbs port as it uses mempool implementations that copy the message into memory regions that are already registered with the NIC. As the PAMI port registers the whole kernel memory region, there is no explicit memory registration required for it and this leads to the Zerocopy EM Send API beating the Regular Send API at 512 KB and the Zerocopy EM Send and Receive API beating the Regular Send and Receive API at 128 KB. In the case of the OFI and MPI implementations, the Zerocopy EM API beating the Regular API at smaller message sizes indicate the use of a registration cache at the networking level to avoid the expensive registration costs for each iteration.

The Zerocopy Direct API performs better than the Zerocopy Entry Method Send and Receive API for all layers except MPI as it does not perform memory registration for every iteration unlike the EM API. As it supports operations that are persistent across iteration boundaries, we register the buffer only at the beginning and avoid registration for every iteration. Of course, this is at the cost of a slightly more complex usage model for the application program as the Direct API entails using the same registered memory regions for persistent interactions.

CHAPTER 5

RELATED WORK

CkDirect[6] was the persistent one-sided interface developed for Charm++. It provides functions to create a buffer handle, associate a handle with a local buffer and perform RDMA operations across iteration boundaries. The development of the Zerocopy Direct API is motivated by the CkDirect API. However, the Zerocopy API is much more flexible by not enforcing tight coupling or association between a local and remote buffer. In the case of the Zerocopy Direct API, both the source and destination buffer information objects, referred to as handles by CkDirect are free to be associated with any other handles without any tight coupling. Additionally, the Zerocopy Direct API allows the application to use both RDMA calls: Get and Put; as compared to the CkDirect API, which was entirely implemented using the RDMA Put call. As the Entry Method API uses modifications to the current entry method model, it proves to be an easier alternative to the CkDirect API in terms of usability and modifying existing code that uses large buffers.

In addition to native RDMA based use cases, the Zerocopy API discussed in this thesis also adds the missing pieces for the previously developed CkDirect API by being fully functional for non-RDMA use-cases and platforms. For API calls between two chares on the same process, it uses a `memcpy` call to perform the zerocopy transfer without the network involvement. Similarly, for API calls between two chares on processes that are running on the same physical host, it uses a shared memory communication channel called Cross Memory Attach (CMA) to perform the transfer without the network involvement. On network layers like TCP and UDP, that do not support native RDMA functionality, the Zerocopy API uses a copy based approach to keep the API functional.

MPI also provides one sided primitives in MPI 2 and MPI 3 standards that are similar to the Zerocopy Direct API and the CkDirect API. However, unlike CkDirect or the Zerocopy Direct API, it requires each participating process to create a window object for communication through the one sided API. The Direct API in Charm++ does not enforce that restriction by allowing any subset of the participating processes to exchange buffer information objects and use the one sided interface. Similarly, the completion detection scheme used by the Zerocopy API is different as compared to MPI. MPI RMA interface provides three schemes for synchronization: 1. Fence, 2. Post-start-complete-wait and 3. Lock/Unlock. The fence mechanism is a collective operation on all processors associated with an `MPI_Win` object. The Post-start-complete-wait scheme uses groups of processes to enforce synchronization. Finally, the Lock and Unlock model that involves only a single process, requires it to wait for the completion of the RMA operation. All the three schemes lead to unnecessary overhead and synchronization between the participating processes in order to notify the sender and receiver processes about the completion. In the Zerocopy Direct API, the completion detection is achieved using callbacks on both the sender and the receiver. Similarly, in the Zerocopy EM API, a callback informs the sender on completion of the RDMA operation and the execution of the entry method containing the `nocopy` parameters indicates the successful completion of all RDMA operations involved in that entry method.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we presented the Zerocopy API to reduce application memory footprint and accelerate large messages. We discuss the design and showcase two approaches to use the Zerocopy API. Firstly, with the Zerocopy Entry Method API that makes it easier for existing Charm++ applications to switch to the Zerocopy API with minor modifications. Secondly, with the Direct API that is more suited for iterative applications that have communication patterns that are persistent across iteration boundaries.

With the Zerocopy API, applications can avoid both sender side and receiver side allocation and copies, reducing their total memory footprint by 50%. In addition to the benefit of reduced memory footprint, our results indicate an improvement of 2 - 90 % in message transfer times using the Zerocopy API for large messages on different machines and interconnects.

6.2 Future Work

For networks requiring explicit registration, the performance can be further improved by minimizing the the expensive memory registration and de-registration operations that interact with the `NIC`. In future, we plan to implement a memory registration cache that can minimize these operations.

Currently, as the Zerocopy API is only supported for point to point messages in Charm++, we also plan to work on optimized versions of the Zerocopy API for collective operations like broadcast, scatter, gather etc. This work can also contribute to improving the startup time of Charm++ applications that use large sized readonly variables.

Since this work is still in early development and research, Adaptive MPI (AMPI)[7] is currently the only use-case for this API. In this regard, we plan to adopt the Zerocopy API in bigger applications that use large messages like OpenAtom and study its performance and scalability.

REFERENCES

[1] L. V. Kale and A. Bhatele, Eds., *Parallel Science and Engineering Applications: The Charm++ Approach.* Taylor & Francis Group, CRC Press, Nov. 2013.

[2] S. Kumar, Y. Sun, and L. V. Kale, "Acceleration of an asynchronous message driven programming paradigm on ibm blue gene/q," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, May 2013.

[3] Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. Olson, "A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[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*, Aug 2015, pp. 34–39.

[5] C. The MPI Forum, "Mpi: A message passing interface," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993. [Online]. Available: http://doi.acm.org/10.1145/169627.169855 pp. 878–883.

[6] E. Bohm, S. Chakravorty, P. Jetley, A. Bhatele, and L. V. Kale, "CkDirect: Unsynchronized One-Sided Communication in a Message-Driven Paradigm ," in *Proceedings of International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, August 2009.

[7] 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.