An OpenSHMEM Runtime for UPC

By

Daniel Lewis

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Middle Tennessee State University

May 2020

ACKNOWLEDGEMENTS

I would like to thank Dr. Aderholdt for his advice and guidance on this thesis. I would also like to thank Dr. Phillips and Dr. Barbosa for their support. Finally, I wish to thank Roger and Norma Lewis and Heather Moulder for their encouragement and patience.

Results presented in this paper were obtained using the Chameleon Cloud testbed supported by the National Science Foundation.

ABSTRACT

Partitioned Global Address Space (PGAS) languages are becoming more important as we move into the Exascale era. The complexities brought on by the architectures of these machines make traditional message passing approaches challenging. Currently, we lack a portable and flexible runtime for PGAS languages. The suggestion is that this is possible with an OpenSHMEM-based runtime.

OpenSHMEM is a PGAS library for the C language aiming to provide a standard Application Programming Interface (API) for Symmetric Hierarchical MEMory (SHMEM). OpenSHMEM is a portable, flexible, and performant API for PGAS runtimes through its use of thread safety semantics and grouping of processes through teams. This allows OpenSHMEM to make performance-oriented decisions on behalf of the language or library being implemented.

This thesis demonstrates a mapping and implementation of a Unified Parallel C (UPC) runtime based on OpenSHMEM, and an evaluation our runtime's performance in comparison to the popular Berkeley UPC implementation. Berkeley UPC uses a runtime based on Global-Address Space Networking (GASNet), a language independent middleware for PGAS programming languages.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I
INTRODUCTION

Partitioned Global Address Space (PGAS) is a parallel processing programming model in which the global address space is partitioned by process. In this programming model, each process has affinity to a portion of the shared space, and can do remote memory access (RMA) to and from the global address space. RMA allows processes within the parallel job to read from or write to the shared portion of memory of other processes within the job. PGAS programming models are emerging as an alternative or complement to Message Passing Interface (MPI) for programs with irregular message patterns [11]. This is due to the benefits of programming with one-sided communication over two-sided communication in programs that have data-dependencies and dynamic communication patterns [29]. In a PGAS programming model, the user can exploit data locality allowing for less network communication and use familiar and flexible shared memory semantics. This is enabled by the memory being logically partitioned such that each processing element (PE) has affinity to a portion of the global address space.

PGAS language runtimes implement the execution model of a PGAS language. Several PGAS languages and library runtimes are implemented using the language independent middleware Global-Address Space Networking (GASNet). GASNet clients include Stanford's Legion [15], Cray's Chapel [7], Rice University's Co-Array Fortran [20], and Berkeley UPC [2]. PGAS runtimes may also leverage high-performance communication libraries such as Unified Communication X (UCX). UCX has support from several hardware vendors including Mellanox and ARM [24]. Prior work has shown that the higher level library OpenSHMEM can serve as a runtime for Co-Array Fortran, in place of a GASNet-based runtime, with a positive performance impact [17]. This suggests that OpenSHMEM can serve as a general high-performance PGAS runtime.

OpenSHMEM is an open specification of a PGAS library that provides an Application

Programming Interface (API) for Symmetric Hierarchical MEMory [22]. This specification, allows for greater portability of OpenSHMEM applications. The use of OpenSHMEM as a runtime affords more portability, flexibility, and ease of programming for PGAS languages over lower-level libraries and frameworks. OpenSHMEM is already widely distributed on the types of systems targeted by PGAS languages and libraries, because it comes with the popular MPI implementations OpenMPI and MVAPICH2-X MPI [23], [18], [16]. An OpenSHMEM runtime also allows for non-standard extensions to PGAS languages and libraries and makes hybrid programming techniques available.

There are several PGAS languages and libraries. One of the most common PGAS languages is Unified Parallel C (UPC) [25]. UPC is an extension of the C language. This means, in general, the compiler can be more portable as it generates regular ANSI C code which is then compiled by the system's C compiler. In order to facilitate our evaluations of OpenSHMEM as a PGAS runtime, we have created a mapping and runtime layer implementation of UPC.

While OpenSHMEM is a flexible API, not all of the UPC language specification maps neatly to OpenSHMEM library calls. This is especially true in regards to memory management. All dynamic shared memory in OpenSHMEM is allocated on the symmetric heap. The symmetric heap is a portion of shared memory which every PE allocates that allows remote memory access (RMA). This means all nodes must allocate shared memory collectively and in the same amounts. It is possible to read and write these locations in memory from other processes in the same job. In UPC, shared memory is not always symmetric or collectively allocated. For instance, in UPC it is possible to allocate shared memory on only one process. This is not possible in OpenSHMEM as all shared memory must be symmetric across all PEs at all times. Therefore, new strategies are needed to navigate these mismatches with limited impacts on performance.

The use of an OpenSHMEM runtime also opens up some opportunities with regard to hybrid programming techniques. For example, a UPC runtime might be implemented in a manner that makes it possible to make OpenSHMEM library calls from UPC programs. This would give the programmer the ability to perform operations present in OpenSHMEM but not in UPC such as the creation and use of teams. Also, calls to OpenMP might be used to increase the performance of certain aspects of the runtime.

In this thesis, we use UPC as a case study for OpenSHMEM as a runtime for language based PGAS programming models.

Our contributions are as follows:

1. We analyze UPC and develop a mapping of compatibility between interfaces of UPC and OpenSHMEM in Chapter IV.

2. We implement a prototype UPC runtime with OpenSHMEM in Chapter V.

3. We evaluate our prototype on a mixture of micro-benchmarks and applications in comparison with Berkeley UPC in Chapter VI.

This thesis is organized as follows: In Chapter II, we discuss the background material of this thesis, in particular execution models, Unified Parallel C, and OpenSHMEM. Chapter III covers how related work motivates our design. We discuss our design of an OpenSHMEM runtime for UPC in Chapter IV. In Chapter V, we cover some of the important details of our implementation. We evaluate the performance of our UPC runtime using several benchmarks and applications in Chapter VI. In Chapter VII, we discuss the results of our evaluations, draw conclusions and explore avenues that could be investigated in the future.

CHAPTER II

BACKGROUND

In this section we discuss the PGAS language and PGAS library involved in our project. We discuss both the UPC language for which we implemented a runtime, and OpenSHMEM which is the PGAS library used in our runtime implementation.

<u>Execution Model</u>

UPC and OpenSHMEM both follow a single process multiple data (SPMD) execution model illustrated in Figure 1. In this execution model, each process has the same source and are created at the beginning of the job and continue until execution is complete. The processes communicate via a communication layer.

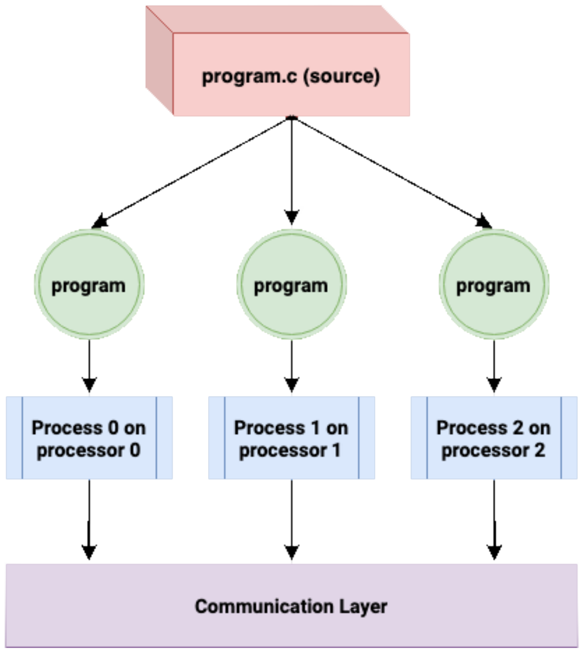

Figure 1: The Single Process Multiple Data model in which each process has the same source, and all processes communicate via a communication layer

Overview of Unified Parallel C

UPC is a superset of the C language aimed at parallel computing [25]. UPC programs follow the PGAS programming model. In the PGAS programming model, every process has access to distributed shared memory in a global address space. The address space is logically partitioned into two categories. The first is private memory that only the owner process can access, and the second is shared memory that is remotely accessible through RMA operations. This is demonstrated in Figure 2. The type of memory allocated is selected by the user using a UPC-based memory allocation function or by declaring a variable with the *shared* keyword. RMA operations are either implicit by changing the value of an element in a variable declared as *shared* or explicit by using a PUT or GET operation.

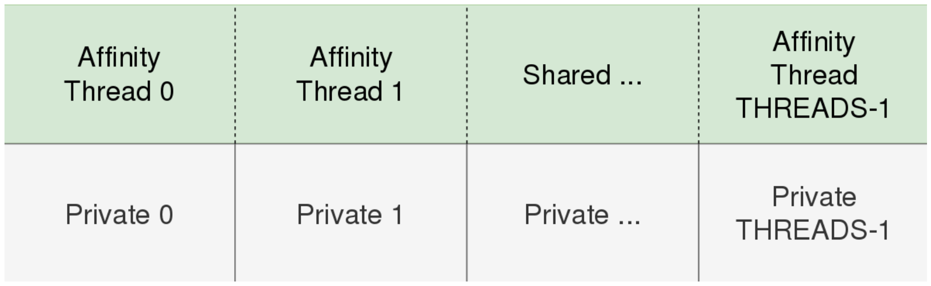

Figure 2: The memory model of UPC. Every UPC thread has a private and shared section of memory.

Compiling UPC programs is a multi-stage process involving translation to ANSI C and compilation of the C source code. This process is shown in Figure 3. UPC programs are first translated from UPC into ANSI C programs with calls to the UPC Runtime (UPCR) interface. This process is shown in Figure 4. UPCR defines an interface between the platform independent ANSI C generated by the translator and the UPC runtime layer [5]. The runtime layer implements the UPC language for a specific architecture. This separation allows for UPC to be implemented on various platforms without the reimplementation of the translator. Because the translation is source to source, translation is not specific to the target architecture. This means translation can happen remotely over http [3].

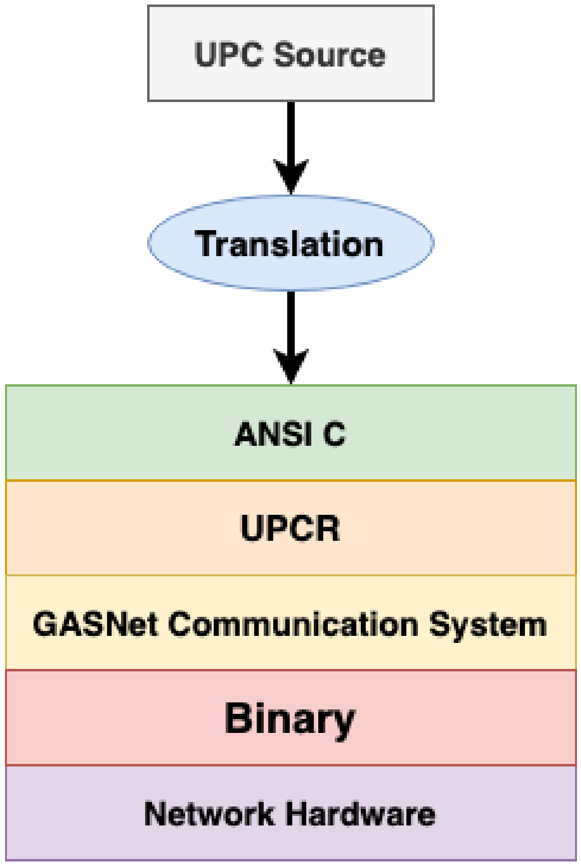

Figure 3: This figure shows the layers of the Berkeley UPC system

As stated above, the execution model of UPC follows a SPMD model. Every process, called threads, operate on a single global address space. Threads in UPC are not the same as operating system threads such as POSIX threads, but in some implementations they can sometimes be implemented as such. To avoid potential confusion we will refer explicitly to the type of thread (UPC thread, POSIX thread or OpenMP thread) in the rest of this document.

Synchronization is managed by using locks, barriers, and fences. Locks in UPC are an opaque datatype. Locks are allocated, locked, unlocked, and freed only via functions the language provides. Manipulating locks in any other fashion is undefined behavior. When locks are allocated, communication between the calling thread and thread 0 is required,

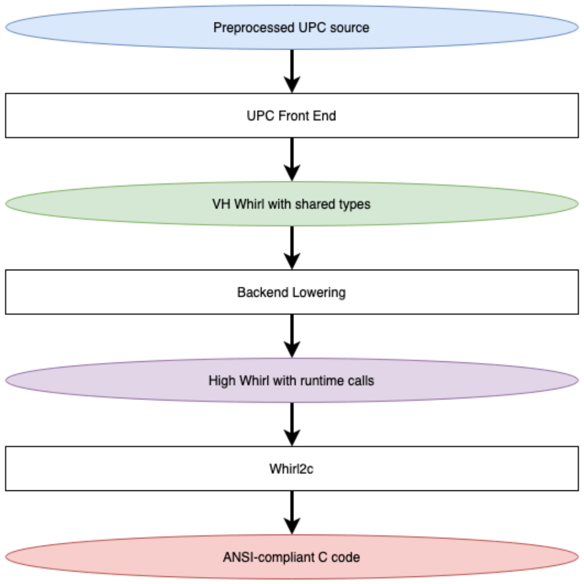

Figure 4: This figure shows the Berkeley UPC translation process.

because all locks must be allocated by thread 0. This negatively affects scalability in UPC, because this can be a bottleneck as UPC thread counts increase.

Barriers in UPC are collective. UPC includes split-phase barriers using the *upc_notify* and *upc_wait* statements. These statements take an optional expression. If this expression is used, barriers with the same expression or no expression will be matched with the barrier. Split-phase barriers help to reduce the amount of time threads spend blocked on synchronization. This works by setting a synchronization point with *upc_notify*. After threads reach this synchronization point they can continue to perform computation that does not depend on the synchronization. The threads will then call *upc_wait* and block until all threads have reached the synchronization point. This type of synchronization is illustrated in Figure 5.

Memory consistency in UPC is controlled by the user on a per variable, statement, or type basis [25]. The user can do this with the *strict* and *relaxed* keywords. In strict mode, shared operations happen in order, and the program executes in a sequential manner. This

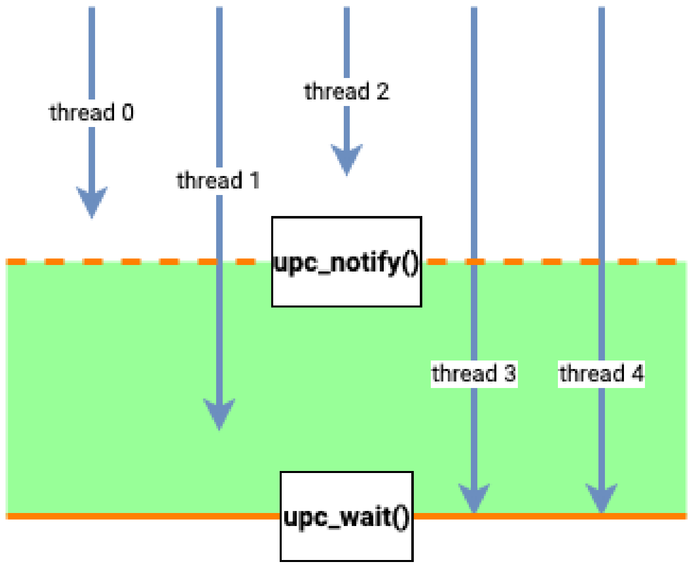

Figure 5: Execution path of a program using a split phase barrier. Threads 3 and 4 wait at upc_wait until 0 and 2 arrive at upc_notify. The green section indicates where computation that does not depend on synchronization is performed.

means communications will arrive in the order they are sent. In relaxed mode this is not the case. Shared operations may be reordered by the compiler or runtime, and the operations may appear out of order to other UPC threads.

UPC programmers can also control the data layout of variables. One way this can be done is with the *shared* keyword [25]. The *shared* keyword is a way the programmer can allocate an array in the shared space. By default, the affinity of elements in shared arrays are distributed in a round robin fashion on a per element basis. The distribution can be adjusted by the user during allocation, and affinity can be distributed in a cyclic (Figure 6), blocked cyclic (Figure 7), or blocked (Figure 8) arrangement depending on what is requested by the user. These arrays can be allocated in a collective fashion in which all threads participate and allocate some amount of memory, or in a non-collective fashion in which only one thread participates and allocates memory. This does not affect the layout of the data.

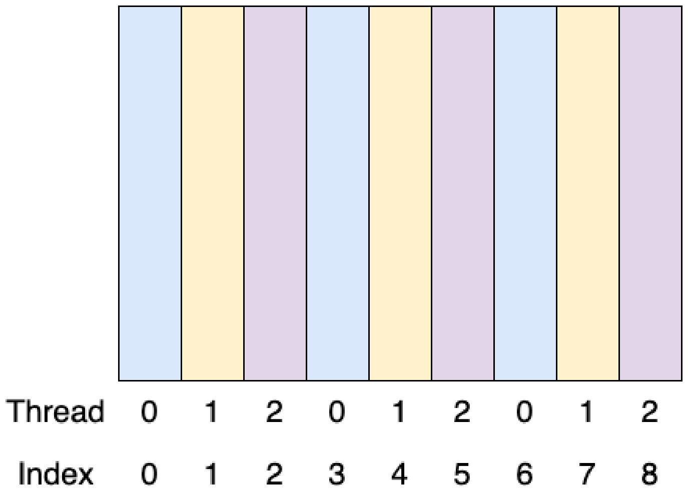

Figure 6: Cyclic layout for a shared array.

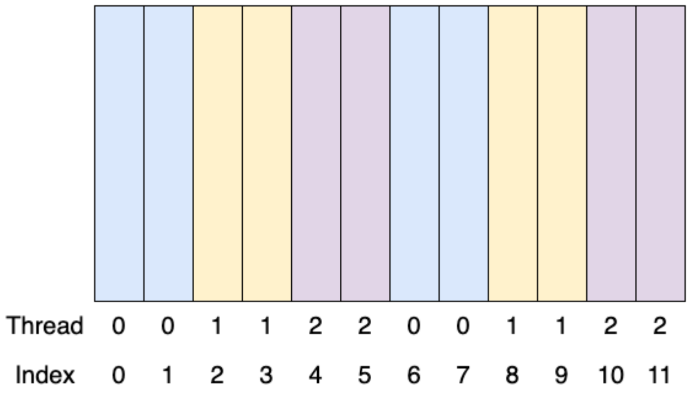

Figure 7: Blocked layout for a shared array.

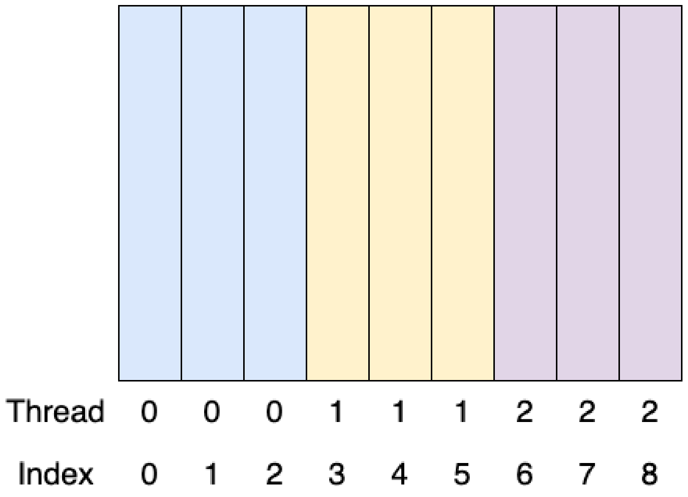

Figure 8: Blocked layout for a shared array.

Overview of OpenSHMEM

OpenSHMEM is a specification of a library based approach to a PGAS programming model [22]. There are several compliant OpenSHMEM implementations [23]. OpenSHMEM has a distinction between memory inside a global address space and private memory. In OpenSHMEM, all memory in the global address space must be symmetrically allocated across PEs (Figure 9). OpenSHMEM provides interfaces to perform RMA operations across all PEs. Unlike some libraries for parallel communication, such as MPI, there is no implicit synchronization. All synchronization must be explicit.

Like UPC, OpenSHMEM follows an SPMD execution model where each PE is launched during initialization and lasts the duration of the entire job. PEs can be synchronized using the locks, barriers, and fences provided by the library. Otherwise, they behave asynchronously, and may take different execution paths. In addition, OpenSHMEM also provides atomic and collective operations as a convenience to the programmer.

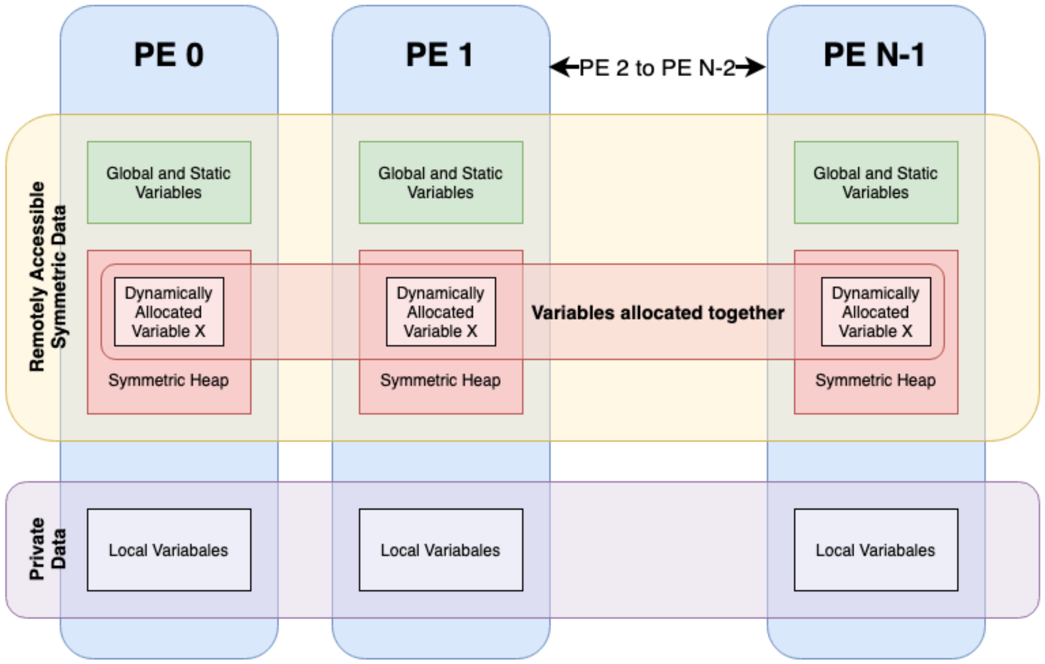
PE 0
PE 1
PE 2 to PE N-2
PE N-1
Remotely Accessible Symmetric Data
Global and Static Variables
Dynamically Allocated Variable X
Variables allocated together
Symmetric Heap
Private Data
Local Variabales

Figure 9: OpenSHMEM Memory Model

CHAPTER III

Related Work

In this section we will discuss work related to this thesis. In particular we will look the PGAS programming model, UPC runtime implementations, and OpenSHMEM's ability to act as a runtime layer.

Baker et al. describe in this conference paper the performance benefits of OpenSHMEM over MPI-3 when implementing the Smith-Waterman algorithm [1]. This shows the benefit of a PGAS programming model when implementing an algorithm which requires non-uniform access patterns. It is also a demonstration of the performance benefits of the implementations of the OpenSHMEM API over the implementations of MPI-3 one-sided communications. Similarly, Grossman et al. found that Graph500 implementations based on OpenSHMEM scaled more readily and were more programmable than the available MPI-based implementations [9]. This was due to the irregular patterns of the Graph500 benchmark that suit the PGAS programming model, and OpenSHMEM's fine-grain communication support for high-performance interconnects.

Over the years several UPC runtimes have been proposed. Zhang et al. present an MPI and POSIX thread based UPC runtime [30]. They used a two-sided communication and a threaded approach. Each UPC thread is implemented with two POSIX threads. One thread performs computation while the other performs all communication. The use of lightweight threads within UPC threads in an OpenSHMEM based design can be found in Chapter IV. In this paper they also discuss the value of and implement atomic operations for UPC.

GNU UPC uses a Portals 4 based runtime [8]. They found that Portals' atomic operations were useful in implementing UPC collective operations, but for some collectives such as *upc_all_prefix_reduce<<TYPE >>()* they simply use the MPI implementation. Because OpenSHMEM has support for teams, contexts, atomic operations and collectives, our approach to collectives is similar as can be seen in Chapter IV.

The OpenSHMEM API can be implemented with various communication libraries which may leverage many different types of transports. The interface makes very few assumptions about how the underlying communication is accomplished. This flexibility increases portability, as there is a separation between the networking hardware and the OpenSHMEM API. This means that as underlying communication libraries change it is less likely that runtimes based on the OpenSHMEM API will need to make drastic changes.

One concern when designing OpenSHMEM based PGAS runtimes is the amount of overhead it will add to the communication layer. It is important that the OpenSHMEM API does not add too much overhead in terms of performance. Coti et al. suggest that OpenSHMEM implementations can be lightweight, and can add very minimal overhead to the communication layer [6].

Namashivayam et al. demonstrate an OpenSHMEM runtime as a replacement for GASNet with Co-Array Fortran [17]. This paper showed promising results that support the idea of OpenSHMEM as a unified runtime for PGAS languages. This paper also contains additional latency tests demonstrating various OpenSHMEM implementations' performance in comparison to GASNet and MPI. GASNet and OpenSHMEM both outperformed MPI-3.0 implementations in one-sided communication tests. OpenSHMEM outperformed or matched GASNet for large message sizes or when there is network contention. OpenSHMEM was also aided by the inclusion of remote atomic operations. These tests suggest that OpenSHMEM is a promising candidate as a performant PGAS runtime, but more testing should be performed to support this idea.

CHAPTER IV

Design of a UPC Runtime

Many of the functions and keywords in the UPC language map easily to functions available in the OpenSHMEM library. These UPC functions and keywords can be mapped directly to OpenSHMEM. Other parts of the UPC language are more difficult to map to OpenSHMEM and require more complex solutions. One example is the split phase barrier and optional barrier expressions. These types of barrier semantics are not present in OpenSHMEM. OpenSHMEM only has a strided collective barrier over an active set. An active set is a grouping of PEs defined by the starting PE, a $\log_2$ stride between the included PEs, and an ending PE. Another important example is memory management. UPC does not maintain symmetry between allocations of shared memory, but this is a requirement in OpenSHMEM. In this chapter we will describe how the UPC runtime was designed using OpenSHMEM.

UPC Types

One of the additions UPC makes to the C language is the inclusion of shared and shared pointer types. These additional types allow for RMA. Internally, these types need to carry additional information a primitive type does not carry, such as the UPC thread to which the pointer points. One way to have a type that can contain all of this information is to segment a register sized primitive type, and use groups of the bits to contain the information that is required. This method would ensure that the type can fit in a register. This could have performance benefits, but makes trade-offs about the maximum sizes of UPC thread counts, address size, and array size based on the size of the register. For this reason, we chose to design our pointers as the complex type shown in Figure 10. This method will scale better as clusters, node counts, and address space increase.

```
struct upcr_pshared_ptr_S {
    char *addr;
    char *loc_addr;
    size_t thread;
    size_t blocksize;
    size_t pos;
    size_t elemsz;
    size_t phase;
} typedef upcr_pshared_ptr_t;
```

Figure 10: Internal pointer to shared type

RMA

UPC defines functions for RMA - *upc_memput()* that copies a number of characters from the calling UPC thread to a remote UPC thread and *upc_memget()* that copies a number of characters from a remote UPC thread to the calling UPC thread. These functions map nicely onto the OpenSHMEM functions *shmem_put()* and *shmem_get()*. These types of functions are common in many PGAS languages, and are required functionality for UPC. One area where UPC offers functionality that is not directly present in OpenSHMEM is *upc_memcpy()*. This function can copy memory between two remote PEs. In our design, we copy the data through a buffer on the calling thread if both the source and destination threads are not the calling thread. This is shown in Figure 11.

Memory Management

UPC and OpenSHMEM diverge significantly when it comes to memory management. OpenSHMEM strictly maintains a symmetric heap, and all allocations must be made collectively. In contrast, UPC allows one thread to non-collectively allocate shared memory on all UPC threads. This makes a direct mapping from OpenSHMEM memory management to UPC memory management impossible.

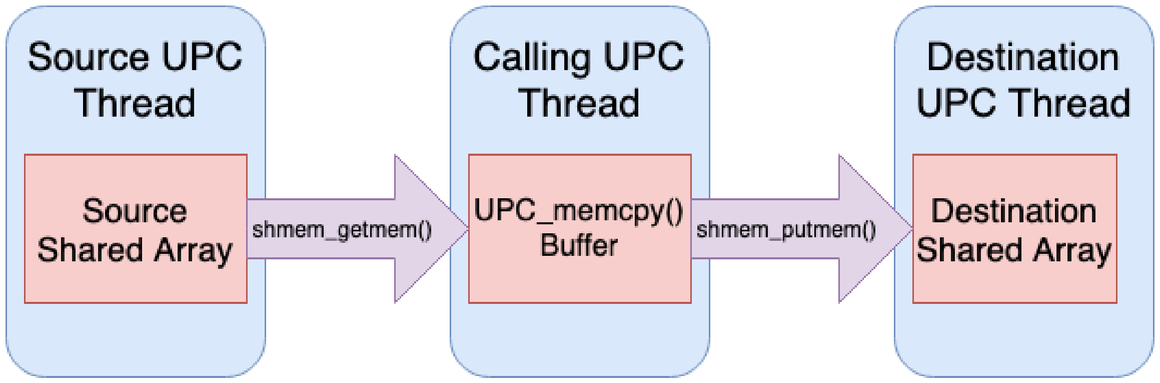

Figure 11: Design of *upc_memcpy()*. The UPC thread calling *upc_memcpy()* does a *shmem_getmem()* from the source to a local shared buffer and then must do a *shmem_putmem()* to the destination UPC thread. This is only required when both the source and destination are not the calling thread.

To work around this fact, our design partitions our symmetric memory into multiple sections. On each UPC thread there is local shared memory, global shared memory and symmetric shared memory. Local shared memory is allocated in a symmetric buffer created during initialization. Allocations for local shared variables made by *upc_alloc()* can occur in this buffer. Likewise, global shared memory is allocated in a separate symmetric buffer, and allocations made by *upc_global_alloc()* happen inside this global buffer. Having separate buffers for these types of allocations means that memory is always symmetric across all UPC threads in the global buffer. In order to facilitate the allocations within these buffers we included a separate runtime level memory allocator.

This memory allocator helps us manage the memory within these buffers. The objects which hold the state for this allocator are allocated on the symmetric heap during initialization. Global memory is distributed, but the state of global shared memory is located on UPC thread 0. By doing RMA operations to UPC thread 0, one UPC thread can allocate memory on all other UPC threads asynchronously.

Synchronization

UPC and OpenSHMEM also have semantic differences in synchronization routines which prohibit a direct mapping from OpenSHMEM to UPC. These differences occur both in the locking and barrier routines of UPC and OpenSHMEM.

The locking routines in OpenSHMEM require that the lock be a symmetric data object of type long in the C language. In UPC there is no requirement that the lock be collectively or symmetrically allocated. This means that in our UPC runtime design we must ensure that all locks are symmetric objects while not enforcing this requirement on the UPC programmer. This is made difficult by the fact that *upc_global_lock_alloc()* is not collective. Our design accomplishes this via allocating the locks on the global shared heap. This means that locks are allocated on every UPC thread. By allocating our locks in this way we ensure that the OpenSHMEM requirement that all locks are allocated symmetrically is met while not enforcing a collective allocation.

UPC defines a split phase barrier which does not have an analogous OpenSHMEM function. This type of barrier is in 2 parts. The first is *upc_notify*. The second is *upc_wait*. We have designed these two functions in terms of *shmem_wait_until()* which blocks a thread until some condition has been reached. In order to notify, every PE involved must do an atomic ADD to every other thread's barrier count. As was mentioned in Chapter II atomic operations are a beneficial feature when implementing PGAS runtimes. The wait is accomplished by calling *shmem_wait_until()* which blocks the thread until the local barrier count is equal to the number of UPC threads. This allows us to support the optional expression and only block the appropriate UPC threads. This would not be possible using OpenSHMEM barrier semantics which are not split phase and can only operate on a strided subset of PEs which constitute the active set.

OpenSHMEM has a strided barrier function over an active set. UPC barriers are collective statements. These statements are translated into UPCR functions when using the Berkeley UPC translator. UPC's barrier statement is equivalent to a strict null access (an RMA access with strict ordering that makes no change) and a *upc_notify* followed by a *upc_wait*. For performance reasons, we did not design our runtime this way when no expression is present. Instead in the case of a *upc_barrier* statement without the optional expression we call *shmem_barrier()*.

Collectives Library

UPC and OpenSHMEM handle collectives in similar ways, and in several cases this makes a mapping from OpenSHMEM to UPC little more than rearranging the arguments to the functions. For example, broadcasts function in a very similar fashion in both the OpenSHMEM library and the UPC language. The *upc_all_broadcast* function copies a block of memory from a single UPC thread to a block of shared memory on every other UPC thread [26]. This maps directly from OpenSHMEM to UPC. This is an example of the features of OpenSHMEM benefiting it's use as a PGAS runtime as was mentioned in Chapter II. However, there are some mismatches particularly when it comes to reductions.

UPC library spec defines two separate semantics for reductions:

1. *upc_all_prefix_reduce<<TYPE>>* which maps simply to an OpenSHMEM reduction, because the destination and source arrays are equal in size.

2. *upc_all_reduce<<TYPE>>* which completely reduces an entire array across all threads down to a single element on thread 0.

In our design of *upc_all_reduce()*, each UPC thread performs the required operation across the portion of the array local to the same UPC thread and then a reduction is done on the local results of the parallel operation. This method does most of the computation in parallel, and minimizes communication.

UPC is also unlike OpenSHMEM in that OpenSHMEM provides separate reduce functions for all operations. UPC provides only one of each type of reduce function which takes an operation flag, and a datatype flag. The operation flag determines the type of operation to be completed on every data element across all participants, and the datatype flag tells the translator the UPCR function to which the call should be translated. In order to implement this in OpenSHMEM, every UPCR function generated must also decide based on the operation flag which OpenSHMEM call is required. In addition, not all of the types supported by UPC reduce have a direct mapping to an OpenSHMEM call and vice versa. For example, in OpenSHMEM there are is no character reduction. This means that some type coercion is also required, for instance, to do a reduction on a character array. In our implementation we cast characters to short integers resulting in character reductions being much larger than they would otherwise need to be. Reductions for larger types, such as integers, would not suffer from this effect as integer reductions are defined in both the UPC Collective Library and OpenSHMEM.

CHAPTER V

Implementation

GASNet to OpenSHMEM

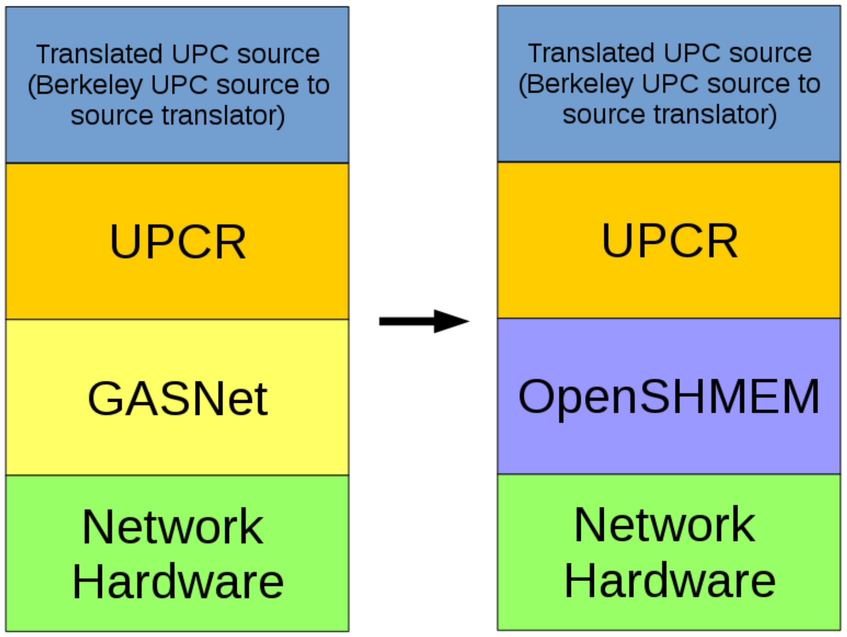
Translated UPC source (Berkeley UPC source to source translator)
UPCR
GASNet
Network Hardware
Translated UPC source (Berkeley UPC source to source translator)
UPCR
OpenSHMEM
Network Hardware

Figure 12: The layers of the Berkeley UPC runtime vs the layers of our OpenSHMEM based runtime

Our implementation is based on Berkeley UPC which is a layered design. Figure 12 shows the layer where we used the UPCR interface to create our UPC runtime. Our UPC runtime implementation mostly takes place at this layer, and does not need to make many changes at any higher layer. We do make some minor changes to the generated C source code. These changes mostly deal with removing GASNet specific macros and inserting our OpenSHMEM specific headers. This is done with a POSIX compliant shell script. This shell script acts as a wrapper for the translator and compilers involved. The user only needs to invoke it in the same way they would invoke any compiler. This will generate a binary with calls to the OpenSHMEM library.

Our UPCR functions and macros should work with any OpenSHMEM implementation. This, along with the fact that UPC translation is available over http, allows for a high degree of portability with our UPC runtime.

UPC Infrastructure

UPC, like OpenSHMEM, requires the execution of an initialization function. This means our implementation of UPC's initialization must call *shmem_init()*. Our UPC runtime also uses this initialization portion to setup data structures, call the UPCR startup functions, and allocate memory for runtime level buffers such as the buffer used by *upc_memcpy()*, local allocations, and global allocations.

CHAPTER VI

Experimental Evaluation

Methodology

In evaluating the performance of our runtime we utilized resources provided by Chameleon Cloud [14]. Chameleon Cloud is a experimental platform aimed at cloud computing research. Our test configuration consisted of Haswell nodes at the Texas Advanced Computing Center (TACC) with Intel Xeon 2.3 GHz processors with 48 hardware threads and 128GB of RAM. Each node had a Mellanox Technologies MT27500 Family ConnectX-3 network controller. Each node was running CentOS 7. For our standard UPC implementation we used Berkeley UPC version 2019.4.4, and for our OpenSHMEM implementations we used MVAPICH2-X 2.3rc2 and OpenMPI 4.0.1 for all our tests. In all of our tests, OpenMPI OpenSHMEM was using the Unified Communication-X communication layer [21], and MVAPICH2-X was using the Unified Communication Runtime (UCR). Both OpenSHMEM implementations used the XPMEM linux kernel module for shared memory communication[28]. XPMEM allows a process to map a memory region from another process into its own address space.

In all of our tests the UPC threads were distributed on the nodes in a round-robin fashion. When using the OpenSHMEM based UPC runtime nodes were bound to a socket in all tests except in our threaded reduction test. Binding to a socket means that each process will be scheduled to a hardware thread on a particular socket.

For our tests on UPC we used the OSU Micro-Benchmark Suite 5.6.2. These benchmarks are designed to test the performance of message passing and PGAS languages and libraries [19]. We intend to show that OpenSHMEM can adequately act as a runtime for UPC without increasing latency and in several cases can reduce latency when compared to GASNet. We also show example runs from several sample programs that are packaged with Berkeley UPC to demonstrate correctness and performance with multiple UPC applications.

OSU Micro-Benchmarks

We tested the PUT latency using the OSU UPC Put Latency Micro-Benchmark with a single pair of processes on 2 nodes. When using the OpenSHMEM based UPC runtime the UPC threads were bound to a socket. This is not possible when using Berkeley UPC. This setup tests the latency over the network and not the latency involved in shared memory operations between two processes on the same node. The test performs 10000 iterations of each size PUT operation. By default the largest size tested is 4MB. Each PUT operation must be complete before the next one can be issued. Our runtime shows larger performance

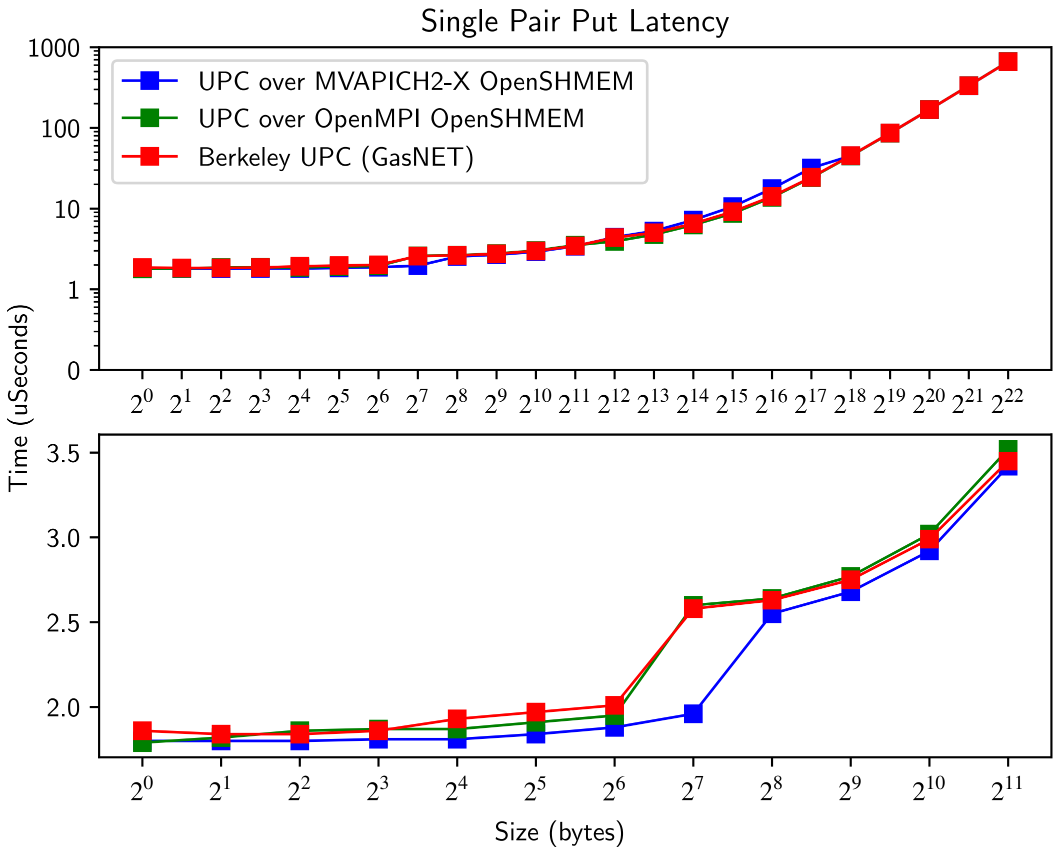

Figure 13: UPC PUT single pair PUT latency. This test measures the latency for *upc_memput()*. Lower is better.

gains with smaller blocks of memory. For small PUT operations 2kB or less our runtime is faster when using MVAPICH2-X OpenSHMEM. With MVAPICH2-X OpenSHMEM and OpenMPI OpenSHMEM we see a percent decrease in latency of 3.23% and 3.76% between our runtime and Berkeley UPC respectively for single byte PUT operations. As we can see in Figure 13, the differences in PUT latency between the OpenSHMEM and GASNet based runtimes are not as significant at the largest default tested size of 4MB. Because the differences between runtimes become very small at large sizes, we did not test any larger sizes. This test suggests that for smaller PUT operations an OpenSHMEM based UPC runtime could have performance benefits.

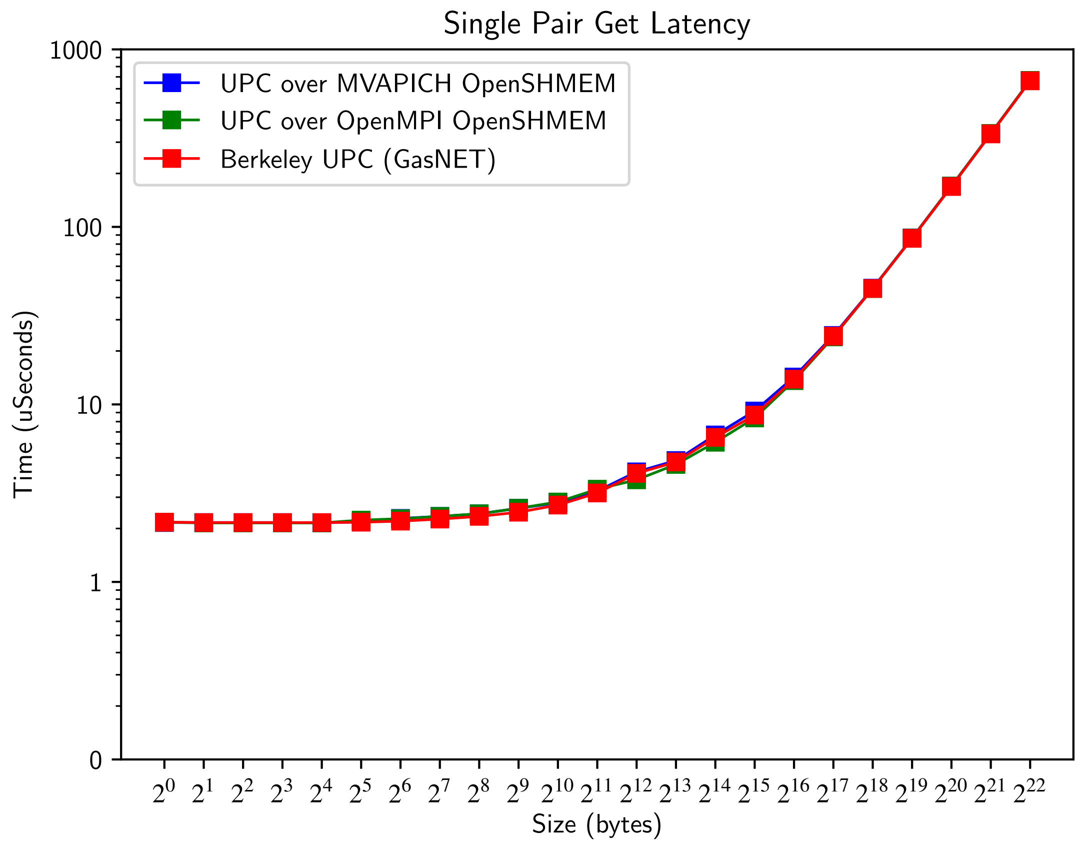

Figure 14: UPC GET single pair GET latency. This test measures the latency for *upc_memget()*. Lower is better.

To test the GET latency, we used the same experimental setup as the above PUT latency test and the OSU UPC Get Latency Micro-Benchmark. For a single byte GET, our runtime using MVAPICH2-X OpenSHMEM saw a latency percentage decrease of 0.46%. We saw no measurable difference in latency between our runtime with OpenMPI OpenSHMEM and Berkeley UPC for a single byte GET. For larger GET operations of 4MB bytes our runtime using MVAPICH2-X OpenSHMEM increases latency by 0.51% compared to Berkeley UPC. Our runtime using OpenMPI OpenSHMEM increases latency over Berkeley UPC by 0.61% for 4MB GET operations. This is shown in Figure 14. Our runtime increases latency by 0.03% on average with OpenMPI OpenSHMEM and 1.73% on average for MVAPICH2-X OpenSHMEM. The differences between our UPC runtime and Berkeley UPC are not significant, and can be attributed to noise on the network.

In PGAS languages, such as UPC, explicit synchronization such as barriers are required. Since, barriers are important to this programming model, we also tested the barrier latency. To do this we used the OSU UPC Barrier Latency Micro-Benchmark. We tested with several UPC thread counts in order to test how the barrier implementations would scale. Each UPC thread was distributed in a round-robin fashion. When using the OpenSHMEM based runtime UPC threads were bound to socket. As can be seen in Figure 15, we did not see equivalent performance. With only 2 threads we see large differences in performance. With only 2 threads our runtime using OpenMPI OpenSHMEM decreases latency by 56.09% while our runtime using MVAPICH2-X increases latency by 61.00%. When we increase the number of threads to 128 we see a latency increase from our runtime using OpenMPI OpenSHMEM of 3997.65%, but when using MVAPICH2-X OpenSHMEM we see an latency increase of only 10.59%. This discrepancy in performance between implementations of OpenSHMEM is likely due to MVAPICH2-X and OpenMPI using different barrier algorithms. MVAPICH2-X uses a node aware dissemination algorithm similar to what Hensgen et al. describe in [12]. While OpenMPI use the pairwise exchange with recursive

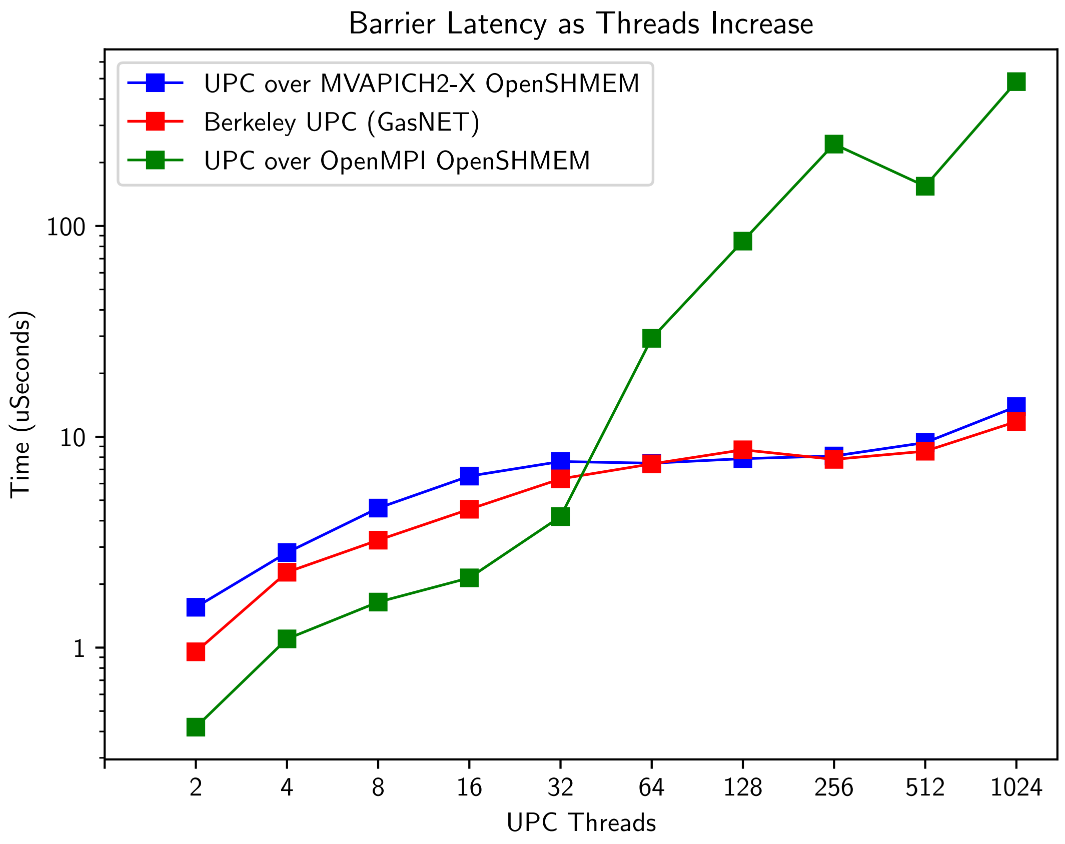

Figure 15: UPC barrier latency. This test measures the latency for *upc_barrier* statements for UPC thread counts 2-1024. Lower is better.

doubling technique described by Gupta et al. by default [10]. Both algorithms are O(NP) in complexity but the communication patterns differ.

We also implemented functions included in the UPC Collectives Utilities library. The UPC collectives library is a required library in the UPC Specification version 1.3. OpenSHMEM defines many similar collective functions such as broadcasts. However, not all UPC collective functions have a trivial mapping to OpenSHMEM collectives. In general, performance is best when the mapping from OpenSHMEM to UPC is simple. A simple mapping keeps the runtime overhead low.

We implemented and tested the *upc_all_broadcast()* function. This function copies a block of memory from a single UPC thread to all other UPC threads. This is a one to all operation, and it maps easily to OpenSHMEM's broadcast function. We tested the performance of our implementation with 1024 threads on 23 nodes. The UPC threads were distributed in a round-robin fashion. When using the OpenSHMEM based runtime, UPC threads were bound to socket. As can be seen in Figure 16, at a broadcast size of 1 byte we show a 98.17% decrease in latency when using MVAPICH2-X OpenSHMEM and with OpenMPI OpenSHMEM we show 61.49% decrease. This may be due to MVAPICH2-X's

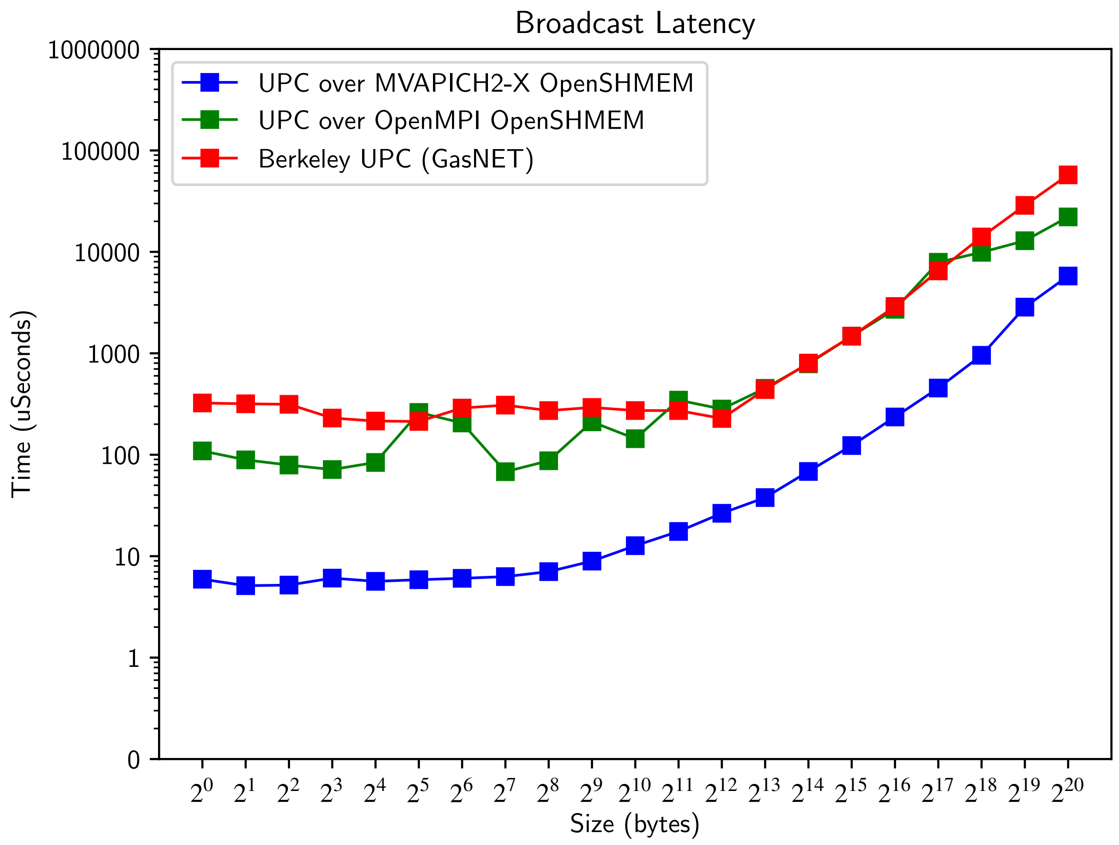

Figure 16: This test measures the latency of *upc_all_broadcast()* function calls as shared array size increases for UPC thread counts of 1024. Lower is better.

algorithm more effectively exploiting shared memory communication between UPC threads on the same node.

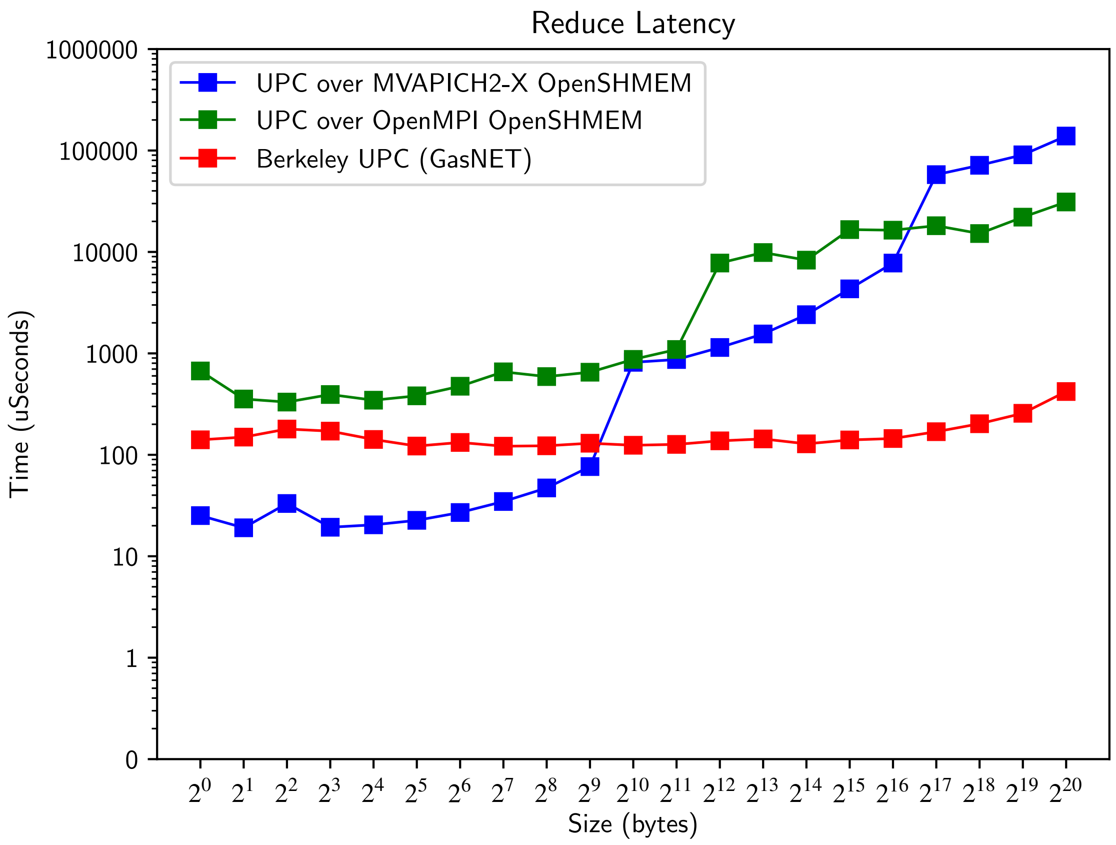

Figure 17: This test measures the latency of *upc_all_reduce<<TYPE >>()* function calls as shared array size increases for UPC thread counts of 1024. Lower is better.

We also tested the *upc_all_reduce<<TYPE >>()* function. This function reduces all elements in a shared array down to one shared element. We used the test from the OSU Micro-Benchmarks to test the performance of our UPC runtime. Our test was ran with 1024 UPC threads distributed across the nodes in a round-robin fashion. When using the OpenSHMEM based runtime, UPC threads were bound to socket. As can be seen in Figure 17, our UPC runtime performs best with smaller reductions. This seems to result from the mismatch between OpenSHMEM's reduction and the UPC reduction that is tested. This mismatch means that our runtime must iteratively perform the operation requested on the local portion of the shared array. The result of this operation is reduced using *shmem_reduce()*. The number of times each thread must do the requested operation increases by the size of the shared array divided by the number of UPC threads. Our runtime using MVAPICH2-X OpenSHMEM performs better than Berkeley UPC for shared arrays smaller than 512 bytes.

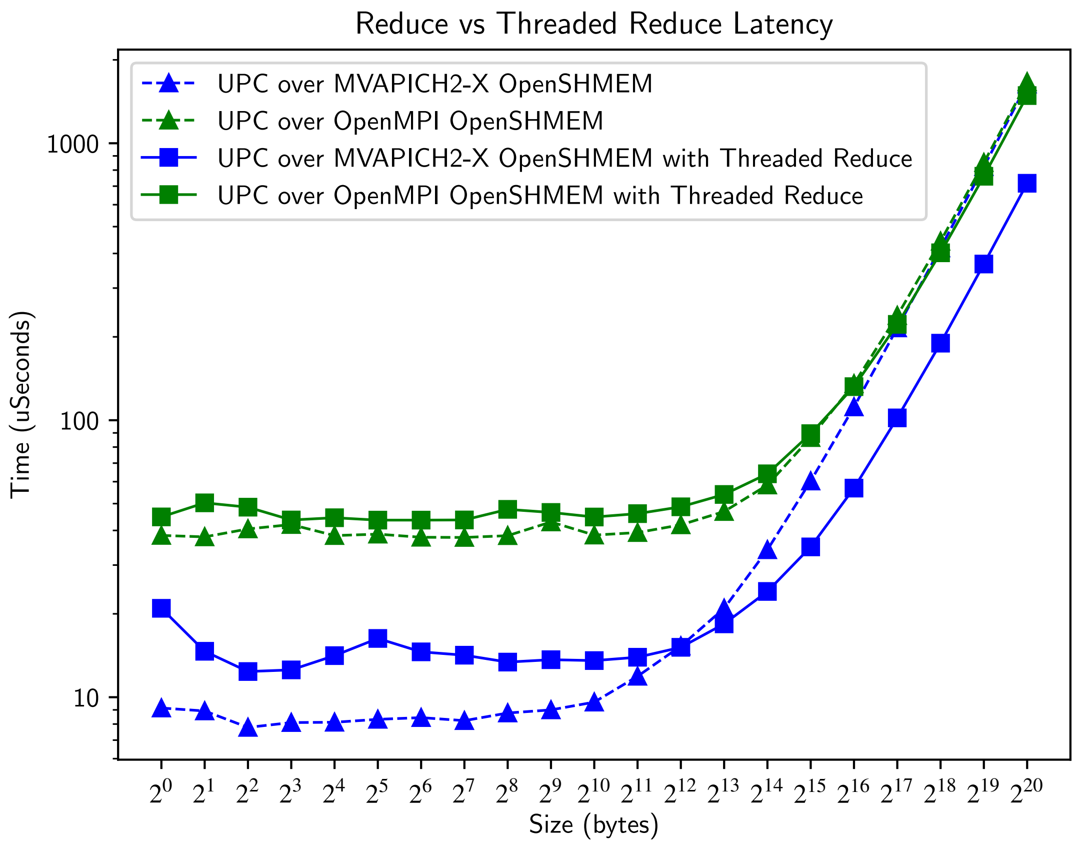

Figure 18: This test measures the latency of upc_all_reduce<<TYPE >>() threaded function calls as the size of the shared array increases for UPC thread counts of 128 on 6 nodes. Lower is better.

The OpenSHMEM implementations we used support the use of OpenMP threads. This allowed us to implement a threaded version of our *upc_all_reduce<<TYPE >>()* function. This threaded version decreases the time needed to do the local operations required in the reduction. We modified the OSU Reduce Latency Test to do a SUM instead of a MAX operation, because a MAX is not as easily threaded in this way. For this test we did not bind UPC threads to socket when using the OpenSHMEM based runtime, because this degrades performance when using OpenMP threads. We ran our test of the threaded reduce on 6 nodes each of which have 48 hardware threads. This gives us a total of 288 hardware threads. It is important that there are enough hardware threads on each node to support both the OpenMP threads and UPC threads or performance will degrade. We ran this test with 128 UPC threads and 2 OpenMP threads per UPC thread which used slightly less than the total number of hardware threads available. There is some performance penalty due to the overhead of launching the OpenMP threads. For MVAPICH2-X OpenSHMEM we begin to see performance benefits at 4kB. Using this threaded approach for a 1MB reduction and 128 UPC threads, we saw a 56.16% decrease in latency with our runtime using MVAPICH2-X OpenSHMEM from the single threaded approach and a 10.75% decrease in latency with OpenMPI OpenSHMEM from the single threaded approach.

As can be seen from our PUT and GET latency tests our OpenSHMEM based runtime does not add overhead with respect to fundamental communication. In addition, the ability to leverage collective operations further increases performance. Collectives operations for which we were able to most directly leverage OpenSHMEM collectives showed the best performance. By using OpenSHMEM implementations' support for OpenMP threads, we were able to improve the performance of reductions over the single threaded approach for reductions over 4kB.

Evaluation of UPC Applications

We also show that our runtime can run unmodified UPC programs. We do this in order to show correctness and performance in real world applications. In these tests, we used 4 nodes and up to 128 UPC threads. UPC threads were distributed in a round-robin fashion, and when using the OpenSHMEM based runtimes the UPC threads were bound to socket. The three programs we tested are CPI [4], Stream-Triad [4] [13], and RandomAccess [4] [13]. All of these programs are example programs from the Berkeley UPC distribution. Two of the programs (RandomAccess and Stream-Triad) are UPC versions of HPC challenge programs [13].

Tests of evaluated applications are presented with error bars that represent an approximate 95% confidence interval above and below the mean. These values are calculated by respectively adding or subtracting 1.96 times the standard error from the mean value.

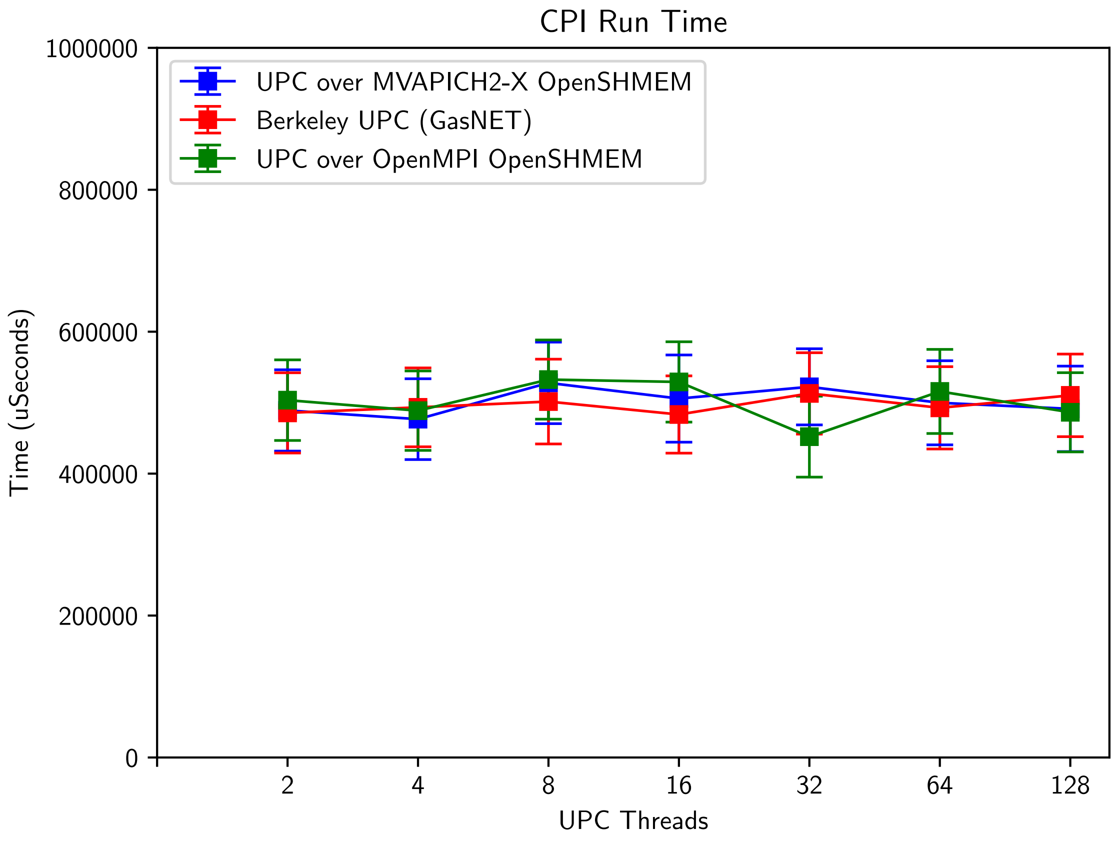

Figure 19: CPI average execution time over 100 runs for Berkeley UPC and UPC with OpenSHMEM Runtimes with UPC thread counts 2-128.

We ran the CPI program from the example programs packaged with Berkeley UPC. In this experiment, we ran the program on 4 nodes with 2-128 UPC threads. We ran the test 100 times for each UPC thread count and averaged the execution times. The runtimes display less than a 2.64% difference in terms of execution time on all the UPC thread counts tested. The standard deviation for this test is fairly large at approximately 290000 uSec for every runtime and thread count. We see the large differences in execution times between runs because this program contains very little synchronization, but we also see very small differences between average execution times between our runtime using OpenSHMEM and Berkeley UPC using GASNet.

We also tested the Stream-Triad program with UPC thread counts of 2 - 128. The test used a vector of 4000000 elements and an alpha of 1.5. This test is strongly scaled. This means the problem size does not increase as the UPC threads increase. These are the default settings for the Berkeley UPC test harness. As can be seen in Figure 20, our runtime using MVAPICH2-X OpenSHMEM shows up to a 39.00% decrease in best time at 64 UPC threads and 5.09% increase at 128 UPC threads.

We do see variation in terms of memory bandwidth. This is shown in Figure 21. Our runtime using MVAPICH2-X and OpenMPI OpenSHMEM shows a bandwidth increase of 63.12% and 49.15% respectively over Berkley UPC at 64 threads. At 128 threads, we show a bandwidth decrease of 4.58% and 36.21%. This drop in bandwidth could be explained by the inclusion of a *upc_barrier* statement in the timed section. As can be seen in Figure 22 the decrease in bandwidth at 128 UPC threads is reflected in an increase in latency for the OSU Barrier Latency test that was ran with the same experimental setup as the Stream-Triad test. Our OpenSHMEM runtimes show an increase in barrier latency at 128 UPC threads. This indicates that bandwidth is not decreasing, but synchronization is affecting the test.

Our final test for UPC is RandomAccess. We ran this test with thread counts of 2-128 UPC threads. As stated before, this test mainly demonstrates interprocess bandwidth. This includes network communication in addition to local shared memory operations. In our testing we used the default table size of 2^{21} words. Each word is represented by a unsigned

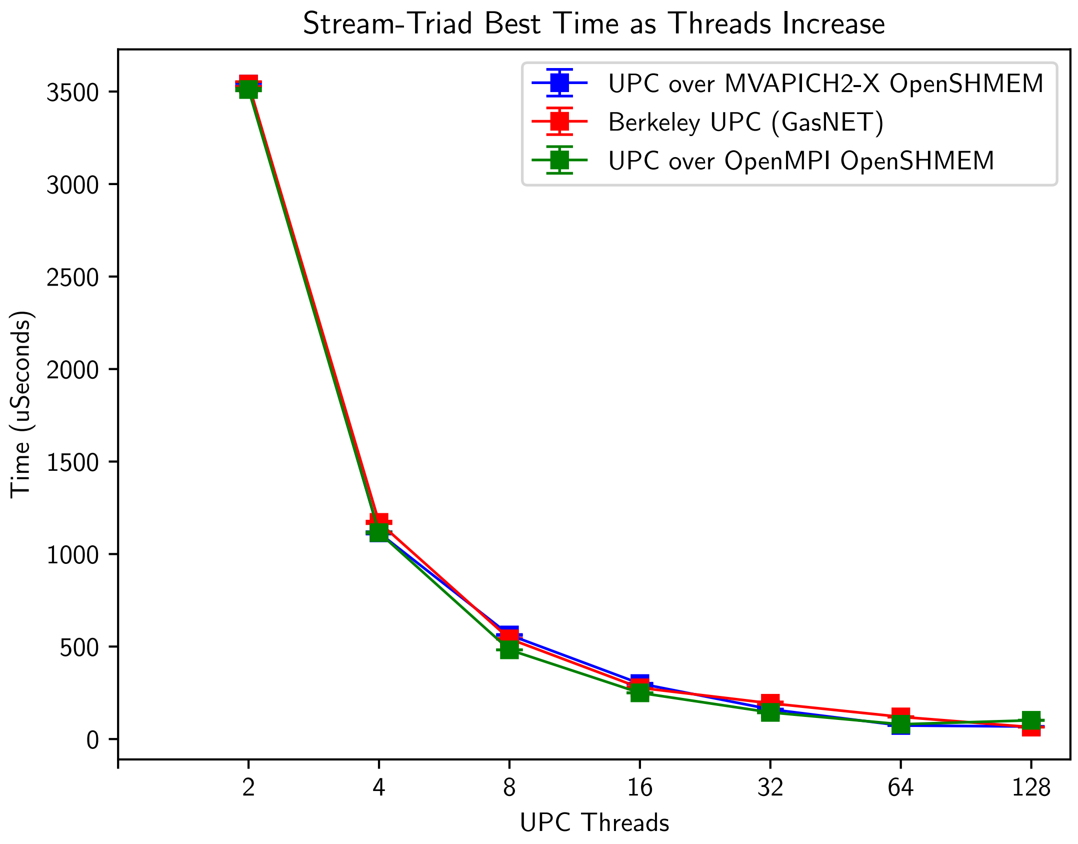

Figure 20: Stream-Triad average execution time for Berkeley UPC and UPC with OpenSHMEM Runtime with UPC thread counts 2-128.

64 bit integer. The test then performs 3,640,838 random RMA updates and measures the time required to perform them. The output of this test is in Giga-Updates Per Second (GUPS). On the nodes used in this test, 12 UPC threads can be bound to socket 0. This is the socket closest to the network interface card (NIC). With 4 nodes we have a total of 48 UPC threads that can be near the NIC. As can be seen in Figure 23, our OpenSHMEM runtime performs well compared to Berkeley UPC while all UPC threads can be bound to socket 0. As we begin to exceed this number at 64 UPC threads, the update latency increases and GUPS decrease. This effect is more pronounced for MVAPICH2-X OpenSHMEM.

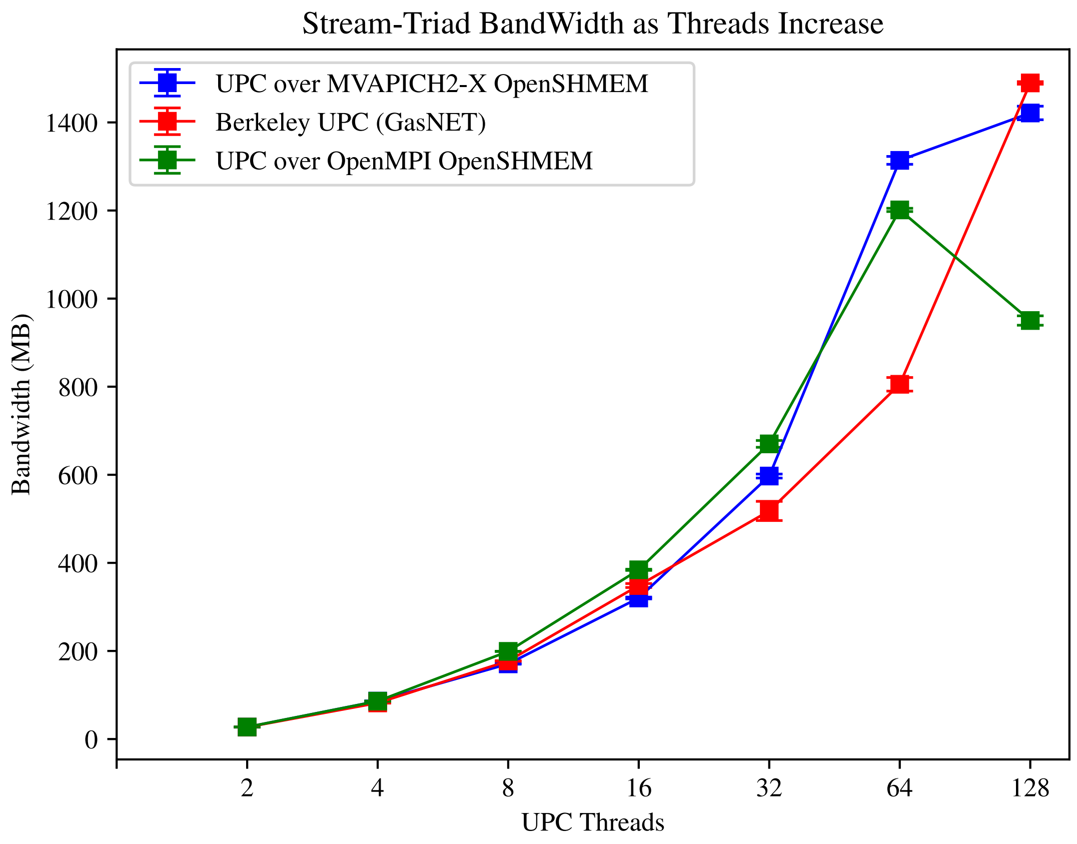

Figure 21: Stream-Triad average memory bandwidth for Berkeley UPC and UPC with OpenSHMEM Runtimes with UPC thread counts 2-128

Berkeley UPC seems to have the same issue, but happens at 128 threads rather than 64 when a larger ratio of threads are distant from the NIC.

These results show that our runtime implementation does not negatively affect normal operation or performance in most circumstances, and that unmodified UPC programs are possible to run using our runtime. This supports the idea that OpenSHMEM can be a performant runtime for UPC.

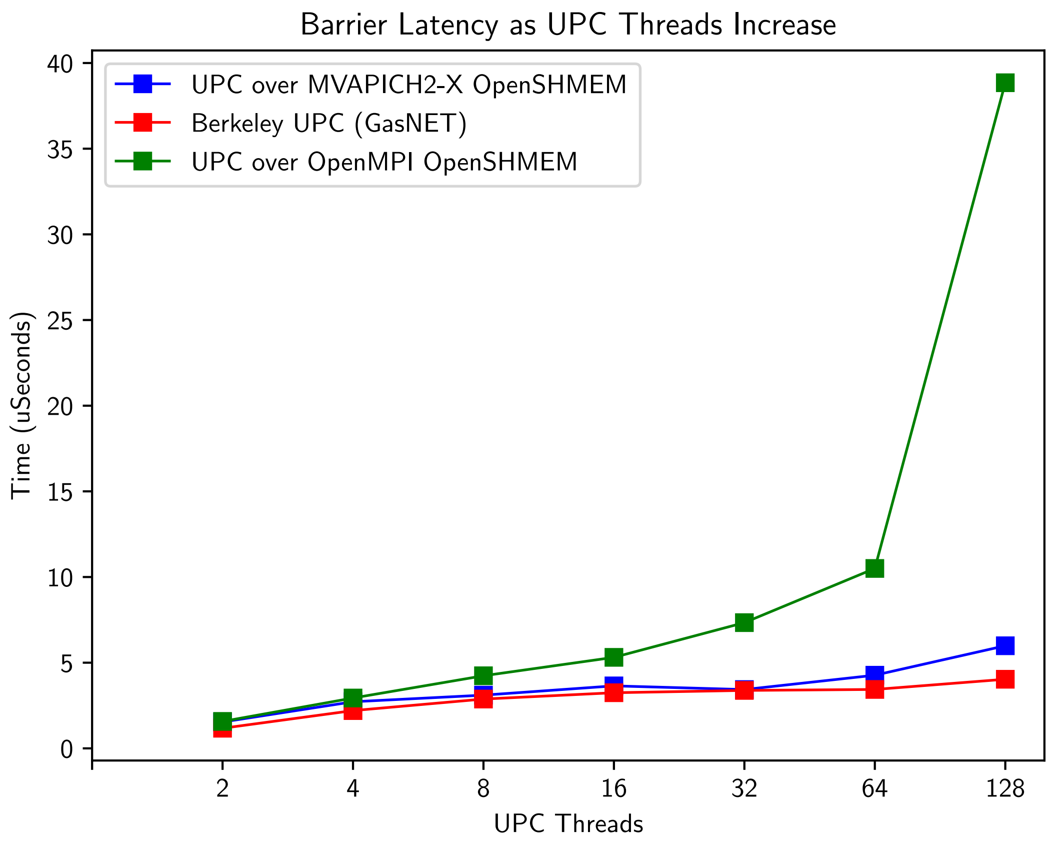

Figure 22: Barrier latency for 2-128 UPC threads. This demonstrates the barrier latency increasing as we see the bandwidth decreasing in the Stream-Triad tests.

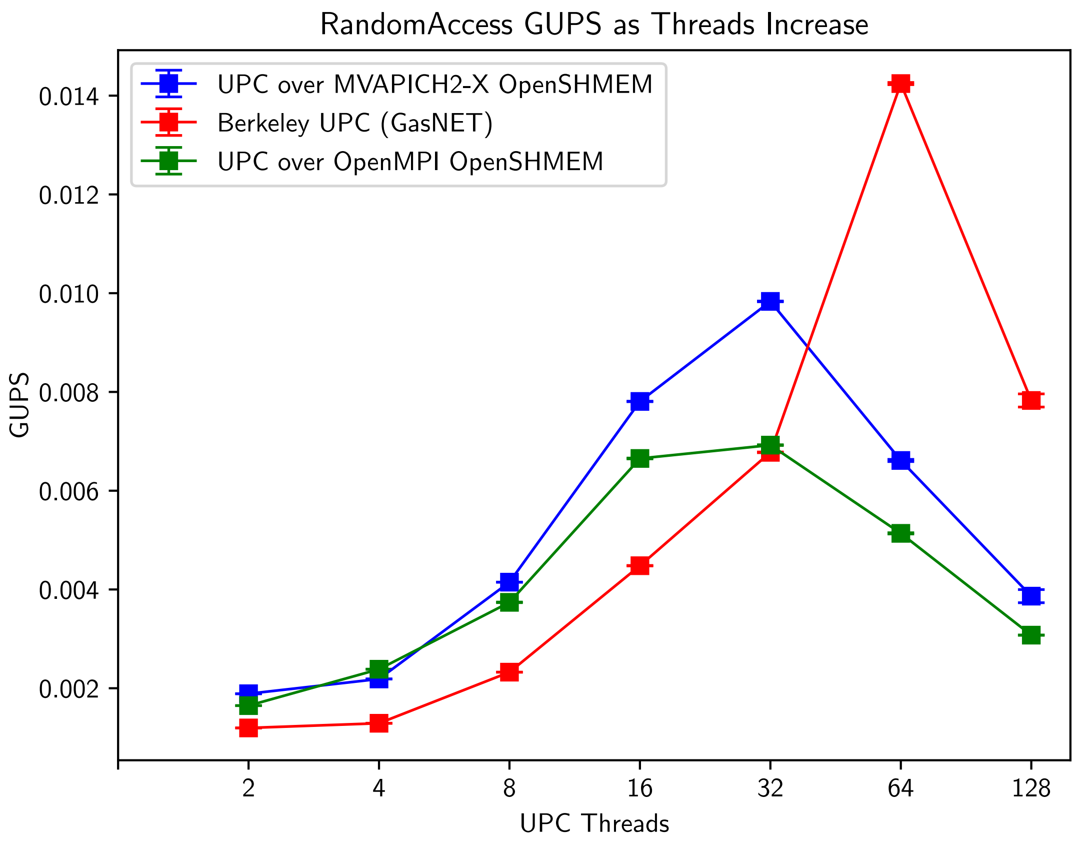

Figure 23: This figure shows execution time for the RandomAccess test for Berkeley UPC and UPC with OpenSHMEM runtimes with 2-128 UPC threads. The Giga-Updates Per Second drop as the number of threads exceeds what is possible to place on socket 0.

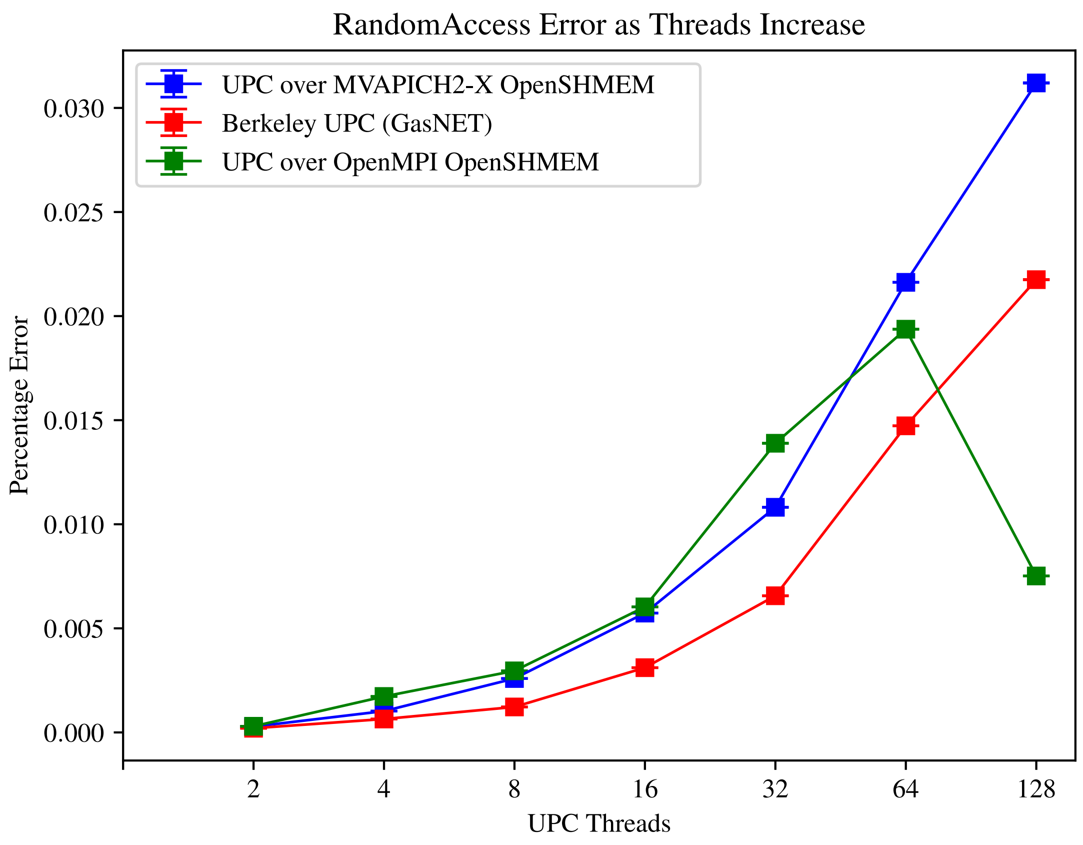

Figure 24: This figure shows the percentage error for the RandomAccess test for Berkeley UPC and UPC with OpenSHMEM runtimes with 2-128 UPC threads.

CHAPTER VII

Conclusion

Results

In this thesis, we analyzed UPC and developed a mapping of compatibility between the interfaces of UPC and OpenSHMEM in Chapter IV. We implemented a prototype UPC runtime with OpenSHMEM in Chapter V. This UPC runtime based on OpenSHMEM was then evaluated using MVAPICH and OpenMPI OpenSHMEM in Chapter VI.

We tested our runtime using the OSU Micro-Benchmarks. Our evaluation of PUT and GET latency with this runtime showed very minor differences for most data sizes of PUTs and GETs. But the OpenSHMEM based runtime did reduce latency when performing PUTs at smaller sizes. Our barrier tests with our UPC runtime using MVAPICH2-X OpenSHMEM showed up to a 17.79% increase in latency over Berkeley UPC. However, the broadcast test showed a split in increase and decrease in latency over Berkeley UPC. Our runtime using MVAPICH2-X showed a decrease in latency of up to 98.70% from Berkeley UPC. Our reduction tests showed split results once again. Our runtime using MVAPICH OpenSHMEM showed a significant decrease in latency for reduction sizes less than 512 bytes. We also demonstrated a possible route for optimization of reductions using lightweight threads.

We also tested the OpenSHMEM based runtime using some example programs. These example programs demonstrate that we can run UPC programs correctly. In addition, we saw no significant difference in the execution times of these programs. This leads us to the conclusion that OpenSHMEM can be a viable runtime for UPC.

Future Work

Our plans for future work include investigating how new OpenSHMEM features such as teams and contexts can be utilized in our runtime. One possible area to explore is how teams can be used for runtime level optimizations. UCX performs network-based atomics if one PE is accessible via the network, it may be possible to leverage teams so that CPU atomics are used when communication is only between shared memory capable PEs. To do this, we could expose an interface to the client language to be used in these situations, or perhaps the runtime could be made to do this optimization automatically.

One area to explore is the design of more intelligent threaded collectives. Some more profiling will be required for the runtime to intelligently create additional threads when required. In this paper, our threaded collectives used OpenMP threads. Other types of lightweight threads could be explored such as POSIX threads. Also, different algorithmic techniques should be explored so that communication patterns can be better optimized for each collective.

By using the proposed function, *shmem_malloc_with_hints()* we may be able to get additional performance from our runtime level operations which require the use of atomics. We would also like to do profiling of our runtime's memory accesses. It could be that some of our memory partitions are more likely to be involved in RMA operations than others. If this is the case, it would be desirable to allocate those memory partitions that are more likely to be involved in RMA closer to the network interconnect. This is not yet supported in the OpenSHMEM specification, but the ability to do this has been implemented by other researchers [27].

BIBLIOGRAPHY

[1] Baker, M. B., Welch, D. A., and Gorentla Venkata, M. Parallelizing the smith-waterman algorithm using openshmem and mpi-3 one-sided interfaces. 9397, 1 2015.

[2] Berkeley Lab,. Users/links. https://gasnet.lbl.gov/, 2020.

[3] Berkeley Labs,. *Manual Reference Pages - UPCC (1)*, 2019.

[4] Berkeley upc. https://bitbucket.org/berkeleylab/upc-runtime/wiki/Home, 2020.

[5] Bonachea, D. *The Berkeley UPC Runtime Specification*, 2002.

[6] Coti, C. and Malony, A. D. On the road to diposh: Adventures in high-performance openshmem. In *SC2019 OpenSHMEM BOF*, 2019.

[7] Cray Inc.,. *Chapel Language Specification Version 0.988*, 2019.

[8] Funck, G. and Vukicevic, N. Upc runtime design utilizing portals-4. http://gccupc.org/documents/portals4/portals4-upc-runtime-design.html, 2020.

[9] Grossman, M., Pritchard, H., Budimlić, Z., and Sarkar, V. Graph500 on openshmem: Using a practical survey of past work to motivate novel algorithmic developments. In *Proceedings of the Second Annual PGAS Applications Workshop*, PAW17, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Gupta, R., Tipparaju, V., Nieplocha, J., and Panda, D. Efficient barrier using remote memory operations on via-based clusters. In *In IEEE Cluster Computing*, page 83. IEEE Computer Society, 2002.

[11] Hamidouche, K., Zhang, J., Panda, D. K., and Tomko, K. Openshmem non-blocking data movement operations with mvapich2-x: Early experiences. In *2016 PGAS Applications Workshop (PAW)*, pages 9–16, Nov 2016.

[12] Hensgen, D., Finkel, R., and Manber, U. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17:1–17, 02 1988.

[13] Hpc challenge benchmark. http://icl.cs.utk.edu/hpcc/.

[14] Keahey, K., Riteau, P., Stanzione, D., Cockerill, T., Mambretti, J., Rad, P., and Ruth, P. Chameleon: a scalable production testbed for computer science research. In Vetter, J., editor, *Contemporary High Performance Computing: From Petascale toward Exascale*, volume 3 of *Chapman & Hall/CRC Computational Science*, chapter 5, pages 123–148. CRC Press, Boca Raton, FL, 1 edition, May 2019.

[15] Legion overview. https://legion.stanford.edu/overview/, 2020.

[16] Mvapich: Mpi over infiniband, omni-path, ethernet/iwarp, and roce. http://mvapich.cse.ohio-state.edu/, 2020.

[17] Namashivayam, N., Eachempati, D., Khaldi, D., and Chapman, B. Openshmem as a portable communication layer for pgas models: A case study with coarray fortran. In *2015 IEEE International Conference on Cluster Computing*, pages 438–447, Sep. 2015.

[18] Open mpi: Open source high performance computing. https://www.open-mpi.org/, 2020.

[19] OSU Network Based Computing Laboratory,. Mvapich: Mpi over infiniband, omni-path, ethernet/iwarp, and roce. https://mvapich.cse.ohio-state.edu/benchmarks/, 2020.

[20] Reid, J. *The new features of Fortran 2018*, 2018.

[21] Shamis, P., Venkata, M. G., Lopez, M. G., Baker, M. B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R. L., Liss, L., and others,. Ucx: an open source

framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[22] The OpenSHMEM Project,. *OpenSHMEM Application Programming Interface*, 2017.

[23] The OpenSHMEM Project,. Related sites. http://openshmem.org/site/Links, 2020.

[24] The Unified Communication X Library. http://www.openucx.org.

[25] UPC Consortium,. *UPC Language Specifications Version 1.3*, 2013.

[26] UPC Consortium,. *UPC Required Library SpecificationsVersion 1.3*, 2013.

[27] Venkata, M. G., Aderholdt, F., and Parchman, Z. Sharp: Towards programming extreme-scale systems with hierarchical heterogeneous memory. In *Conference: 6th International Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications*, 2017.

[28] Xpmem linux cross memory attach. https://gitlab.com/hjelmn/xpmem, 2020.

[29] Zhang, J., Behzad, B., and Snir, M. Optimizing the barnes-hut algorithm in upc. *Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, 11 2011.

[30] Zhang Zhang,, Savant, J., and Seidel, S. A upc runtime system based on mpi and posix threads. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 8 pp.–, 2006.