Collective Communication for the RISC-V xBGAS ISA Extension

Brody Williams
brody.williams@ttu.edu
Texas Tech University
Lubbock, Texas

Xi Wang
xi.wang@ttu.edu
Texas Tech University
Lubbock, Texas

John D. Leidel
jleidel@tactcomplabs.com
Tactical Computing Laboratories
Muenster, Texas

Yong Chen
yong.chen@ttu.edu
Texas Tech University
Lubbock, Texas

ABSTRACT

Parallel programming methodologies are fundamentally dissimilar to those of conventional programming, and software developers without the requisite skillset often find it difficult to adapt to these new methods. This is particularly true for parallel programming in a distributed address space, which is necessary for any meaningful degree of scalability. As such, an approach that combines a more intuitive interface together with excellent performance within the distributed address space model is desired. In this work, we present our initial API design and implementation as well as the underlying algorithms for a collective communication library built for the Extended Base Global Address Space (xBGAS) extension to the RISC-V microarchitecture. Our runtime library is designed to enact the Partitioned Global Address Space model (PGAS) in an attempt to alleviate the difficulty associated with traditional distributed address space programming while the underlying collective implementation is formulated to prevent the loss of, and even improve, performance over traditional solutions.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**; • **Computer systems organization →** *Architectures.*

KEYWORDS

PGAS, RISC-V, Remote Memory Access, Parallel Programming, Collectives

ACM Reference Format:
Brody Williams, Xi Wang, John D. Leidel, and Yong Chen. 2019. Collective Communication for the RISC-V xBGAS ISA Extension. In *48th International Conference on Parallel Processing: Workshops (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3339186.3339196

1 INTRODUCTION

Parallel computing has long been an intrinsic component of high-performance computing (HPC), but in the past was not necessary for most mainstream devices and workloads. As such, the ability to understand and develop parallel programs was typically a niche skill requisite only for developers who worked within the HPC field. In recent years, however, the practical end of Moore′s Law has thrust the paradigm into the mainstream. Faced with radically diminishing returns in single-core processor performance gains, manufacturers have been forced to implement multicore processors to improve performance by leveraging parallelism. As a result, application developers for these platforms have been forced to adopt shared address space parallel programming practices in order to fully utilize a system's resources. Consequently, straightforward directive-based models for programming within this model such as OpenMP and OpenACC have been well received and widely adopted. For developers whose activities are limited to such an environment, these APIs have proven both effective and sufficient for their needs.

Unfortunately, the scalability of shared address space systems is inherently limited and unsuitable for large-scale and more complex tasks. Those working with such problems must instead utilize distributed address space models, or a solution which incorporates both paradigms, for parallel computing at scale. Understanding of, and development within, these distributed address space models is usually considered more challenging for programmers when compared with their shared address space counterparts. The most widespread solution for programming within this model is the Message-Passing Interface, or MPI, which, although widely adopted, and proven effective, is not without its own limitations. The MPI runtime library, while comprehensive, is large and often daunting to the uninitiated. In addition, the software protocols MPI uses to communicate and pass data, particularly within the more commonly utilized two-sided communication scheme, force certain overheads and inherently limit performance.

Clearly, in this new era of computing, dominated by multicore processors and characterized by distributed systems with extensive remote data transactions, the need for an intuitive distributed address space parallel programming model with excellent performance has never been greater.

The Partitioned Global Address Space, or PGAS, is one such model introduced in an attempt to alleviate the difficulty associated with distributed address space programming. The PGAS model aspires to provide developers with an interface similar to the more familiar shared memory systems, but with the enhanced scalability characteristic of its distributed memory analogs. Conceptually, this is accomplished by building a global address space that, while directly accessible by any processing element, remains physically distributed. Notable existing implementations of the PGAS model such as Unified Parallel C (UPC), Global Arrays (GA), CoArray Fortran (CAF), Chapel, X10, and OpenSHMEM are implemented as either library extensions of an existing programming language or as an entirely new language. They provide developers with the simpler interface desired, but rely on MPI, Remote-Direct Memory Access (RDMA), or other low-level communication protocols to provide semantic support.

We have previously proposed the Extended Base Global Address Space, or xBGAS, microarchitecure extension to the RISC-V instruction set architecture (ISA) as an efficient means for providing a high performance, scalable, shared address space environment [12] [22]. We believe that xBGAS possesses the potential to further refine, and considerably improve the performance of, the PGAS model by incorporating remote addressing capabilities directly into the microarchitecture to support remote communication. In this work, we show our initial design and implementation for collective communication operations within the xBGAS runtime API [9] and evaluate the simulated performance to test the project's current viability and potential.

The contribution of this work is three-fold. First, we propose an API interface for the collectives of the RISC-V xBGAS extension that is designed to improve the overall usability of xBGAS and provide necessary functionality for a variety of applications. Second, we design the algorithms for the high-performance xBGAS collectives based on the ISA support for remote memory accesses in the xBGAS extension. Third, we implement the proposed collective operations in the xBGAS runtime library and evaluate the associated performance [25].

The remainder of this work is organized as follows. Section 2 briefly discusses previous related works that focus on improving the performance of collective operations utilizing MPI and OpenSHMEM. Section 3 provides an overview of the xBGAS architecture extension. Section 4 introduces the initial design and implementation of our collective communication runtime library. Section 5 describes our simulation environment and presents an evaluation of our current work. Section 6 details our analysis and conclusions for this work in its current state. Finally, Section 7 concludes with a brief discussion of future work.

2 RELATED WORK

Many implementations of the PGAS programming paradigm exist today as either active research projects or tools for software development. As such, it is unsurprising that considerable research efforts have been expended in an attempt to improve their performance. This includes endeavors specifically targeting collective communication. However, since many of these implementations take place at a relatively high level in software, as mentioned in Section 1, most efforts are limited to seeking performance improvements through algorithmic means. These techniques can be applied at the API level [19], but more often occur in the subordinate protocols used to realize the remote data transactions. Optimizations made at this level tend to have a far greater impact on performance that those made in the calling API.

Among the systems utilized to implement this low-level support, MPI is by far the most common. As a result, the MPI collective libraries have been the subject of a significant portion of these research efforts. Numerous studies have sought to improve performance through optimization of certain collective operations [1] [17] [3] [21] [2] while others, in contrast, have ventured a more comprehensive approach [20]. Although both methodologies have often been thoroughly successful, they are fundamentally hampered by the limitations of MPI.

In addition to these algorithmic approaches, techniques that include Remote Direct Memory Access (RDMA) are becoming more common and often result in more impressive performance improvements than those that rely on software optimizations alone [16] [13] [26]. In regard to to OpenSHMEM, both the standalone algorithmic techniques and those that incorporate RDMA are being explored [4] [6] [27] [14].

Although significant study has been devoted to improving collective communication performance, including for the PGAS model, as far as we are aware no existing work has been performed utilizing one-sided communication at the microarchitectural level as we attempt with xBGAS.

3 THE XBGAS EXTENSION

3.1 Functionality

The Extended Base Global Address Space project, hereafter referred to as xBGAS, is a proposed extension to the open-source RISC-V RV64I ISA that is designed to provide extended addressing capabilities and facilitate fast inter-node communication via remote load and store instructions.

As xBGAS is a general design independent of any particular system architecture, we believe that it possesses the potential to improve functionality and performance in a variety of application domains including high performance analytics, memory-mapped I/O, and high performance computing within the Partitioned Global Address Space model.

Our work thus far has been primarily targeted at the PGAS programming model. We believe that xBGAS is inherently suited to provide microarchitecture level support to the PGAS model by realizing the latter's one-sided communication calls through the use of extended ISA.

Existing PGAS implementations, as discussed in Section 1 and 2, already provide an intuitive PGAS interface as desired for a more developer-friendly experience. However, most of these high-level software solutions rely on MPI or other message-passing protocols to realize the low-level communication necessary between processing elements. As a result, each is subject to accompanying performance overheads related to socket setup, handshaking between processes, and system calls. Even systems that utilize remote direct memory access, or RDMA, are constrained by expensive function calls within libraries that themselves require considerable background knowledge to utilize effectively. Further, many such implementations also require enforcement of special synchronization and cache coherence protocols in order to ensure data consistency.

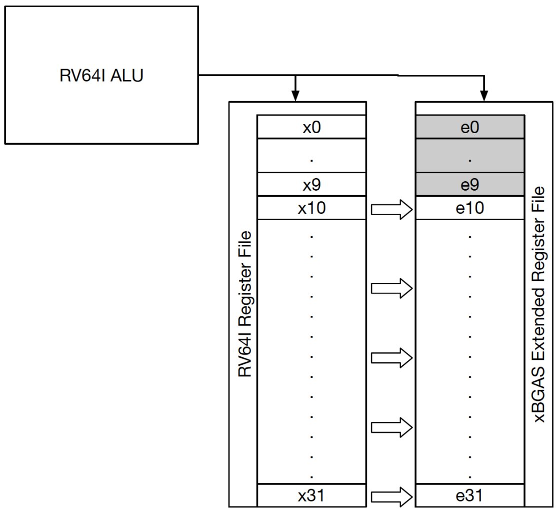

Figure 1: Extended xBGAS Register File

In contrast, xBGAS harnesses its unique remote load and store instructions to achieve a more intuitive form of one-sided communication. Use of these instructions allows for remote data transfers directly from the user-space and avoids both kernel involvement and data copying between network layers. We further postulate that the intuitive and lightweight xBGAS library will also reduce overhead even when compared to RDMA approaches.

3.2 Architecture & Instructions

In order to facilitate the use of remote load and store instructions as described in Section 3.1, several unique components are added to the standard RISC-V architecture. The first is a set of 32 xBGAS-specific registers, known as extended registers or "e" registers and designated as *e0-e31*, which are added to the RISC-V register file. These registers are analogous to the standard base registers, denoted by *x0-x31*, as shown in Figure 1. The extended and base registers are used in conjunction to form extended 128-bit addresses, wherein the extended register holds a unique object ID corresponding to a remote resource and the base register contains a standard 64-bit local memory address.

The second component necessary for the realization of xBGAS is a special hardware structure added to each physically disparate processing element know as the Object Look-Aside Buffer, or OLB. The OLB contains a mapping of every unique object ID to a remote physical address. Whenever a remote instruction is executed, the upper 64-bits of the address are retrieved from the specified extended register. If the value is equal to 0, representing the local processing element, a local memory operation is performed at the address given in the base register. Otherwise, the OLB is visited in order to translate the object ID into a remote physical address. The memory operation is then performed at the remote address given in the base register. It is important to note here that the xBGAS extension remains fully compatible with the standard RISC-V RV64I

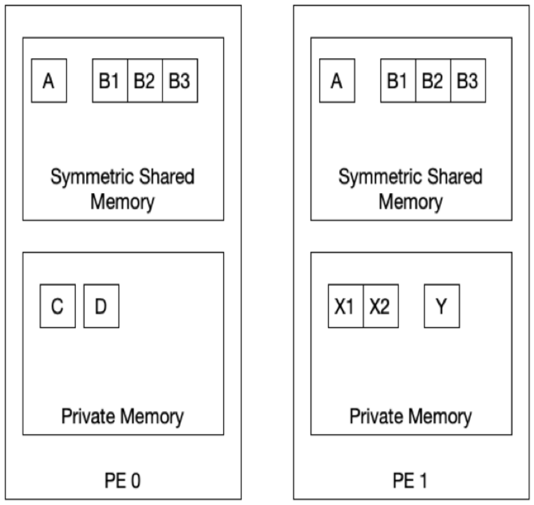

Figure 2: PGAS Memory Model

specification [23]. If the user disables the xBGAS extension, program execution will proceed normally utilizing only local memory access patterns without any adverse effects.

The xBGAS instructions themselves can be grouped into three categories:

- Base Integer Load/Store Instructions : These instructions utilize the same two-operand mnemonic found in the standard RISC-V ISA and automatically employ the extended register that naturally corresponds to the provided base register to form the remote effective address. As an example, the instruction:

  eld rd, imm(rs1)

  loads the 64-bit value from the effective address formed by *(ext1+rs1)+imm* (where imm is an immediate value) into rd.

- Raw Integer Load/Store Instructions : This set of instructions provides similar functionality to the Base Integer Load/Store instructions, but allows the user to explicitly specify both the extended and base register to form the remote effective address. Due to the reduced availability of encoding space, no immediate addressing is allowed for Raw-Type instructions. Comparable to the example given above, the instruction:

  erld rd, rs1, ext2

  loads a 64-bit value into rd. However, here the source effective address is formed by combining the object ID in ext2 together with the address in rs1.

- Address Management Instructions: The final category of xBGAS instructions do not directly perform any remote data accesses. Instead, they provide the capability to directly manipulate extended register contents for later use in other instructions.

The detailed xBGAS instructions can be found in the xBGAS specification [8].

3.3 API

The xBGAS API is formulated to provide the user with an intuitive C-based interface characteristic of the PGAS model that particularly mimics the structure used by the OpenSHMEM API[18]. The foundation of the underlying runtime environment is built upon the SHMEM-style implementation pioneered by Cray and also used by OpenSHMEM. This implementation provides both shared and private memory segments within each processing element. Calls that allocate memory within the shared address space are executed by each processing element. These allocations share either the same physical address with their corresponding remote allocations or, more commonly, the same offset from the beginning of the shared segment. In this manner, the shared-data segment of each processing element is kept fully symmetric with that of its peers. An example of this PGAS memory model with two PEs and both private and shared memory segments is shown in Figure 2. Remote data communication is then naturally accomplished through one-sided remote get and put calls to complementary memory allocations between processing elements.

Our previous work has provided the xBGAS library with a number of routines that implement functionality typically necessary for the PGAS and distributed address space parallel programming paradigms [9]. These include functions that initialize and terminate the xBGAS runtime environment, allocate and deallocate symmetric shared memory, query the number of running processing elements, return the unique ID of the calling processing element, and a simple barrier. As performance is of the utmost concern, the xBGAS library is designed to be as lightweight as possible and directly translates these high-level function calls into assembly instructions whenever possible.

The get and put functions are particularly significant as they allow the user to perform remote data transactions using the one-sided communication model. Explicit calls exist for different data types for ease of use. The mapping between these types and their corresponding type names, as used in function calls, is given in Table 1. The prototypes for these functions, which make use of the xBGAS extended load and store instructions, are shown below.

```
void xbrtime_TYPENAME_put(TYPE *dest, const TYPE *src,
size_t nelems, int stride, int pe)
```

```
void xbrtime_TYPENAME_get(TYPE *dest, const TYPE *src,
size_t nelems, int stride, int pe)
```

Above, dest is a pointer to a destination address, src is pointer to a source address, nelems is total number of elements on which the data transaction is to be performed, stride is the stride size between consecutive elements at src and dest, and pe is the unique ID of the target processing element. Although not shown, non-blocking forms of both get and put are also included in the library for the given data types. These remote data access functions are further optimized in the underlying assembly code by utilizing loop unrolling when nelems exceeds a given threshold.

Experimentation with the xBGAS library in this state led to the realization that it was missing several key features. Among those features were collective communication operations. In the next section we detail our initial design and implementation of collective operations built for xBGAS using the point-to-point protocols described above.

Table 1: xBGAS Matched Type Names & Types

TYPENAME	TYPE
float	float
double	double
longdouble	long double
char	char
uchar	unsigned char
schar	signed char
ushort	unsigned short
short	short
uint	unsigned int
int	int
ulong	unsgined long
long	long
ulonglong	unsigned long long
longlong	long long
uint8	uint8_t
int8	int8_t
uint16	uint16_t
int16	int16_t
uint32	uint32_t
int32	int32_t
uint64	uint64_t
int64	int64_t
size	size_t
ptrdiff	ptrdiff_t

4 COLLECTIVE OPERATIONS

4.1 Design Considerations

Given that a considerable portion of the communication that takes place within the distributed address space model does so in the form of collective communication [20], it is unsurprising that substantial research efforts have been directed at optimizing these operations. Numerous studies have shown that, regrettably, there is no universally optimal solution suited to every occasion [24]. Instead, the algorithm best suited to provide the highest performance for a given collective call depends on a number of factors including the data transaction size, the number of processing elements participating in the operation, and network topology [20]. As such, most state-of-the-art solutions include a variety of algorithms which are dynamically chosen from at runtime based on the arguments of a specific call [15]. It follows then, that the xBGAS collective library must follow a similar pattern in order to achieve a comprehensive solution.

4.2 The Binomial Tree

As such an inclusive realization will be a considerable undertaking, it is important that our initial implementation provide the necessary functionality while remaining as generally applicable as possible. Consequently, we build our early library around the use of a binomial tree algorithm and incorporate recursive doubling and halving principles in order to minimize network congestion and optimize dissemination across physically distributed resources [5].

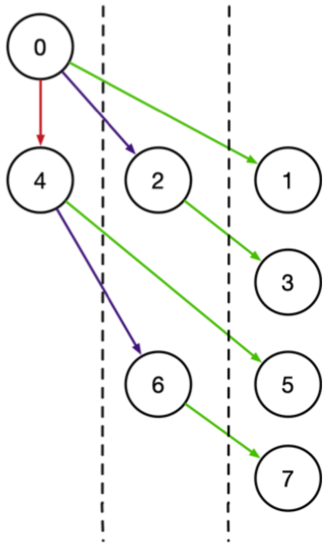

Figure 3: Binomial Tree with Recursive Halving

The binomial tree structure, as shown in Figure 3 possesses several key characteristics that make it ideal for such a broad scenario. First, the binomial tree requires a minimal degree of connectivity among processing elements in order to function. This allows us to forgo making any assumptions about network topology and ensure our collective library will perform effectively regardless of whether it is utilized on a torus or hypercube topology. Further, previous studies have shown that tree-based algorithms, including the binomial tree, typically produce the highest performance for smaller data transaction sizes where communication latency is the greatest bottleneck [20]. We anticipate that an appreciable portion of our expected workloads use these small transaction sizes. Regardless of data transaction size, the structure of the binomial tree provides an upper bound of $O(\lceil log_2 N \rceil)$ communication steps, where N is the number of processing elements. Finally, the binomial tree algorithm can be adapted to implement four of the most common collective operations utilized in distributed address space parallel programming: broadcast, reduction, scatter, and gather. Given that these are the collective operations most often utilized, and that they can be combined together to accomplish the semantics of several more complex operations, we have elected to implement them as the basis of our initial collective library.

In the following subsections, we introduce the API level function calls and variants of the binomial tree algorithm associated with each of the previously enumerated collective operations. We conclude with a brief comparison of our implementation against that of OpenSHMEM model.

4.3 Broadcast

We first present our design and implementation of the broadcast collective operation. The prototype for the C-language broadcast function is shown below.

```
void xbrtime_TYPENAME_broadcast(TYPE *dest,
const TYPE *src, size_t nelems, int stride, int root)
```

The TYPENAME and TYPE placeholders follow the same conventions previously discussed for the get and put operations and are shown in Table 1. The function call itself is designed to present an interface that is as simple as possible and emulates the calls of the xBGAS get and put functions. This also benefits the underlying implementation which makes direct use of put operation. Above, dest is a pointer to the symmetric destination address for broadcast values on each PE, src is a pointer to the (not-necessarily shared) address for these values on the root pe, nelems represent the total number of data items to be broadcast to each PE, stride refers to the stride size between consecutive elements at both src and dest, and root is the rank of the root PE.

Algorithm 1: Broadcast (dest, src, nelems, stride, root)

```
n_pes ⟵ Number of PEs calling collective operation
log_rank ⟵ Calling PE rank
if $log\_rank \geq root$ then
    vir_rank ⟵ (log_rank - root)
else
    vir_rank ⟵ (log_rank + n_pes) - root
mask ⟵ $(2^{\lceil log_2(n\_pes) \rceil} - 1)$
for $i = (\lceil log_2(n\_pes) \rceil - 1)$ to $i = 0$ do
    mask ⟵ mask XOR $2^i$
    if (vir_rank AND mask) == 0 then
        if (vir_rank AND $2^i$) == 0 then
            vir_part ⟵ (vir_rank XOR $2^i$) mod n_pes
            log_part ⟵ (vir_partner + root) mod n_pes
            if $vir\_rank < vir\_part$ then
                put(dest, src, nelems, stride, log_part)
```

The algorithm used to implement the broadcast semantics, as shown in Algorithm 1, is the most straightforward of the four we will present and includes several important components that will be replicated in the others. Each algorithm, including broadcast, first fetches both the number of PEs involved in the current collective operation and its own logical rank. These values are stored in variables *n_pes* and *log_rank*, respectively. It is then necessary to assign each PE a virtual rank, given as *vir_rank*, in order to accommodate collective operations where the root PE is not equal to rank 0. These virtual ranks are assigned such that the root PE always receives *vir_rank* 0. Consecutive virtual ranks are then allocated in sequence to each PE based on its logical rank relative to the root. For example, if a collective call is made with 7 PEs (whose logical ranks are given by 0-6) and PE 4 is taken to be the root for the call, then virtual ranks are assigned as shown in Table 2.

After these requisite variables are prepared, we are able to start our broadcast in earnest. Through the use of the tree, we are able to accomplish our overall task via stages of point-to-point communication. This is realized using a loop with $\lceil log_2(n\_pes) \rceil$ iterations in order to fully disseminate the broadcast values. Operating under the assumption that PE ranks are likely to be assigned sequentially within a given node, we apply the concept of recursive-halving to our broadcast pattern in an effort to more efficiently distribute subsequent communication within the tree and minimize network

Table 2: Logical to Virtual Rank Mapping

log_rank	vir_rank
0	3
1	4
2	5
3	6
4	0
5	1
6	2

congestion. In order to do so, the loop index, *i*, takes the value ($\lceil log_2(n\_pes)\rceil$ -1) for the first iteration and is then decremented for each subsequent iteration. The final iteration takes place when *i = 0*. During each iteration of the loop, we utilize a previously initialized mask to perform a bitwise comparison against each PE's virtual rank. This mask is modified at each stage of the loop in order to isolate individual bits in a left to right manner. If a given PE proves to be a candidate for communication relative to the current iteration, it is assigned a virtual partner that is a distance of 2^i from itself. A final verification is made to ensure that a PE's virtual rank is less than that of its virtual partner. This prevents any invalid communication from taking place when n_pes is not a power of two. Finally, if all the above conditions are met, a remote *put* from the qualifying virtual rank to its virtual partner of the broadcast values is performed. While not shown in Algorithm 1, a barrier operation takes place at the end of each loop iteration to ensure correct synchronization.

4.4 Reduction

Our reduction collective library function, as seen below, is very similar to the one used by broadcast.

```
void xbrtime_TYPENAME_reduce_OP(TYPE *dest,
const TYPE *src, size_t nelems, int stride, int root)
```

As before, dest and src are pointers to corresponding addresses, nelems it the total number of data elements that the reduction operation is to be performed on, stride is the stride size, and root is the root PE rank. However, there is one important distinction. As the reduction algorithm makes use of the *get* function, as opposed to the *put* in broadcast, the address at src must be a shared symmetric address while dest, significant only for the root PE, may be either shared or private. Currently, our reduction implementation supports sum, product, min, and max operations for all types listed in Table 1. Additionally, bitwise AND, bitwise OR, and bitwise XOR are supported for non-floating point types.

Similar to the function call itself, the reduction collective algorithm, as seen in Algorithm 2, bears an unmistakable resemblance to that of its broadcast analog. Variables for *n_pes*, *log_rank*, and *vir_rank* are assigned as before. Thereafter, slight differences occur. Most conspicuously, two extra variables, *l_buff* and *s_buff*, are utilized. These variables, a local private buffer and a shared address buffer, respectively, are employed in order to prevent any unintended overwriting of values on any PE. As shown, each PE begins by loading its shared buffer with its personal reduction values before communication begins. Afterwards, our main communication

Algorithm 2: Reduction (dest, src, nelems, stride, root)

```
n_pes ← Number of PEs calling collective operation
log_rank ← Calling PE rank
if $log\_rank \geq root$ then
    vir_rank ← (log_rank - root)
else
    vir_rank ← (log_rank + n_pes) - root
for $i = 0$ to $i = (nelems - 1)$ do
    s_buff[$i \times stride$] = src[$i \times stride$]
mask ← ($2^{\lceil log_2(n\_pes)\rceil}$ - 1)
for $i = 0$ to $i = (\lceil log_2(n\_pes)\rceil - 1)$ do
    mask ← mask XOR $2^i$
    if $(vir\_rank\ OR\ mask) == mask$ then
        if $(vir\_rank\ AND\ 2^i) == 0$ then
            vir_part ← (vir_rank XOR $2^i$)  mod n_pes
            log_part ← (vir_partner + root)  mod n_pes
            if $vir\_rank < vir\_part$ then
                get(l_buff, src, nelems, stride, log_part)
                for $j = 0$ to $j = (nelems - 1)$ do
                    s_buff[$j \times stride$] = s_buff[$j \times stride$] OP
                      temp[$j \times stride$]
if $vir\_rank == 0$ then
    for $k = 0$ to $k = (nelems - 1)$ do
        dest[$k \times stride$] = s_buff[$k \times stride$]
```

loop begins, although this time with the important difference that our loop index, *i*, starts at 0 and increases until the final iteration is performed when $i = \lceil log_2(n\_pes)\rceil$. This detail allows our mask to isolate bits in each PE's virtual rank from right to left as opposed to the left to right manner used for broadcast. Combined with slightly altered bitwise comparisons, this allows our tree to reverse the direction of data flow. Utilizing the *get* function, and the same virtual partner scheme as in broadcast, data is now moved from the leaves of the tree towards the root node with recursive doubling to preserve efficiency. At each intermediate stage, reduction values are placed by *get* into the calling PEs private buffer. The specified reduction operation is then performed on these values and the aggregate results of previous iterations. The product of this calculation is then placed in the PE's shared buffer for use during the next iteration. At the end of the loop, the final reduction values in the root PE's shared buffer are migrated to dest and the buffers are freed.

4.5 Scatter

The two last collective operations that we implement as part of our initial collective library are scatter and gather. Although both retain many elements found in the broadcast and reduction implementations, they are slightly more complex in order to accommodate an enhanced degree of versatility. We will first examine the scatter collective whose function prototype is shown below.

```
void xbrtime_TYPENAME_scatter(TYPE *dest, const TYPE *src,
int *pe_msgs[], int *pe_disp[], size_t nelems, int root)
```

Arguments *dest*, *src*, and *root* function as previously described. Similar to the preceding collectives, *nelems* represents the total number of data elements to be scattered from the root PE. We also now make use of two new arguments, *pe_msgs* and *pe_disp*, both of which are pointers to integer arrays of length *n_pes*. The array located at *pe_msgs* contains the number of data elements that are to be scattered to each PE indexed by PE logical rank. In the same manner, the array at *pe_disp* provides the offsets that each PE's values begin at relative to *src*. Inclusion of these arguments allows our scatter collective to distribute a distinct number of data elements to each PE.

Algorithm 3: Scatter (dest, src, pe_msgs, pe_disp, nelems, root)

```
n_pes ⟵ Number of PEs calling collective operation
log_rank ⟵ Calling PE rank
if log_rank ≥ root then
    vir_rank ⟵ (log_rank - root)
else
    vir_rank ⟵ (log_rank + n_pes) - root
for i = 0 to i = (n_pes - 1) do
    Calculate adj_disp[i] relative to root PE rank
if vir_pe == 0 then
    for i = 0 to i = (n_pes - 1) do
        Load s_buff[i] with messages relative to vir_rank
mask ⟵ $(2^{\lceil log_2(n\_pes) \rceil} - 1)$
for i = ($\lceil log_2(n\_pes) \rceil$ - 1) to i = 0 do
    mask ⟵ mask XOR $2^i$
    if (vir_rank AND mask) == 0 then
        if (vir_rank AND $2^i$) == 0 then
            vir_part ⟵ (vir_rank XOR $2^i$)  mod n_pes
            log_part ⟵ (vir_partner + root)  mod n_pes
            if vir_rank < vir_part then
                msg_size ⟵ # elems for vir_part and children
                put(s_buff[(adj_disp [vir_part])],
                  s_buff[(adj_disp [vir_part])], msg_size, 1,
                  log_part)
for i = 0 to pe_msgs[vir_rank] do
    dest[i] ⟵ l_buff[(adj_disp [vir_rank])]
```

Predictably, the underlying scatter algorithm, show in Algorithm 3, must also be more complex in order to accommodate both this flexibility and the semantics of the scatter collective itself. In stark contrast with the broadcast and reduction collectives, the objective of a scatter operation is to distribute a distinct segment of the original data values on the root to each PE. Therefore, the data transferred to each target PE during every iteration of the binomial tree pattern must include not only the data for the target PE itself, but also the values that are destined for its children. This ensures that the required data is present to be passed during subsequent iterations of the loop.

Another obstacle emerges as a result of scatter calls made with non-zero rank roots and our recursive halving methodology. Since the data values to be scattered at *src* are ordered according to logical rank, the individual PE-specific segments of a combined message intended for a target PE may not be contiguous. As an example, consider the same scheme used in our broadcast example, and detailed in Table 2, wherein we have 7 PEs and PE 4 is the logical rank of the root PE. If we perform a scatter operation using the same organization virtual rank 0 (logical rank 4) will need to perform a *put* operation to virtual rank 2 during the second loop iteration. This *put* will need transfer the data intended for virtual rank 2 as well as virtual rank 3 so that it may be passed on in the next iteration. However, the data for virtual rank 2 and virtual rank 3 is not contiguous at src. As a *put* cannot be performed on non-contiguous data, two separate messages would need to be generated in order to transfer the required values. Proceeding in this manner would severely impact performance and necessitate considerable changes to the communication structure.

Our implementation resolves this problem by reordering the values at src in a shared address buffer on the root PE by virtual rank before communication begins. This ordering is then maintained in the shared buffer of each PE that is used during the tree communication pattern. The reordering guarantees that the data for each tree node and its children is contiguous and ensures that a single *put* is sufficient at each stage of communication. An additional array of adjusted displacements, *adj_disp*, is created and used to ensure correct indexing during this process. The message size at each stage is dynamically calculated from the number of data elements assigned to be scattered to each PE. The loop iteration index, mask, virtual partner assignment, and eligibility for communication behave as in the broadcast algorithm. After the necessary communication is complete, each PE relocates its assigned values from its buffer to the corresponding address at dest.

4.6 Gather

Our final collective operation implemented is the gather operation. As it is symmetric to the scatter operation in the same manner that reduction is to broadcast, the function prototype is similarly mirrored.

```
void xbrtime_TYPENAME_gather(TYPE *dest, const TYPE *src,
int *pe_msgs[], int *pe_disp[], size_t nelems, int root)
```

In this incarnation, *nelems* is the total number of elements that will be gathered to the root PE, *pe_msgs* is a pointer to an array dictating how many values to gather from each PE, and *pe_disp* describes the displacements relative to dest where each PE's values will be placed on the root PE.

The gather algorithm is given in Algorithm 4. Once more we proceed in much the selfsame manner as is shown for the other collectives. As we desire the same flexibility to gather a distinct number of values from each PE, we mimic the modifications necessary for the scatter operation. Each PE first calculates the prerequisite adjusted displacement values and then loads its shared address buffer with its candidate gather data at the appropriate offset. The tree communication loop then proceeds using recursive doubling and the *get* method to systemically aggregate elements from each child node toward the root PE. Analogous to the scatter algorithm, the message size is dynamically calculated for each loop iteration. After the communication procedure is complete, the root PE only needs

Algorithm 4: Gather (dest, src, pe_msgs, pe_disp, nelems, root)

```
n_pes ← Number of PEs calling collective operation
log_rank ← Calling PE rank
if $log\_rank \geq root$ then
    vir_rank ← (log_rank - root)
else
    vir_rank ← (log_rank + n_pes) - root
for $i = 0$ to $i = (n\_pes - 1)$ do
    Calculate adj_disp[$i$] relative to root PE rank
for $i = 0$ to $i = (pe\_msgs[log\_rank])$ do
    s_buff[($adj\_disp[vir\_rank]$) + $i$] ← src[$i$]
mask ← ($2^{\lceil log_2(n\_pes) \rceil}$ - 1)
for $i = 0$ to $i = (\lceil log_2(n\_pes) \rceil - 1)$ do
    mask ← mask XOR $2^i$
    if (vir_rank OR mask) == mask then
        if (vir_rank AND $2^i$) == 0 then
            vir_part ← (vir_rank XOR $2^i$) mod n_pes
            log_part ← (vir_partner + root) mod n_pes
            if $vir\_rank < vir\_part$ then
                msg_size ← # elems for vir_part and children
                get(s_buff[($adj\_disp[vir\_part]$)],
                  s_buff[($adj\_disp[vir\_part]$)], msg_size, 1,
                  log_part)
if vir_rank == 0 then
    for $i = 0$ to (nelems - 1) do
        Load dest[$i$] with messages relative to log_rank
```

to reorder the values with respect to logical rank to complete the operation.

4.7 OpenSHMEM Comparison

We conclude this section by summarizing our observations regarding the state and functionality of our initial collective library implementation with the state-of-the-art PGAS solution offered by the OpenSHMEM model. One of the most obvious differences visible to the end user between our design and that of OpenSHMEM API is the collective function calls themselves. Whereas OpenSHMEM utilizes calls that are distinguished by the underlying data type size [18], our library chooses to provide explicit calls for each data type supported. While we acknowledge that this increases the size of the code base, we believe this explicit naming will be more intuitive for developers who might not possess the necessary background knowledge regarding data type sizes.

We also observe that, under certain circumstances, our collectives library may offer increased versatility, albeit of at the cost of extra function arguments. For example, our design allows for the use of varied stride sizes in both the broadcast and reduction collectives. Currently, the OpenSHMEM model does not support a non-default stride size for these operations. Our library also features the scatter operation, while this functionality is not provided in the OpenSHMEM API.

Conversely, the results of OpenSHMEM's reduction collective as well as those of collect and fcollect, its interpretation of the gather operation, are automatically distributed to each PE within the calling set. Our implementation does not currently feature single calls with the same functionality and must instead be accomplished through the use of a broadcast operation following the original call. We do also recognize that the OpenSHMEM SHCOLL [14] collective implementation is much more fully featured than our current design and is likely to produce better performance results against a diverse set of benchmarks.

5 EVALUATION

5.1 Implementation and Simulation

We implement and simulate our work with an environment based on RISC-V cores with the RV64I instruction set architecture (ISA). The xBGAS simulation infrastructure is implemented in the RISC-V Spike ISA simulator, which is extended with support for the xBGAS ISA and MPICH 3.2 to simulate inter-node communication between processing elements [10]. The implementation of our proposed collectives and the associated APIs are integrated into the xBGAS runtime library [9]. The tested workloads are compiled by the xBGAS RISC-V GNU compiler toolchain (riscv64-unknown-elf-gcc) to translate the extended xBGAS instructions into binaries that can be recognized by the Spike simulator. The simulation environment itself consists of 12 RISC-V cores where each core is configured with a 256-Entry TLB and 8-way set associative L1 (16KB) and L2 (8MB) caches. For those who may be interested, demonstration videos providing a brief tutorial of the xBGAS environment and the collective library may be found at [22] and [25], respectively.

5.2 Benchmarks

We evaluate the feasibility of our approach and test the performance of our current implementation using the NAS Integer Sort and GUPs benchmarks adapted from Oak Ridge National Lab's OpenSHMEM benchmark suite [7]. These benchmarks are commonly used for testing performance in distributed address space programming models and both make use of the reduction and broadcast collective operations. In order to maintain the integrity of results, we modify the code for the benchmarks as little as possible and replace only OpenSHMEM library calls with their xBGAS equivalents. The GUPs benchmark is run with the verification features enabled to guarantee correct execution. NAS Integer Sort is similarly executed with its detailed timing functionality enabled and using the class B problem size. Results for the two benchmarks are reported below for simulations with 1, 2, 4, and 8 PEs.

5.3 Results & Analysis

Performance for each of our benchmarks is reported in terms of *operations per second*. The GUPs benchmark, true to its namesake, natively returns results expressed in terms of billions of operations per second. The results for Integer Sort, however, are detailed as millions of operations per second. So that we may more easily compare these two benchmarks, the GUPs results are transformed and presented in terms of millions of operations per second.

The performance results for the GUPs benchmark are shown in Figure 4. The total number of operations performed, as well as the number per PE, are shown. Overall, the total number of operations performed scales fairly linearly as the number of PEs increases.

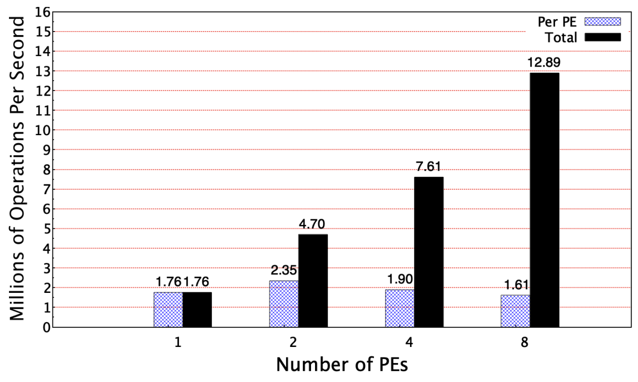

Figure 4: GUPs Performance

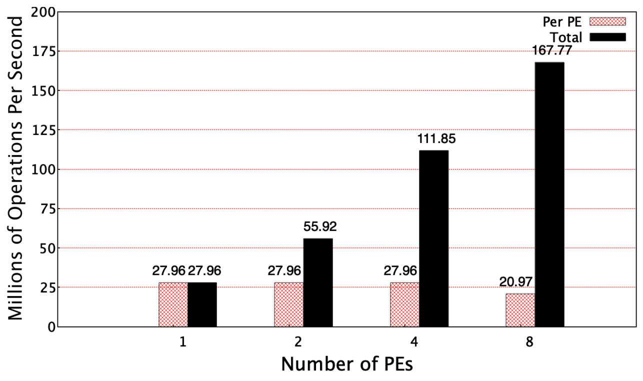

Figure 5: Integer Sort Performance

Notably, the performance exhibited per PE exceeds the baseline in the case of both 2 and 4 PEs while it decreases for 8 PEs. Peak performance of 2.35 million operations per second is achieved with 2 PEs.

Similar results are demonstrated from our testing on the NAS Integer Sort benchmark. As presented in Figure 5, the same linear increase in the total number of operations per second occurs for 2 and 4 PEs. The number of operations per PE also remains consistent across these tests. This trend begins to deteriorate, however, when 8 PEs are simulated. Performance per PE drops by about 25% here and causes a decrease in the overall performance.

Altogether, given our current simulation environment and the initial collective implementation currently in place, the results gathered thus far are promising. We are optimistic that further research and development will allow us to achieve more impressive performance results across a variety of different benchmarks. We will continue to port further benchmarks and applications and expect to report more results to the community in the near future.

6 CONCLUSION

In this work, we have presented our initial design and implementation of a collective communication library for the RISC-V xBGAS ISA extension. Our library leverages the unique microarchitecture-level advantages offered by xBGAS to the PGAS programming model to provide efficient collective communication built from point-to-point protocols.

We have shown the API-level collective function calls that are designed to be both intuitive for the user and flexible enough for a variety of situations. Further, we have presented in detail the underlying variants of the binomial tree algorithm used to realize these collective operations. Finally, we discussed our efforts to simulate and evaluate our prototype library using well-known distributed address space benchmarks.

We will continue our research with the xBGAS ISA extension and its collective library in the hope that it will become widely advantageous for use in modern high performance computing. We invite anyone interested in the project to experiment with the xBGAS code base [10] [9] and/or contact the authors.

7 FUTURE WORK

Overall, our xBGAS collective library is still in the early stages of development. Further work is needed to provide a comprehensive solution that is able to dynamically adapt and perform well in a variety of different contexts. Foremost, algorithms optimized for larger message sizes and specific network topologies need to be added to our existing binomial tree methodology. We also plan to add support for further collective operations including personalized all-to-all communication as well as explicit reduction-to-all and gather-to-all calls. Integration of collective functionality between a subset of PEs [4], location aware communication optimization using the xBGAS OLB, and non-blocking collectives are further areas we intend to explore. Finally, we are currently working to improve our simulation capabilities by combining the Sandia Structural Toolkit and STAKE to provide a cycle-accurate infrastructure [11].

ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers for their valuable feedback. This research is supported in part by the National Science Foundation under grants CNS-1338078, CNS-1362134, and CCF-1409946.

REFERENCES

[1] Mike Barnett, Lance Shuler, Robert van De Geijn, Satya Gupta, David G Payne, and Jerrell Watts. 1994. Interprocessor collective communication library (InterCom). In *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE, 357–364.
[2] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems* 8, 11 (1997), 1143–1156.
[3] Kiril Dichev, Vladimir Rychkov, and Alexey Lastovetsky. 2010. Two algorithms of irregular scatter/gather operations for heterogeneous platforms. In *European MPI Users' Group Meeting*. Springer, 289–293.
[4] James Dinan and Mario Flajslik. 2014. Contexts: a mechanism for high throughput communication in OpenSHMEM. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 10.
[5] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. 2003. *Introduction to parallel computing*. Pearson Education.
[6] Jithin Jose, Krishna Kandalla, Jie Zhang, Sreeram Potluri, and DKDK Panda. 2013. Optimizing collective communication in openshmem. In *7th International Conference on PGAS Programming Models*. 185.
[7] Oak Ridge National Labs. [n. d.]. Oak Ridge OpenSHMEM Benchmarks. https://github.com/ornl-languages/osb
[8] Tactical Computing Labs. [n. d.]. RISC-V Extended Addressing Architecture Extension Specifi
cation Codenamed: xBGAS. https://github.com/tactcomplabs/xbgas-archspec

[9] Tactical Computing Labs. [n. d.]. xBGAS Machine-Level Runtime Library. https://github.com/tactcomplabs/xbgas-runtime
[10] Tactical Computing Labs. [n. d.]. xBGAS RISC-V ToolChain. https://github.com/tactcomplabs/xbgas-tools
[11] John D Leidel. 2018. Stake: a coupled simulation environment for RISC-V memory experiments. In *Proceedings of the International Symposium on Memory Systems*. ACM, 365–376.
[12] John D Leidel, Xi Wang, Frank Conlon, Yong Chen, David Donofrio, Farzad Fatollahi-Fard, and Kurt Keville. 2018. xBGAS: Toward a RISC-V ISA Extension for Global, Scalable Shared Memory. In *MCHPCâĂŽ18: Workshop on Memory Centric High Performance Computing*.
[13] Amith R Mamidala, Jiuxing Liu, and Dhabaleswar K Panda. 2004. Efficient Barrier and Allreduce on Infiniband clusters using multicast and adaptive algorithms. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*. IEEE, 135–144.
[14] Srđan Milaković, Zoran Budimlić, Howard Pritchard, Anthony Curtis, Barbara Chapman, and Vivek Sarkar. 2018. SHCOLL-A Standalone Implementation of OpenSHMEM-Style Collectives API. In *Workshop on OpenSHMEM and Related Technologies*. Springer, 90–106.
[15] Rajesh Nishtala, Yili Zheng, Paul H Hargrove, and Katherine A Yelick. 2011. Tuning collective communication for Partitioned Global Address Space programming models. *Parallel Comput.* 37, 9 (2011), 576–591.
[16] Ying Qian and Ahmad Afsahi. 2008. Efficient shared memory and RDMA based collectives on multi-rail QsNet II SMP clusters. *Cluster Computing* 11, 4 (2008), 341–354.
[17] Rolf Rabenseifner. 2004. Optimization of collective reduction operations. In *International Conference on Computational Science*. Springer, 1–9.
[18] Open Source Software Solutions. [n. d.]. OpenSHMEM 1.4 Specification. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf
[19] Carlos Teijeiro, Guillermo L Taboada, Juan Touriño, Ramón Doallo, José C Mouriño, Damián A Mallón, and Brian Wibecan. 2013. Design and Implementation of an Extended Collectives Library for Unified Parallel C. *Journal of Computer Science and Technology* 28, 1 (2013), 72–89.
[20] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[21] Jesper Larsson Traff. 2004. Hierarchical gather/scatter algorithms with graceful degradation. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, 80.
[22] Xi Wang. [n. d.]. xBGAS Demo and Tutorial Video. https://www.youtube.com/watch?v=IeIpJkMjMuc&feature=youtu.be
[23] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. 2014. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0.* Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.
[24] Udayanga Wickramasinghe and Andrew Lumsdaine. 2016. A survey of methods for collective communication optimization and tuning. *arXiv preprint arXiv:1611.06334* (2016).
[25] Brody Williams. [n. d.]. xBGAS Collective Demo Video. https://www.youtube.com/watch?v=08CMiQ8XVnU&feature=youtu.be
[26] Joachim Worringen. 2003. Pipelining and overlapping for MPI collective operations. In *28th Annual IEEE International Conference on Local Computer Networks, 2003. LCN'03. Proceedings.* IEEE, 548–557.
[27] Changil Yoon, Vikas Aggarwal, Vrishali Hajare, Alan D George, and Max Billingsley III. 2011. GSHMEM: A portable library for lightweight, shared-memory, parallel programming. *Proc. of Partitioned Global Address Space, Galveston, Texas* (2011).