Asynchronous Many-Task Programming With OpenSHMEM[*]

Max Grossman, Vivek Kumar, Zoran Budimlić, and Vivek Sarkar

Rice University

Abstract. Partitioned Global Address Space (PGAS) programming models combine shared and distributed memory features, and provide a foundation for high-productivity parallel programming using lightweight one-sided communications. The *OpenSHMEM* programming interface has recently begun gaining popularity as a lightweight library-based approach for developing PGAS applications, in part through its use of a *symmetric heap* to realize more efficient implementations of global pointers than in other PGAS systems. However, current approaches to *hybrid* inter-node and intra-node parallel programming in OpenSHMEM rely on the use of multithreaded programming models (e.g., pthreads, OpenMP) that harness intra-node parallelism but are opaque to the OpenSHMEM runtime. This OpenSHMEM+X approach can encounter performance challenges such as bottlenecks on shared resources, long pause times due to load imbalances, and poor data locality. Furthermore, OpenSHMEM+X requires the expertise of hero-level programmers, compared to the use of just OpenSHMEM. All of these are hard challenges to mitigate with incremental changes. This situation will worsen as computing nodes increase their use of accelerators and heterogeneous memories. In this paper, we introduce the *AsyncSHMEM* PGAS library which supports a tighter integration of shared and distributed memory parallelism than past OpenSHMEM implementations. AsyncSHMEM integrates the existing OpenSHMEM reference implementation with a thread-pool-based, intra-node, work-stealing runtime. It aims to prepare OpenSHMEM for future generations of HPC systems by enabling the use of asynchronous computation to hide data transfer latencies, supporting tight interoperability of OpenSHMEM with task parallel programming, improving load balance (both of communication and computation), and enhancing locality. In this paper we present the design of AsyncSHMEM, and demonstrate the performance of our initial AsyncSHMEM implementation by performing a scalability analysis of two benchmarks on the Titan supercomputer. These early results are promising, and demonstrate that AsyncSHMEM is more programmable than the OpenSHMEM+OpenMP model, while delivering comparable performance for a regular benchmark (ISx) and superior performance for an irregular benchmark (UTS).

1 Introduction

Computing systems are rapidly moving toward exascale, requiring highly programmable means of specifying the communication and computation to be carried out

[*] This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication: The 3rd Workshop on OpenSHMEM and Related Technologies, August 2016, Baltimore, USA

by the machine. Because of the complexity of these systems, existing communication models for High Performance Computing (HPC) often run into performance and programmability limitations, as they can make it difficult to identify and exploit opportunities for computation-communication overlap. Existing communication models also lack tight integration with multi-threaded programming models, often requiring overly coarse or error-prone synchronization between the communication and multi-threaded components of applications.

Distributed memory systems with large amounts of parallelism available per node are notoriously difficult to program. Prevailing distributed memory approaches, such as MPI [23], UPC [11], or OpenSHMEM [7], are designed for scalability and communication. For certain applications they may not be well suited as a programming model for exploiting intra-node parallelism. On the other hand, prevailing programming models for exploiting intra-node parallelism, such as OpenMP [9], Cilk [12], and TBB [21] are not well suited for use in a distributed memory environment as the parallel programming paradigms used (tasks or groups of tasks, parallel loops, task synchronization) do not translate well or easily to a distributed memory environment.

The dominant solution to this problem so far has been to combine the distributed-memory and shared-memory programming models into ”X+Y”, e.g., MPI+OpenMP or OpenSHMEM+OpenMP. While such approaches to *hybrid* inter-node and intra-node parallel programming are attractive as they require no changes to either programming model, they also come with several challenges. First, the programming concepts for inter- and intra-node parallelism are often incompatible. For example, MPI communication and synchronization within OpenMP parallel regions may have undefined behavior. This forces some restrictions on how constructs can be used (for example, forcing all MPI communication to be done outside of the OpenMP parallel regions). Second, the fact that each runtime is unaware of the other can lead to performance or correctness problems (e.g. overly coarse-grain synchronization or deadlock) when using them together. Third, in-depth expertise in either distributed memory programming models or shared-memory programming models is rare, and expertise in both even more so. Fewer and fewer application developers are able to effectively program these hybrid software systems as they become more complex.

In this paper we propose AsyncSHMEM, a unified programming model that integrates Habanero tasking concepts [8] with the OpenSHMEM PGAS model. The Habanero tasking model is especially suited for this kind of implementation, since its asynchronous nature allows OpenSHMEM communication to be treated as tasks in a unified runtime system. AsyncSHMEM allows programmers to write code that exploits intra-node parallelism using Habanero tasks and distributed execution/communication using OpenSHMEM. AsyncSHMEM includes extensions to the OpenSHMEM specification for asynchronous task creation, extensions for tying together OpenSHMEM communication and Habanero tasking, and a runtime implementation that performs unified computation and communication scheduling of AsyncSHMEM programs.

We have implemented and evaluated two different implementations of the AsyncSHMEM interface. The first is referred to as the *Fork-Join approach* and is a lightweight integration of our task-based, multi-threaded runtime with the OpenSHMEM runtime with constraints on the programmer similar to those imposed by an OpenSH-

MEM+OpenMP approach. The second is referred to as the *Offload approach* and offers a tighter integration of the OpenSHMEM and tasking runtimes that permits OpenSHMEM calls to be performed from within parallel tasks. The runtime ensures that all OpenSHMEM operations are offloaded to a single runtime thread before calling in to the OpenSHMEM runtime. The Fork-Join approach offers small overheads but a more complicated programming model and is more restrictive in the use of the OpenSHMEM tasking API extensions. The Offload approach ensures that all OpenSHMEM operations are issued from a single thread, removing the need for a thread-safe OpenSHMEM implementation. We note that this *communication thread* is not dedicated exclusively to OpenSHMEM operations, and is also used to execute user-created computational tasks if needed. The advantage of the Offload approach is that it supports a more flexible and intuitive programming model than the Fork-Join approach, and can also support higher degrees of communication-computation overlap.

The main contributions of this paper are as follows:

- The definition of the AsyncSHMEM programming interface, with extensions to OpenSHMEM to support asynchronous tasking.
- Two runtime implementations for AsyncSHMEM that perform unified computation and communication scheduling of AsyncSHMEM programs.
- A preliminary performance evaluation and comparison of these two implementations with flat OpenSHMEM and OpenSHMEM+OpenMP models, using two different applications and scaling them up to 16K cores on the Titan supercomputer.

The rest of the paper is organized as follows. Section 2 provides background on the Habanero tasking model that we use as inspiration for the proposed OpenSHMEM tasking extensions, as well as the OpenSHMEM PGAS programming model. Section 3 describes our extensions to the OpenSHMEM API specification and our two implementations of the AsyncSHMEM runtime in detail. Section 4 explains our experimental methodology. Section 5 presents and discusses experimental results comparing the performance of our two AsyncSHMEM implementations against OpenSHMEM and OpenSHMEM+OpenMP implementations of two benchmarks, UTS and ISx. This is followed by a discussion of related work in Section 6. Finally, Section 7 concludes the paper.

2 Background

In this section we describe the programming concepts and existing implementations that serve as the foundation for the hybrid AsyncSHMEM model: Habanero Tasking and OpenSHMEM.

2.1 Habanero Tasking

The Habanero task-parallel programming model [5] offers an `async-finish` API for exploiting intra-node parallelism. The Habanero-C Library (HClib) is a native library-based implementation of the Habanero programming model that offers C and C++ APIs.

Here we briefly describe relevant features of both the abstract Habanero programming model and its HClib implementation. More details can be found in [22].

The Habanero `async` construct is used to create an asynchronous child task of the current task executing some user-defined computation. The `finish` construct is used to join all child `async` tasks (including any transitively spawned tasks) created inside of a logical scope. The `forasync` construct offers a parallel loop implementation which can be used to efficiently create many parallel tasks.

The Habanero model also supports defining dependencies between tasks using standard parallel programming constructs: promises and futures. A *promise* is a write-only value container which is initially empty. In the Habanero model, a promise can be satisfied once by having some value placed inside of it by any task. Every promise has a *future* associated with it, which can be used to read the value stored in the promise. At creation time tasks can be declared to be dependent on the satisfaction of a promise by registering on its future. This ensures that a task will not execute until that promise has been satisfied. In Habanero, the `asyncAwait` construct launches a task whose execution is predicated on a user-defined set of futures. User-created tasks can also explicitly block on futures while executing.

In the Habanero model, a `place` can be used to specify a hardware node within a hierarchical, intra-node place tree [24]. The `asyncAt` construct accepts a `place` argument, and creates a task that must be executed at that `place`.

HClib is a C/C++ library implementation that implements the abstract Habanero programming model. HClib sits on top of a multi-threaded, work-stealing, task-based runtime. HClib uses lightweight, runtime-managed stacks from the Boost Fibers [16] library to support blocking tasks without blocking the underlying runtime worker threads. Past work has shown HClib to be competitive in performance with industry-standard multi-threaded runtimes for a variety of workloads [13].

HClib serves as the foundation for the intra-node tasking implementation of AsyncSHMEM described in this paper.

2.2 OpenSHMEM

SHMEM is a communication library used for Partitioned Global Address Space (PGAS) [20] style programming. The SHMEM communications library was originally developed as a proprietary application interface by Cray for their T3D systems [15]. Since then different vendors have come up with variations of the SHMEM library implementation to match their individual requirements. These implementations have over the years diverged because of the lack of a standard specification. OpenSHMEM [7] is an open source community effort to unify all SHMEM library development effort.

3 AsyncSHMEM

In this section we present proposed API extensions to the OpenSHMEM specification, as well as two runtime implementations of those extensions.

3.1 API Extensions

The existing OpenSHMEM specification focuses on performing communication to and from processing elements (PEs) in a PGAS communication model. This work extends the OpenSHMEM specification with APIs for both creating asynchronously executing tasks as well as declaring dependencies between communication and computation. In this section, we briefly cover the major API extensions. Due to space limitations, these descriptions are not intended to be a comprehensive specification of these new APIs.

In general, the semantics of OpenSHMEM APIs in AsyncSHMEM are the same as any specification-compliant OpenSHMEM runtime. For collective routines, we expect that only a single call is made from each PE. The ordering of OpenSHMEM operations coming from independent tasks must be ensured using task-level synchronization constructs. For example, if a programmer requires that a `shmem_fence` call is made between two OpenSHMEM operations occurring in other tasks, it is their responsibility to ensure that the inter-task dependencies between those tasks ensure that ordering. The atomicity of atomic OpenSHMEM operations is guaranteed relative to other PEs as well as relative to all threads.

```
void shmem_task_nbi(void (*body)(void *), void *user_data);
```

`shmem_task_nbi` creates an asynchronously executing task defined by the user function `body` which is passed `user_data` when launched by the runtime.

```
void shmem_parallel_for_nbi(void (*body)(int, void *),
        void *user_data, int lower_bound, int upper_bound);
```

`shmem_parallel_for_nbi` provides a one-dimensional parallel loop construct for AsyncSHMEM programs, where the bounds of the parallel loop are defined by `lower_bound` and `upper_bound`. Each iteration of the parallel loop executes `body` and is passed both its iteration index and `user_data`.

```
void shmem_task_scope_begin();
void shmem_task_scope_end();
```

A pair of `shmem_task_scope_begin` and `shmem_task_scope_end` calls are analogous to a `finish` scope in the Habanero task parallel programming model. `shmem_task_scope_end` blocks until all transitively spawned child tasks since the last `shmem_task_scope_begin` have completed.

```
void shmem_task_nbi_when(void (*body)(void *), void *user_data,
        TYPE *ivar, int cmp, TYPE cmp_value);
```

The existing OpenSHMEM Wait APIs allow an OpenSHMEM PE to block and wait for a value in the symmetric heap to meet some condition. The `shmem_task_nbi_when` API is similar, but rather than blocking makes the execution of an asynchronous task predicated on a condition. This is similar to the concept of promises and futures introduced in Section 2. This API also allows remote communication to create local work on a PE.

3.2 Fork-Join Implementation

The Fork-Join approach is an implementation of AsyncSHMEM that supports most of the proposed extensions from Section 3.1. It is open source and available at `https://github.com/openshmem-org/openshmem-async`.

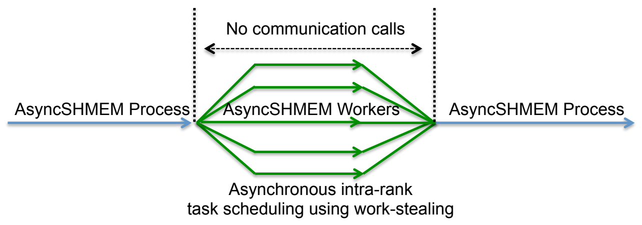


Fig. 1. Fork-Join asynchronous task programming model in OpenSHMEM. The intra-rank asynchronous child tasks cannot make any communication calls.

This particular implementation of AsyncSHMEM integrates asynchronous task parallelism wihout making any changes to the core OpenSHMEM runtime. Changes are limited to the user-level API's in OpenSHMEM. The goal of the Fork-Join implementation was to study the impact of supporting basic asynchronous tasking in OpenSHMEM. In this approach, only the main thread (or process) is allowed to perform OpenSHMEM communication operations (blocking puts and gets, collectives). The asynchronous child tasks are not allowed to perform communication. The main thread can create child tasks by calling `shmem_task_nbi` or `shmem_parallel_for_nbi`. These child tasks can further create arbitrarily nested tasks. Synchronization over these tasks can be achieved either by explicitly creating task synchronization scopes by using `shmem_task_scope_begin` and `shmem_task_scope_end`, or implicitly by calling `shmem_barrier_all`. The `shmem_init` call starts a top-level synchronization scope by calling `shmem_task_scope_begin` internally. Each `shmem_barrier_all` call includes an implicit sequence of `shmem_task_scope_end` and `shmem_task_scope_begin` calls, i.e., it first closes the current synchronization scope and then starts a new scope. The call to `shmem_finalize` internally calls `shmem_task_scope_end` to close the top-level synchronization scope. The programmer is allowed to create arbitrarily nested task synchronization scopes using `shmem_task_scope_begin` and `shmem_task_scope_end`. We call this implementation of AsyncSHMEM a *Fork-Join approach* because of the implicit task synchronization scopes integrated inside the call to `shmem_barrier_all`, causing a join at each barrier but allowing the forking of asynchronous tasks between barriers. A typical usage of this implementation is shown in Figure 1, which closely mirrors an OpenSHMEM+OpenMP based hybrid programming model.

3.3 Offload Implementation

Similarly to the Fork-Join approach, the Offload approach does not require modifications to any existing OpenSHMEM implementations but does support a tighter integration of PGAS and task parallel programming with more flexible APIs. The Offload implementation is open source and available at `https://github.com/habanero-rice/hclib/tree/resource_workers/modules/openshmem`.

Similar to [17] and [8], the Offload implementation ensures all OpenSHMEM operations are issued by a single worker thread in the multi-threaded, work-stealing runtime. However, the Offload approach differs in that no worker thread is dedicated exclusively to performing communication. Instead, the communication worker thread is free to execute user-written computation tasks if no communication work can be found.

To better illustrate the Offload approach, we will walk through the execution of a `shmem_int_put` operation in the Offload approach's runtime:

1. An arbitrary task in a given PE calls the OpenSHMEM `shmem_int_put` API as usual, but using the AsyncSHMEM library. Under the covers, this call results in the creation of a task that wraps a call to the `shmem_int_put` API of an OpenSHMEM implementation. That task is placed on the work-stealing deque of the communication worker. No threads are allowed to steal communication tasks from the communication worker.
2. Because `shmem_int_put` is a blocking operation, the stack of the currently executing task is saved as a continuation and its execution is predicated on the completion of the created `shmem_int_put` task. The worker thread that performed this OpenSHMEM operation is then able to continue executing useful work even while the `shmem_int_put` operation is incomplete.
3. At some point in the future, the communication worker thread discovers an OpenSHMEM operation has been placed in its work-stealing deque, picks it up, and performs the actual `shmem_int_put` operation using an available OpenSHMEM implementation. If the communication worker thread has no communication to perform, it behaves just as any other worker thread in the runtime system by executing user-written computation tasks.
4. Once this communication task has completed on the communication worker thread, the continuation task's dependency is satisfied and it is made eligible for execution again.

Unlike the Fork-Join approach, this approach places no limitations on where OpenSHMEM calls can be made. This flexibility comes at the cost of increased runtime complexity. For example, OpenSHMEM locks must be handled carefully. If two independent tasks on the same node are locking the same OpenSHMEM lock, naive offload of lock operations can easily lead to deadlock scenarios. Instead, lock operations targeting the same lock object are chained using futures to ensure only a single task in each node tries to enter the lock at a time.

4 Experimental Methodology

Before detailing our experimental results with AsyncSHMEM, we first explain our experimental methodology in this section.

4.1 Benchmarks

We have used the following two benchmarks for evaluation of AsyncSHMEM: a) Integer Sorting (ISx) [14], and b) Unbalanced Tree Search (UTS) [19].

ISx: ISx is a scalable integer sorting benchmark that was inspired by the NAS Parallel Benchmark integer sort. It uses a parallel bucket sorting algorithm. The reference implementation of ISx uses OpenSHMEM only. To ensure a fair comparison, we also implement an OpenSHMEM+OpenMP version of ISx as part of this work. The OpenSHMEM+OpenMP and AsyncSHMEM versions of ISx are identical and simply replace OpenMP loop parallelism with `shmem_parallel_for_nbi`. Our experiments use the weak scaling version of ISx. In the OpenSHMEM version, the total number of sorting keys per rank is 2^{25}, whereas in both multi-threaded versions it is $\text{N}\times 2^{25}$, where N is the total number of threads per rank. Hence, across all versions of ISx, the total number of keys per node on Titan is 2^{29}.

UTS: The UTS benchmark performs the parallel traversal of a randomly generated unbalanced tree. The reference UTS implementation only includes OpenSHMEM parallelism, so as part of this work we implement an AsyncSHMEM version, an OpenSHMEM+OpenMP version, and an OpenSHMEM+OpenMP Tasks version. The OpenSHMEM+OpenMP Tasks and AsyncSHMEM versions are nearly identical in structure, using tasking APIs to cleanly express the recursive, irregular parallelism of UTS. The OpenSHMEM+OpenMP implementation is a heavily hand-optimized SPMD implementation, for which the development time was much greater.

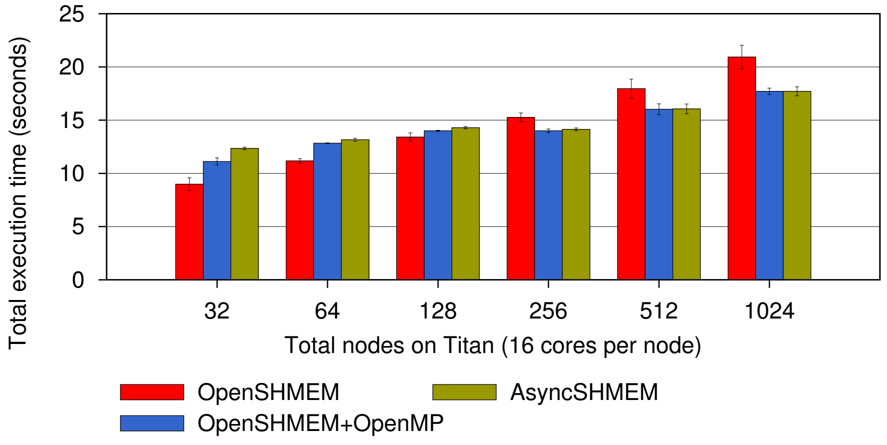
4.2 Experimental Infrastructure and Measurements

We performed all experiments on the Titan supercomputer at the Oak Ridge National Laboratory. This is a Cray XK7 system with each node containing an AMD Opteron 6274 CPU. There are two sockets per node (8 cores per socket) and an NVIDIA Tesla K20X GPU. For each OpenSHMEM-only version of benchmarks, we allocate one rank per core, i.e., 16 ranks per node for ISx and one rank per node for UTS. In both AsyncSHMEM and OpenMP versions we allocate one rank per socket with 8 threads per rank for ISx and one rank per node with 16 threads for UTS. We do not make use of the GPUs in these experiments, though our proposed changes do not affect the ability of OpenSHMEM to use GPU accelerators. Prior studies have found that on Cray supercomputers a communication heavy job can vary in performance across different job launches due to node allocation policies and other communication intensive jobs running in the neighborhood [4]. To ensure fair comparison across different versions of benchmark, we run each version as a part of single job launch on a given number of nodes.

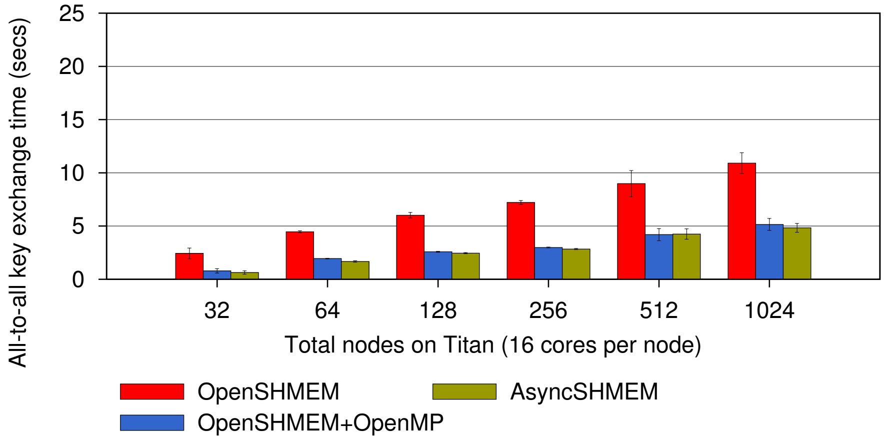
5 Results

5.1 ISx

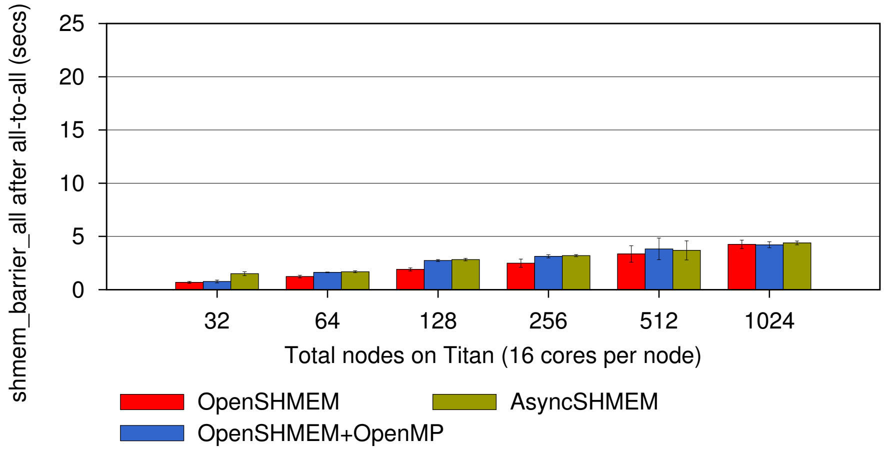
In this section we perform weak scaling experiments (details in Section 4) of all three versions of ISx using the Fork-Join implementation of AsyncSHMEM. The results of



(a) Total execution time



(b) Time spent in all-to-all key exchange



(c) Time spent inside `shmem_barrier_all` call after all-to-all key exchange

Fig. 2. Weak scaling of ISx with total number of keys per node remaining constant in each version

this experiment are shown in Figure 2. Figure 2(a) shows the total execution time (computation and communication) at each node count. Figure 2(b) shows the time spent in the all-to-all key exchange communication call. Figure 2(c) shows the time spent in barrier synchronization (`shmem_barrier_all`) after the all-to-all communication call for key exchange. Total communication time is the sum of time spent in all-to-all and `shmem_barrier_all` calls.

From Figure 2, we can see that at large node counts (256, 512 and 1024), both threaded versions of ISx (AsyncSHMEM and OpenMP) are relatively faster than the reference single threaded OpenSHMEM version. However, at smaller node counts (32 and 64 nodes), the reference OpenSHMEM version shows better performance than the threaded versions. These variations are due to NUMA effects as well as the time spend in all-to-all communication. ISx is a memory intensive application. Both threaded versions running with 8 threads per rank (one rank per socket) use $8\times$ more memory per rank than the single threaded reference version that uses 8 ranks per socket. Titan nodes have NUMA architecture. We used local allocation policy that favors memory allocations on the NUMA domain the rank is executing. This is more beneficial for the single threaded reference version, while in the threaded version the threads running on different NUMA domain will contend for the same memory locations. Due to the relatively fast key exchange time at 32 and 64 nodes (Figure 2(b)), memory access advantage of the reference OpenSHMEM version outweights the communication reduction of the threaded versions. With the increase in number of nodes, OpenSHMEM version of ISx has a much higher number of ranks participating in the all-to-all communication than the AsyncSHMEM and OpenMP versions, resulting in large communication cost.

5.2 UTS

Relative to ISx, UTS is a more irregular application which further stresses the intra-node load balancing and inter-node communication-computation overlap of AsyncSHMEM. For these experiments, we investigate the strong scaling of UTS on the provided T1XXL dataset to demonstrate the improvement in computation-communication overlap achievable using AsyncSHMEM.

Figure 3 plots the overall performance of UTS using OpenSHMEM, OpenSHMEM+OpenMP, OpenSHMEM+OpenMP Tasks, and AsyncSHMEM.

As expected, the reference OpenSHMEM implementation performs worst as we only use a single PE per node[1].

Because of the lack of integration between OpenSHMEM and OpenMP, the OpenSHMEM+OpenMP Tasks implementation also performs slowly as coarse-grain synchronization is required to join all tasks before performing distributed load balancing using OpenSHMEM. Despite using $16\times$ as many cores as the OpenSHMEM reference implementation, the OpenSHMEM+OpenMP Tasks version generally only completes $2\times$ faster.

Our optimized OpenSHMEM+OpenMP implementation performs similarly to AsyncSHMEM, though shows worse scalability beyond 128 nodes. We also note that

[1] Running multiple PEs per node in this environment with the reference OpenSHMEM implementation caused hangs which we were unable to diagnose

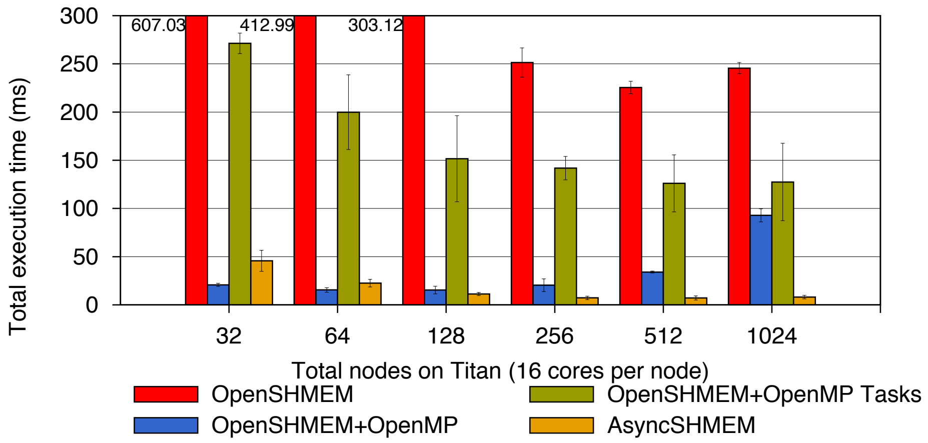


Fig. 3. Strong scaling of UTS on the T1XXL dataset.

it took significantly more development effort to build an efficient version of UTS using SPMD-style OpenSHMEM+OpenMP.

As part of our UTS implementation, we explored using more complex techniques for distributed load balancing, as this is one of the primary bottlenecks for UTS performance. In particular, we experimented with using the proposed `shmem_task_nbi_when` extension to allow PEs to alert other PEs when work was available to be stolen in the hope that load balancing could occur in the background rather than in bulk-synchronous fashion. The challenge with this approach appears to lie in designing a `shmem_task_nbi_when` implementation that balances low latency between a symmetric variable being modified and the dependent task being launched with overheads from checking symmetric variable values. In our initial implementation of this API, we were unable to find an appropriate balance between these two and so UTS implementations that took advantage of more novel APIs were not able to out-perform or out-scale more conventional implementations.

6 Related Work

6.1 Combining Distributed Programming Models with Task-Parallel Programming

The *Partitioned Global Address Space* (PGAS) programming model [25] strikes a balance between shared and distributed memory models. It combines the ease of programming with a global address space with performance improvements from locality awareness. PGAS languages include Co-Array Fortran [18], Titanium [26], UPC [11], X10 [10] and Chapel [6]. These languages rely on compiler transformations to convert user code to native code. Some of these languages, such as Titanium, X10 and Chapel, use code transformations to provide dynamic tasking capabilities using a work-stealing scheduler for load balancing of the dynamically spawned asynchronous tasks.

Another related piece of work is HCMPI [8], a language-based implementation which combines MPI communication with Habanero tasking using a dedicated communication worker (similar to the Offload approach).

Language-based approaches to hybrid multi-node, multi-threaded programming have some inherent disadvantages relative to library-based techniques. Users have to first learn a new language, which often does not have mature debugging or performance analysis tools. Language-based approaches are also associated with significant development and maintenance costs. To avoid these shortcomings HabaneroUPC++ [17] introduced a compiler-free PGAS library that supports integration of intra-node and inter-node parallelism. It uses the UPC++ [27] library to provide PGAS communication and function shipping, and the C++ interface of the HClib library to provide intra-rank task scheduling. HabaneroUPC++ uses C++11 lambda-based user interfaces for launching asynchronous tasks.

6.2 Thread-Safe OpenSHMEM Proposals

Recently, the OpenSHMEM Threading Committee has been exploring extensions to the OpenSHMEM specification to support its use in multi-threaded environments on multi-core systems. Discussions in the OpenSHMEM Threading Committee have focused on three approaches to adding the concept of thread-safety to the OpenSHMEM specification. While AsyncSHMEM is not a thread-safe extension to OpenSHMEM per se, it has the same high-level goal as these thread-safety proposals: improving the usability and performance of OpenSHMEM programs on multi-core platforms.

One proposal would make the entire OpenSHMEM runtime thread-safe by ensuring any code blocks that share resources are mutually exclusive. While this proposal is powerful in its simplicity and would have minimal impact on the existing OpenSHMEM APIs, the overheads from full thread-safety could quickly become a performance bottleneck for future multi-threaded OpenSHMEM applications. This proposal is summarized in Issue #218 on the OpenSHMEM Redmine [2]. Today, this proposal is orthogonal to the work on AsyncSHMEM. Because AsyncSHMEM serializes all OpenSHMEM communication through a single thread, any concurrent data structures within the OpenSHMEM implementation itself would only add unnecessary overhead. However, if in the future we were to explore multiple communication worker threads in the Offload approach then this thread-safety proposal would be one way to enable that work.

The second proposal would introduce the concept of thread registration to OpenSHMEM, in which any thread that wishes to make OpenSHMEM calls would have to register itself with the OpenSHMEM runtime. The runtime would be responsible for managing any thread-private or shared resources among registered threads. This proposal would also have minimal impact on the existing OpenSHMEM APIs, simply requiring that programmers remember to register threads before making any OpenSHMEM calls. Explicit thread registration would enable better handling of multi-threaded programs by the OpenSHMEM runtime, likely leading to improved performance than the simple thread-safety proposal. This proposal was put forward by Cray, and is summarized in [3]. Similar to the first simple thread safety proposal, this thread registration proposal is orthogonal to AsyncSHMEM until we consider multiple communication worker threads in the Offload approach.

The third proposal focuses on adding the idea of an OpenSHMEM context to the OpenSHMEM specification. A context would encapsulate all of the resources necessary to issue OpenSHMEM operations, and it would be the programmer's responsibility to ensure only a single thread operates on a context at a time. However, different threads could use different contexts to issue OpenSHMEM operations in parallel. This proposal would be the most disruptive to the existing OpenSHMEM specification and requires the most programmer effort, but could also benefit both multi- and single-threaded OpenSHMEM applications by enabling the creation of multiple independent communication streams. This proposal was made by Intel, and is summarized in [1]. Unlike the previous two proposals, OpenSHMEM contexts could be useful in conjuction with AsyncSHMEM. Contexts would enable AsyncSHMEM to keep multiple streams of communication in-flight at once as long as no ordering constraints (e.g. via `shmem_fence`) prevented that.

The main way in which AsyncSHMEM differentiates itself is by being a complete extension to the OpenSHMEM specification, adding the concept of intra-node parallelism to OpenSHMEM's existing inter-node parallelism. This integration enables a better performing runtime implementation as well as the exploration of other novel APIs, such as `shmem_task_nbi_when`. However, the three thread-safety proposals above are more general in that they enable combining OpenSHMEM with any multi-threading programming model (e.g. OpenMP, pthreads, Cilk).

7 Conclusion

In this paper we present work on integrating task-parallel, multi-threaded programming models with the OpenSHMEM PGAS communication model. We present extensions to the OpenSHMEM specification to enable the creation of asynchronous, intra-node tasks and to allow local computation to be dependent on remote communication. We describe and implement two different approaches to implementing these extensions: the Fork-Join and Offload approaches. The Fork-Join approach is simple, but is similar to existing OpenSHMEM+X approaches in its limitations on the use of computation and communication APIs together. The Offload approach requires more complex run-time support, but offers more flexibility in how tasks and communication can be used together.

Our experimental evaluation shows that AsyncSHMEM performs similarly to existing OpenSHMEM+X approaches for regular applications and outperforms them for more irregular workloads. In our experience, the flexibility of the Offload approach also dramatically improves application programmability and maintainability.

There are many future directions for this work. We plan to focus development efforts on the Offload implementation, as the programmability and flexibility benefits it offers make it a better candidate for exploring more novel task-based extensions to the OpenSHMEM specification. We will perform more in-depth analysis of the performance characteristics of the ISx, UTS, and other benchmarks running on the Offload implementation. This investigation will focus on both quantifying overheads introduced by our implementation as well as pinpointing benefits.

Acknowledgments

This research was funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory.

References

1. OpenSHMEM context extension proposal draft, https://github.com/jdinan/openshmem-contexts
2. OpenSHMEM Redmine Issue #218 - Thread Safety Proposal, http://www.openshmem.org/redmine/issues/218
3. Thread-safe SHMEM Extensions, http://www.csm.ornl.gov/workshops/openshmem2014/documents/Thred-safeSHMEM_Extensions.pdf
4. Bhatele, A., Mohror, K., Langer, S.H., Isaacs, K.E.: There goes the neighborhood: Performance degradation due to nearby jobs. In: SC. pp. 41:1–41:12. ACM (2013)
5. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the New Adventures of Old X10. In: PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (2011)

6. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (Aug 2007)
7. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing openshmem: Shmem for the pgas community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. p. 2. ACM (2010)
8. Chatterjee, S.: Integrating Asynchronous Task Parallelism with MPI. In: IPDPS '13: Proceedings of the 2013 IEEE International Symposium on Parallel&Distributed Processing. IEEE Computer Society (2013)
9. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng. 5(1), 46–55 (Jan 1998)
10. Ebcioglu, K., Saraswat, V., Sarkar, V.: X10: an experimental language for high productivity programming of scalable systems. In: Proceedings of the Second Workshop on Productivity and Performance in High-End Computing. pp. 45–52. Citeseer (2005)
11. El-Ghazawi, T., Smith, L.: UPC: unified parallel C. In: SC (2006)
12. Frigo, M.: Multithreaded programming in Cilk. In: PASCO '07. pp. 13–14 (2007)
13. Grossman, M., Shirako, J., Sarkar, V.: Openmp as a high-level specification language for parallelism. In: IWOMP'16 (2016)
14. Hanebutte, U., Hemstad, J.: Isx: A scalable integer sort for co-design in the exascale era. In: Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on. pp. 102–104 (Sept 2015)
15. Kessler, R.E., Schwarzmeier, J.L.: Cray T3D: A new dimension for Cray research. In: Compcon Spring'93, Digest of Papers. pp. 176–182. IEEE (1993)
16. Kowalke, O.: Boost C++ Libraries, `https://olk.github.io/libs/fiber/doc/html/`
17. Kumar, V., Zheng, Y., Cavé, V., Budimlić, Z., Sarkar, V.: HabaneroUPC++: A compiler-free PGAS library. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 5:1–5:10. PGAS '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2676870.2676879`
18. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. SIGPLAN Fortran Forum 17(2), 1–31 (Aug 1998)
19. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: an unbalanced tree search benchmark. In: LCPC. pp. 235–250 (2007)
20. PGAS: Partitioned Global Address Space. `http://www.pgas.org/` (2011)
21. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism (2010)
22. Habanero-C Overview. `https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C` (Rice University, 2013)
23. Snir, M., Otto, S.W., Walker, D.W., Dongarra, J., Huss-Lederman, S.: MPI: The Complete Reference (1995)
24. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In: LCPC'09: Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science, vol. 5898. Springer (2009)
25. Yelick, K.e.a.: Productivity and performance using partitioned global address space languages. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation. pp. 24–32. PASCO '07, ACM (2007)
26. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. In: In ACM. pp. 10–11 (1998)
27. Zheng, Y., Kamil, A., Driscoll, M.B., Shan, H., Yelick, K.: UPC++: a PGAS extension for C++. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. pp. 1105–1114. IEEE (2014)