CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:5288–5310
Published online 25 June 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3569

Low-level PGAS computing on many-core processors with TSHMEM[§]

Bryant C. Lam[*,†], Alan D. George, Herman Lam and Vikas Aggarwal[‡]

NSF Center for High-Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA

SUMMARY

Diminishing returns from increased clock frequencies and instruction-level parallelism have forced computer architects to adopt architectures that exploit wider parallelism through multiple processor cores. While emerging many-core architectures have progressed at a remarkable rate, concerns arise regarding the performance and productivity of numerous parallel-programming tools for application development. Development of parallel applications on many-core processors often requires developers to familiarize themselves with unique characteristics of a target platform while attempting to maximize performance and maintain correctness of their applications. The family of partitioned global address space (PGAS) programming models comprises the current state of the art in balancing performance and programmability. One such PGAS approach is SHMEM, a lightweight, shared-memory programming library that has demonstrated high performance and productivity potential for parallel-computing systems with distributed-memory architectures. In the paper, we present research, design, and analysis of a new SHMEM infrastructure specifically crafted for low-level PGAS on modern and emerging many-core processors featuring dozens of cores and more. Our approach (with a new library known as TSHMEM) is investigated and evaluated atop two generations of Tilera architectures, which are among the most sophisticated and scalable many-core processors to date, and is intended to enable similar libraries atop other architectures now emerging. In developing TSHMEM, we explore design decisions and their impact on parallel performance for the Tilera TILE-Gx and TILE*Pro* many-core architectures, and then evaluate the designs and algorithms within TSHMEM through microbenchmarking and applications studies with other communication libraries. Our results with barrier primitives provided by the Tilera libraries show dissimilar performance between the TILE-Gx and TILE*Pro*; therefore, TSHMEM's barrier design takes an alternative approach and leverages the on-chip mesh network to provide consistent low-latency performance. In addition, our experiments with TSHMEM show that naive collective algorithms consistently outperformed linear distributed collective algorithms when executed in an SMP-centric environment. In leveraging these insights for the design of TSHMEM, our approach outperforms the OpenSHMEM reference implementation, achieves similar to positive performance over OpenMP and OSHMPI atop MPICH, and supports similar libraries in delivering high-performance parallel computing to emerging many-core systems.

Received 20 August 2014; Revised 17 May 2015; Accepted 17 May 2015

KEY WORDS: many-core; PGAS; SHMEM; OpenSHMEM; OpenMP; parallel programming; performance analysis; high-performance computing

1. INTRODUCTION

Parallel programming is experiencing explosive growth in demand because of processor architectures shifting toward many processing cores in an effort to maintain performance progression,

*Correspondence to: Bryant C. Lam, NSF CHREC Center, Room 317, Larsen Hall, University of Florida, Gainesville, FL 32611, USA.

†E-mail: blam@chrec.org

‡Current address: Oracle Labs, Belmont, CA 94002.

§Extension of Conference Paper: An earlier version of this work featured a preliminary design of TSHMEM [1] and its performance with several parallel application kernels [2]. This work expands on those publications with new collective algorithms in TSHMEM and their performance; microbenchmark results for TSHMEM, the OpenSHMEM reference implementation, and an implementation of OpenSHMEM atop MPI-3 (OSHMPI) on the TILE-Gx; new performance numbers featuring MDE version 4.2.2 on TILE-Gx; and an expanded case study with SHMEM and OpenMP applications.

especially in the face of technological and physical limitations. With the emergence of many-core processors into the high-performance computing (HPC) scene, there is strong interest in evaluating and evolving existing parallel-programming models, tools, and libraries. This evolution is necessary to best exploit the increasing single-device parallelism from multi-core and many-core processors, especially in a field focused on massively distributed supercomputers.

High-performance computing has traditionally focused on models such as message passing with MPI [3] or shared memory with OpenMP [4], but interest is rising for a partitioned global address space (PGAS) abstraction with its potential to provide high-performing libraries and languages around a straightforward memory and communication model. Notable members of the PGAS family include SHMEM [5, 6], Unified Parallel C, Global Arrays, Co-Array Fortran, Titanium, GASPI, MPI-3 RMA [7], X10, and Chapel.

In the paper, we present research, design, and analysis for a new SHMEM infrastructure for low-level PGAS semantics on modern and emerging many-core processors. We approach our investigation and evaluation with a new SHMEM library known as TSHMEM [1], based on the OpenSHMEM version 1.0 specification, with the intended objective of exploring SHMEM and PGAS semantics on many-core processors and enable similar libraries to fully leverage these emerging devices. TSHMEM serves as the basis for our performance evaluation of communication algorithms as they pertain to SHMEM functionality, with focus on design exploration and maximizing the capabilities of the Tilera TILE-Gx and TILE*Pro* many-core architectures. While exploring the design decisions that define TSHMEM, we strive to achieve high realizable performance via microbenchmarking and application studies, comparing results with alternative libraries and programming environments. In doing so, TSHMEM aims to deliver a high-performance, many-core programming library that offers insights into performance for a variety of communication algorithms in the context of highly parallel, many-core processors.

The remainder of the paper is organized as follows. Section 2 provides background on the SHMEM library and standardization efforts via OpenSHMEM, our previous research with GSHMEM (i.e., SHMEM for clusters), a synopsis of OpenMP, and a brief introduction to Tilera's many-core architectures. Section 3 presents several microbenchmarking results on Tilera TILE-Gx8036 and TILE*Pro*64 processors. Section 4 delves into the design of TSHMEM, with performance results and analysis for functionality defined by the OpenSHMEM specification. Section 5 presents several application studies with TSHMEM performance. Finally, Section 6 provides conclusions and directions for future work.

2. BACKGROUND

The single-program, multiple-data (SPMD) programming style is highly amenable for tasks on large parallel systems, enabling diverse programming models such as active message passing, distributed shared memory, and PGAS. This section provides a brief background of SHMEM, GSHMEM, and Tilera, which form the foundation of our experience and design for TSHMEM. A synopsis of OpenMP is also provided, as it serves as one of the parallel-programming environments that TSHMEM is compared with in Section 5.

2.1. SHMEM and OpenSHMEM

The SHMEM communication library adheres to a strict PGAS model whereby each cooperating parallel process (also known as a processing element, or PE) consists of a shared symmetric partition within the global address space. Each symmetric partition consists of symmetric objects (variables or arrays) of the same size, type, and relative address on all PEs. Originally developed to provide shared-memory semantics on the distributed-memory Cray T3D supercomputer, SHMEM closely models SPMD via its symmetric, partitioned, global address space.

There are two types of symmetric objects that can reside in the symmetric partitions: static and dynamic. Static variables reside in the heap segment of the program executable and are allocated during link time. These static variables, when parallelized as multiple processes, appear at the same virtual address to all processes running the same executable, thus ensuring its symmetry across

DOI: 10.1002/cpe

all partitions. Dynamic symmetric variables, in contrast, are allocated at runtime on all PEs via SHMEM's dynamic memory allocation function `shmalloc()`. These dynamic variables, however, may or may not be allocated at the same virtual address on all PEs, but are at the same offset relative to the start of each symmetric partition.

SHMEM provides several routines for explicit communication between PEs, including one-sided data transfers (puts and gets), blocking barrier synchronization, and collective operations, as illustrated by the basic subset of available routines listed in Table I. In addition to being a high-performance, lightweight library, SHMEM has historically provided for atomic memory operations not available in popular library alternatives until recently (e.g., MPI 3.0).

Commercial SHMEM implementations have emerged from vendors such as Cray, SGI, and Quadrics. Application portability between variants, however, proved difficult because of different functional semantics, incompatible APIs, or system-specific implementations. This situation had regrettably fragmented developer adoption in the HPC community. Fortunately, SHMEM has seen renewed interest in the form of OpenSHMEM, a community-led effort to create a standard specification for SHMEM functions and semantics [8]. Version 1.0 of the OpenSHMEM specification [9] has already seen research and industry adoption in various implementations: the OpenSHMEM reference implementation [10], MVAPICH2-X [11], OSHMPI [12], Portals-SHMEM [13], POSH (Paris-OpenSHMEM) [14], and through vendors such as SGI [5], Cray [15], and Mellanox [16].

Table I. Basic subset of OpenSHMEM functions.

Category	Example Functions
Environment	
Setup and initialization	start_pes()
Environment query	shmem_my_pe() shmem_n_pes()
Memory allocation	shmalloc(), shfree()
Data transfer	
Elemental put/get	shmem_int_p() shmem_int_g()
Block put/get	shmem_putmem() shmem_getmem()
Strided put/get	shmem_int_iput() shmem_int_iget()
Synchronization	
Barrier	shmem_barrier() shmem_barrier_all()
Communications sync	shmem_fence() shmem_quiet()
Point-to-point sync	shmem_wait() shmem_wait_until()
Collective communication	
Broadcast	shmem_broadcast32()
Collection	shmem_collect32() shmem_fcollect32()
Reduction	shmem_int_sum_to_all() shmem_long_prod_to_all()
Atomic operations	
Atomic Swap	shmem_swap()

DOI: 10.1002/cpe

2.2. GASNet and the OpenSHMEM reference implementation

The OpenSHMEM community has a reference implementation of their library with primary source-code contributions from the University of Houston and Oak Ridge National Laboratory [10]. This reference implementation is compliant with version 1.0 of the OpenSHMEM specification and is implemented atop GASNet [17], a low-level networking layer and communications middleware for supporting SPMD parallel-programming models such as PGAS. GASNet defines a core and an extended networking API that are implemented via conduits. These conduits enable support for numerous networking technologies and systems. By leveraging GASNet's conduit abstraction, the OpenSHMEM reference implementation is portable to numerous cluster-based systems.

2.3. GSHMEM

Our prior work with SHMEM involved the design and evaluation of an OpenSHMEM library called GatorSHMEM (GSHMEM) [18] atop GASNet [17]. GSHMEM targeted a draft version of the OpenSHMEM v1.0 specification in order to evaluate its existing functionality and propose several new additions for future revisions. Built for x86_64-based cluster systems, experimental results via microbenchmarking showed that GSHMEM performance is comparable with that of a proprietary Quadrics implementation of SHMEM and an MPI library (MVAPICH) over InfiniBand. Additionally, two application case studies with GSHMEM demonstrated the library's portability across two distinct systems with vastly disparate interconnection technologies. GSHMEM proved that, by leveraging GASNet, SHMEM implementations can be made modern and portable over different architectures and system hierarchies without sacrificing high-performance or developer productivity.

2.4. OSHMPI: OpenSHMEM using MPI-3

The MPI 3.0 represents a significant revision to the MPI standard by including support for one-sided communication and introducing new semantics for memory consistency and ordering [7]. Hammond, et al. developed an OpenSHMEM library [12] using MPI-3's one-sided, remote-memory-access (RMA) operations and demonstrated comparable results against other SHMEM implementations such as the OpenSHMEM reference implementation, MVAPICH2-X, and Portals-SHMEM. Of note, OSHMPI was able to outperform its competitors in the SMP intranode configuration, suggesting its suitability for platforms such as the TILE-Gx.

2.5. OpenMP

The OpenMP (Open Multi-Processing) specification defines a collection of library routines, compiler directives, and environment variables that enable application parallelization via multiple threads of execution [4]. Standardized in 1997, OpenMP has been widely adopted and is portable across multiple platforms.

OpenMP commonly exploits symmetric-multiprocessing (SMP) architectures by enabling both data-level and thread-level parallelism. Parallelization is typically achieved via a fork-and-join approach controlled by compiler directives whereby a master thread will fork several child threads when encountering an OpenMP parallelization section. The child threads may be assigned to different processing cores and operate independently, thereby sharing the computational load with the master. Threads are also capable of accessing shared-memory variables and data structures to assist computation. At the end of each parallel section, child threads are joined with the master thread and the parallel section closes. The master thread continues on with sequential code execution until another parallel section is encountered.

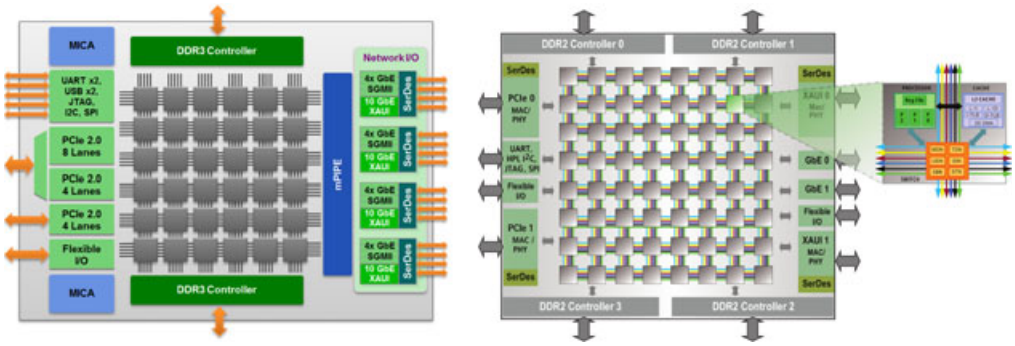
While other multi-threading APIs exist (e.g., POSIX threads), OpenMP is comparatively easier to use for developers that desire an incremental path to application parallelization for their existing sequential code. With the emergence of many-core processors such as Tilera's TILE-Gx and Intel's Xeon Phi, OpenMP is evolving to become a viable choice for single-device supercomputing tasks.

DOI: 10.1002/cpe

2.6. Tilera many-core processors

Tilera Corporation develops commercial many-core processors emphasizing high performance and low power in the cloud-computing, general, and embedded markets. Each Tilera many-core processor is designed as a scalable 2D mesh of tiles, with each tile consisting of a processing core and cache system attached to several on-chip networks via a non-blocking cut-through switch. Referred to as the Tilera iMesh (intelligent Mesh), their scalable 2D mesh consists of networks that provide data routing between memory controllers, caches, and external I/O and enables developers to explicitly transfer data between tiles via a low-level user-accessible dynamic network.

Our work focuses on the 36-core TILE-Gx8036 (Figure 1a) with its predecessor, the 64-core TILE*Pro*64 (Figure 1b), as a reference point for comparison. Their architectural characteristics are detailed in Table II. The TILE*Pro* is Tilera's previous generation of many-core processors with 32-bit processing cores interconnected via four dynamically dimension-order-routed networks and one developer-defined statically routed network. In contrast, the TILE-Gx is Tilera's current generation of 64-bit many-core processors. Differentiated by a substantially redesigned architecture, the TILE-Gx family exhibits upgraded processing cores and improved iMesh interconnects attached to five dynamic networks between the tiles and I/O. The TILE-Gx also includes hardware accelerators not found on previous Tilera processors: mPIPE (multicore Programmable Intelligent Packet Engine) for wire-speed packet classification, distribution, and load balancing; and MiCA (Multicore iMesh Coprocessing Accelerator) for cryptographic and compression acceleration. Other members of the TILE-Gx family include the 9-core TILE-Gx8009, 16-core TILE-Gx8016, and 72-core TILE-Gx8072.



(a) TILE-Gx8036 (b) TILE*Pro*64

Figure 1. Architecture diagrams for (a) TILE-Gx8036 [19] and (b) TILE*Pro*64 [20].

Table II. Architectural comparison for TILE-Gx8036 and TILE*Pro*64.

TILE-Gx8036	**TILEPro64**
36 tiles of 64-bit VLIW processors	64 tiles of 32-bit VLIW processors
32K L1i, 32K L1d, 256K L2 cache per tile	16K L1i, 8K L1d, 64K L2 cache per tile
Up to 750 billion operations per second	Up to 443 billion operations per second
60 Tbps of on-chip mesh interconnect	37 Tbps of on-chip mesh interconnect
500 Gbps (62.5 GB/s) of memory bandwidth	200 Gbps (25 GB/s) of memory bandwidth
1.0 to 1.5 GHz operating frequency	700 or 866 MHz operating frequency
10 to 55W (22W typical @ 1.0 GHz)	19 to 23W at 700 MHz
two DDR3 memory controllers	four DDR2 memory controllers
mPIPE for wire-speed packet processing	
MiCA for crypto and compression	

DOI: 10.1002/cpe

3. DEVICE PERFORMANCE STUDIES

Tilera provides the Tilera Multicore Components (TMC) library for general application development, suitable for a variety of task models and featuring components that developers can leverage for their routines. In addition, the gxio library provides programmability for features specific to TILE-Gx devices, such as mPIPE and MiCA. For ease of development on their many-core devices, Tilera provides a customized Eclipse IDE installation with numerous extensions, such as state trackers for individual tiles. These libraries and development tools are packaged from Tilera in a Multicore Development Environment (MDE) distributable with the necessary drivers and boot images for development on their platforms. Our work uses MDE version 4.2.2 on the TILE-Gx and MDE version 3.0.3 for the TILE*Pro*. The software versions packaged in our MDE releases are similar. The main difference between major MDE releases is the target architecture supported (version 3 corresponding to TILE*Pro* and version 4 for TILE-Gx).

Benchmarking these libraries is necessary to determine the upper bound on performance realizable for any library design (e.g., TSHMEM) or application. Routines relevant to the functionality required in TSHMEM are microbenchmarked to compare performance and overhead. Platforms targeted by our research are the TILEmpower-Gx server with a single TILE-Gx8036 operating at 1.0 GHz, and the TILEncore*Pro*-64 with a single TILE*Pro*64 operating at 700 MHz. A host machine is required for PCIe-card platforms such as the TILEncore*Pro*-64, while it is an option for standalone server platforms such as the TILEmpower-Gx.

In Sections 4 and 5, a performance analysis for the design of TSHMEM is conducted on the TILE-Gx. While TSHMEM supports both the TILE-Gx and TILE*Pro* architectures, TSHMEM performance numbers on the TILE*Pro* are not provided in later sections. Focus is emphasized on the newer, current-generation TILE-Gx architecture because of its higher relevance for this work and the decreased support for the older TILE*Pro*. Instead, microbenchmarking results in this section will provide general trends for expected performance with TSHMEM on the TILE*Pro*. Microbenchmark executions in this section are composed of an average of at least 1000 iterations.

3.1. Memory hierarchy

Before discussing the microbenchmarks, a brief synopsis of Tilera's memory hierarchy is necessary. Each physical tile on the TILE-Gx and TILE*Pro* consists of a processor with L1i, L1d, and L2 caches. Tilera employs several techniques to reduce latency for external memory operations, one of which is the Dynamic Distributed Cache (DDC). Tilera's DDC presents a large L3 unified cache that is the aggregation of L2 caches from all tiles. Each physical memory address is dynamically assigned to a home tile to manage, allowing memory requests to be potentially fulfilled from the caches of other tiles instead of memory, thereby maximizing on-chip performance.

The method by which memory addresses are assigned to home tiles is memory homing. Tilera's memory hierarchy provides for three classes of homing: local homing; remote homing; and hash-for-home. Local homing assigns a page of memory to the same tile accessing it. For memory regions exhibiting high locality, this approach provides for a potentially faster hit latency. Unfortunately, local homing loses the advantage of DDC as these pages cannot be distributed to other tiles' L2 caches. As a result, local homing is most suitable in cases such as small private data that can entirely reside in L2 cache, such as program stack data. Remote homing is the contra to local, whereby memory pages are homed on a tile other than the one currently accessing the data. This strategy is most useful in producer-consumer relationships when the producer can set a page for remote homing and write directly into the home tile's cache, avoiding unnecessary access to its own cache. The home tile as consumer can then directly consume the result from its own cache. Finally, hash-for-home is similar to remote homing; however, instead of homing a page to a single tile, the page is hashed and distributed across multiple tiles. This method allows for distributed memory accesses across the entire L3 DDC, reducing bottlenecks at any individual tile's cache. Hash-for-home is inappropriate for private single-reader data that are more suitable for local or remote homing, but excels for

DOI: 10.1002/cpe

memory shared between multiple threads or processes. By default, hash-for-home is used for a majority of data and instruction memory as it provides excellent performance for shared memory and good performance for private memory.

3.2. Tilera Multicore Components common memory

The TMC library provides routines for allocating shared memory between processes. Referred to as common memory, it differentiates itself from traditional cross-process shared-memory mappings in that all participating processes will map the shared-memory region at the same virtual address, enabling processes to share pointers into common memory. Additionally, any process can create new mappings which become visible to others, removing the restriction that all shared memory must be created from a parent process. TSHMEM leverages common memory to provide the PGAS model and shared-memory semantics of SHMEM. The bandwidth of memory-copy operations to and from this shared memory is decisively important in determining TSHMEM's overall performance because of its significant use in one-sided data transfers.

Figure 2 shows microbenchmark results for `memcpy()` operations between shared-memory segments via TMC common memory. Effective bandwidth on the TILE-Gx36 is much higher than on TILE*Pro*64 for all transfer sizes. This performance difference can be attributed to several reasons. The TILE*Pro*'s iMesh consists of four dynamic networks, one of which is dedicated to memory operations and another for cache coherency communication among tiles. The TILE-Gx's iMesh, however, has been redesigned to include five dynamic networks, two of which are now dedicated for memory request and response operations and one for cache coherency. As a result, TILE-Gx memory performance is substantially improved.

Effective bandwidth on TILE-Gx36 experiences three transitions in performance. The first two transitions are attributed to and occur at the L1d (32 KB) and L2 (256 KB) cache sizes, indicating representative performance for the cache system. The L1d cache performance tops out around 3.0 GB/s and the L2 cache performance reaches a peak around 2.87 GB/s. While the L1d and L2 cache performances are similar in this situation, this result can be influenced by the MDE version of the Tilera software installation and the performance optimizations introduced with newer MDE minor releases. Our previous work [1] used MDE 4.0.0, while this work uses MDE 4.2.2, providing higher realizable mesh bandwidth for cache and memory transfers.

The third performance transition on TILE-Gx36 is attributed to Tilera's L3 DDC. Effective bandwidth decreases steadily between the L2 cache size of 256 KB to the L3 DDC limit of 9 MB (for

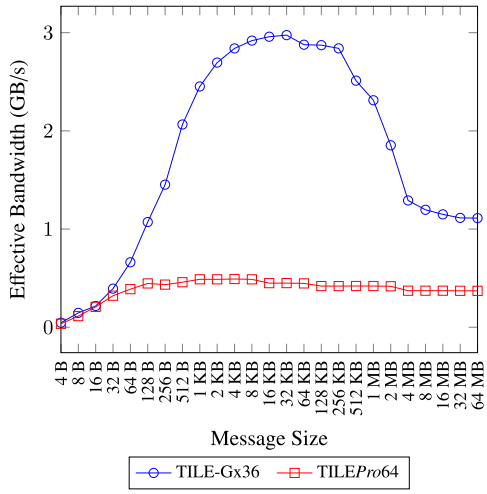


Figure 2. Effective transfer bandwidth (cache and memory) for shared-memory copy operations on one core of TILE-Gx36 and TILE*Pro*64. Transfers are from Tilera's common memory to another common memory region. From Table II, the TILE-Gx8036 provides up to 62.5 GB/s of aggregate, theoretical memory bandwidth.

DOI: 10.1002/cpe

TILE-Gx8036) as the L2 caches on the device are exhausted. The performance of memory-to-memory transfers is approximately 1.1 GB/s for transfer sizes beyond 9 MB (projected aggregate bandwidth of 40 GB/s) and remains constant as transfer size increases. The TILE*Pro*64 follows the same trends experienced with TILE-Gx36, but at a less-pronounced performance benefit. Performance is stable at or near 0.50 GB/s through the L1d and L2 cache sizes and decreases into memory-to-memory transfers (0.37 GB/s, projected aggregate of 23.7 GB/s). These results represent our practical experience in determining the realistically achievable memory bandwidths for these architectures. As such, we make no claims to verify the theoretical aggregate bandwidths provided by Tilera in Table II because of non-trivial variations with methods for empirically measuring aggregate bandwidth as well as the result's limited applicability in our experiments. These memory-bandwidth results are revisited in Section 4 when TSHMEM one-sided put/get performance is analyzed.

3.3. Tilera Multicore Components User Dynamic Network helper functions

Tilera provides access to the UDN (User Dynamic Network), a low-latency direction-order-routed dynamic network on their iMesh. Developers attach a one-word header to each payload with information about the destination tile and transfer the data packet via the UDN—at a rate of one word per hop, per clock cycle—into one of four demultiplexing queues at the destination. Each receiving queue on the UDN can accommodate up to a payload size of 127 words (8-byte word on the TILE-Gx, 4-byte word on the TILE*Pro*), making the UDN suitable for small-sized explicit communication.

The TMC library provides UDN helper routines that facilitate these transfers via two-sided send-and-receive calls. We microbenchmark the UDN's latency performance of minimum-sized payloads on the TILE-Gx36 and TILE*Pro*64 between pairs of tiles with varying distances: *neighbors* for transfers between adjacent tiles; *sides* for transfers horizontally or vertically across the test area; and *corners* for diagonal transfers over the entire test area. The effective test area on both devices is 6×6 tiles, providing full coverage of the TILE-Gx36. Timing is performed on the sender tile as a halved average between a one-word send and a one-word acknowledgment from the receiver.

Average one-way latencies are depicted in Figure 3. For each case, average latencies were consistent with low variance of up to 1 ns, regardless of the message direction. Each case can be broken down into two components: setup-and-teardown time and network-traversal time. The clock frequency and packet-switching rate are known, allowing us to roughly determine the setup-and-teardown time. Our TILE-Gx36 operates at 1 GHz, requiring 1 ns to route one word/hop. In comparison, the TILE*Pro*64 at 700 MHz requires 1.43 ns. The number of hops in a 6×6 mesh network is 1, 5, and 10 for neighbor-to-neighbor, side-to-side, and corner-to-corner, respectively;

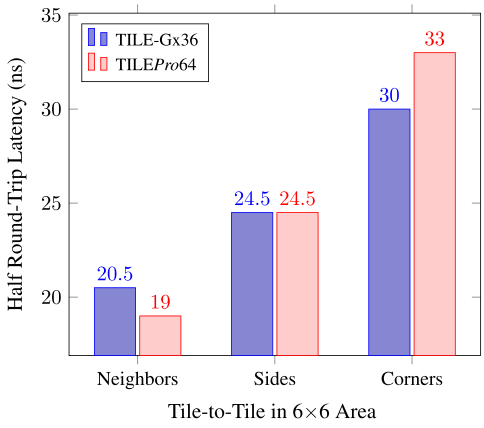


Figure 3. Average half round-trip latencies (100 million iterations) on User Dynamic Network between adjacent tiles (neighbors), tiles across the area (sides), and tiles on opposite corners of the effective area (corners). TILE-Gx36 has higher latency because of setup-and-teardown on a 64-bit switching fabric versus TILE*Pro*64's 32-bit fabric.

DOI: 10.1002/cpe

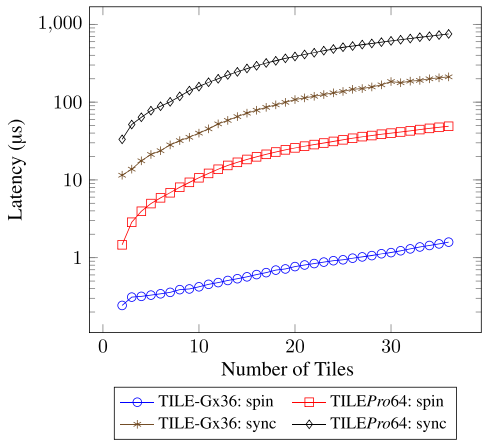


Figure 4. Latencies of Tilera Multicore Components spin and sync barriers. Spin barriers leverage spin polling to outperform the sync barriers' use of process preemption.

therefore, the estimated setup-and-teardown time is roughly 19.5 ns for the TILE-Gx and 18 ns for the TILE*Pro*. Because of the longer setup-and-teardown time, the TILE-Gx has a higher average latency for the neighbor-to-neighbor case, but exhibits equal or lower average latency for side-to-side and corner-to-corner as the number of hops increases. These latency tests have focused on minimum-sized payloads, but actual data transferred are doubled on TILE-Gx because of a 64-bit switching fabric compared with 32-bit on TILE*Pro*.

3.4. Tilera Multicore Components spin and sync barriers

The TMC library provides two types of barriers for synchronization: spin and sync. True to its name, the spin barrier will block processing and poll continuously until the correct number of tasks has reached the barrier. This polling results in lower overhead but incurs significant performance degradation if the currently blocking task is context-switched out for a new task. As such, spin barriers should only be used when there is only one task per tile. In contrast, the sync barrier interacts with the Linux scheduler and notifies it when the barrier begins to block. The scheduler can swap out the task while it waits and replace it for another task to continue processing. The sync barrier incurs a larger performance penalty than spin, but allows for additional use cases when the restrictions of a spin barrier are inappropriate. The semantics for these two barrier types require a state variable backed by shared memory, and therefore rely on the memory technology.

Latency results for spin and sync barriers are shown in Figure 4. As expected, spin barriers vastly outperform sync barriers because of their polling nature, with latencies of 1.6 µs and 49.0 µs at 36 tiles for the TILE-Gx36 and TILE*Pro*64, respectively, compared to 211 µs and 754 µs. Furthermore, the barriers for the TILE-Gx significantly outperform the TILE*Pro*'s because of different memory technologies (DDR3 versus DDR2). Because SHMEM focuses on low-overhead, low-latency performance, the TMC spin barrier for TILE-Gx is an appealing candidate for use in TSHMEM, but its performance difference with the spin barrier on TILE*Pro* poses a challenge in realizing the same low-latency performance for the TILE*Pro*.

4. DESIGN OVERVIEW OF TSHMEM

The software architecture of TSHMEM leverages the Tilera TMC libraries to provide an OpenSHMEM-compliant high-performance library for Tilera many-core processors. TSHMEM targets the OpenSHMEM v1.0 specification and implements all functionality required by SHMEM applications, with exception of support for static symmetric-variable transfers using SHMEM atomic operations. All other SHMEM functionality, including collectives and atomic operations with dynamic variables, is supported.

DOI: 10.1002/cpe

The following subsections are ordered categorically according to Table I, each including design description and performance results for the TILE-Gx8036. The performance of TSHMEM is compared with the microbenchmark results for the TILE-Gx in Section 3 and with other OpenSHMEM implementations: the OpenSHMEM reference implementation version 1.0f (referred to afterward as simply OpenSHMEM or OSH), and OSHMPI (git commit 1f33a2735b on 20140819) atop MPICH [21] version 3.1.3. The underlying functionality in the OpenSHMEM reference implementation is provided by GASNet version 1.22.0, cross-compiled for the TILE-Gx architecture with GASNet's SMP conduit. In contrast to the GASNet middleware in the OpenSHMEM reference implementation, TSHMEM does not leverage any middleware, instead opting to design its functionality with device primitives and algorithm exploration for higher device utilization and bare-metal performance. Execution runs with MPICH use `mpiexec -bind-to core:1` to set CPU affinity. All compilations were carried out with the TILE-Gx compiler based on GCC version 4.4.7. Latency benchmarks for put/get and collectives are provided by the OSU micro-benchmarks suite [22].

4.1. Environment setup and initialization

The SHMEM implementations typically consist of the library to which applications are linked and an executable launcher that sets up the initial environment, forks the requested number of processes, and executes the desired application. TSHMEM's executable launcher initializes the environment by setting up Tilera's TMC common memory in order to create a globally shared space visible to all processes, and setting up the UDN for explicit communication between the tiles participating in SHMEM. After forking, each process uniquely binds to a tile, creating a one-to-one mapping. After `exec()`, the application calls `start_pes()` to finish initialization. At this time, the globally shared memory is partitioned symmetrically among participating tiles (providing the PGAS memory model), and each tile reports its partition's starting address to every other tile via the UDN.

Dynamic symmetric memory is managed via `shmalloc()` and `shfree()`. TSHMEM's design of `shmalloc()` consists of a doubly linked list tracking the memory segments being used in the current tile's symmetric partition. Memory is kept implicitly symmetric by the constraints imposed when using `shmalloc()`, requiring applications to call the routine on all PEs with the same size argument at the same location in the program execution path.

4.2. Point-to-point data transfers

OpenSHMEM specifies several categories of point-to-point, one-sided data transfers consisting of elemental, bulk, and strided put/get operations. Elemental put/get functions operate on single-element symmetric objects (e.g., short, int, float), whereas bulk functions operate on contiguous data. Strided operations allow the transfer of data with strides between consecutive elements in the source and/or target arrays. In the v1.0 specification, put operations will return from the function once the data transfer is in flight and the local buffer is available for reuse by the calling PE. Get operations, in contrast, will block and not return until the requested memory is visible to the local PE.

4.2.1. Dynamically allocated symmetric objects. At the startup of a SHMEM program, shared-memory partitions are given to each tile. Because of the symmetry of each partition, a tile in TSHMEM can determine the virtual address of any other tile's dynamic symmetric object by calculating the offset of its own object from its partition's start address and then adding the offset to the target tile's partition start address. The data transfer is then facilitated with a `memcpy()` operation using the calculated virtual address into TMC common memory.

4.2.2. Statically allocated symmetric objects. Static symmetric objects are treated very differently from their dynamic counterpart. These objects are allocated statically into the program's heap space at link time and are symmetric because the virtual addresses of the program heap are identical when parallel processes are instantiated from the same executable. Unfortunately, the heap space resides in private memory of a process and is not directly accessible to other processes.

DOI: 10.1002/cpe

TSHMEM facilitates data transfer for static symmetric objects via UDN interrupts. The put/get functions check the data target and source addresses to see if either address does not reside in the globally partitioned shared space. If an address does not reside in the shared space, it is assumed to be a static symmetric variable. The local tile will notify the remote tile over UDN, causing an interrupt and forcing the remote tile to service the operation only when the local tile cannot. If one of the addresses is dynamic, either the local or the remote tile will be able to directly access that dynamic memory to service the request. For example, if the local tile cannot *get* from a remote tile's static symmetric variable, the remote tile can instead *put* into a dynamic symmetric variable on the local tile. This scenario represents a static-to-dynamic or dynamic-to-static transfer and incurs a minimal performance impact compared to dynamic-to-dynamic transfers. In the case when both target and source addresses point to static symmetric variables, neither the local nor remote tile will be able to completely service the operation. For these static-to-static transfers, a temporary shared-memory buffer is created to assist in the transfer, but incurs an additional memory copy operation as overhead. Unfortunately, static symmetric-variable transfers in TSHMEM are not currently supported on the TILE*Pro* architecture because of lack of support for UDN interrupts.

4.2.3. Performance of SHMEM put/get. Figure 5 shows the effective bandwidth for dynamic-to-dynamic `shmem_putmem()` and `shmem_getmem()` transfers in TSHMEM, OpenSHMEM, and OSHMPI. For TSHMEM, put performance closely aligns with get performance. The dynamic put/get design in TSHMEM demonstrates low overhead as the realizable performance closely matches the TILE-Gx performance from the common memory microbenchmark in Figure 2 using the hash-for-home strategy described in Section 3.1.

Both put and get performances are higher in TSHMEM when comparing these results with OpenSHMEM and OSHMPI. The results are illustrated in Figures 6a–6d with latency performance on a logarithmic scale for dynamic and static transfers. Small-message dynamic put latencies (less than 1 KB) with TSHMEM are three to four times faster than those in OpenSHMEM because of TSHMEM's bare-metal implementation with minimized overhead and an explicit `memcpy()` operation. In contrast, OpenSHMEM incurs larger overhead when passing these put operations to GASNet's active-message interface and allowing its generalized SMP conduit to handle the message transfers and acknowledgments. Likewise, TSHMEM small-message put operations are two to three times faster than those from OSHMPI. TSHMEM also exhibits a slight performance benefit of 0.1 μs for small-message get operations over OpenSHMEM and OSHMPI.

With the case of static transfers, latency performances incur a penalty compared with dynamic transfers. In TSHMEM, this behavior is expected because of the use of a temporary shared-space buffer to aid in transfers between static symmetric variables. The performance of these static transfers in TSHMEM, however, is consistently higher than that of OpenSHMEM and OSHMPI for small to medium transfers, supporting the approach we have taken with the design of TSHMEM by leveraging the UDN when appropriate.

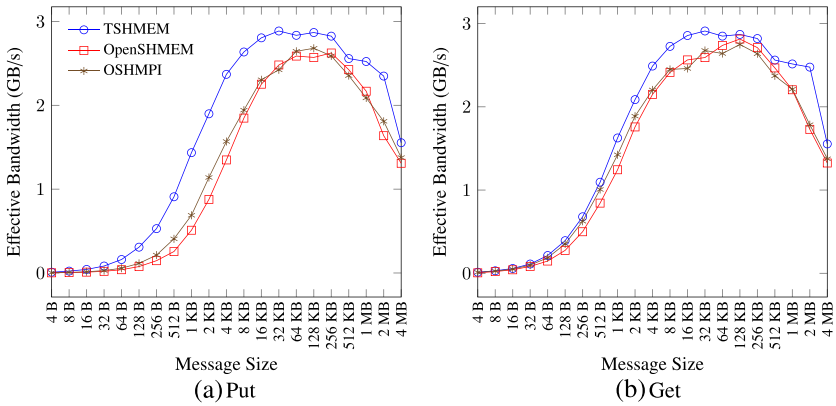


Figure 5. Effective bandwidth of SHMEM put/get transfers on TILE-Gx36. (a) put and (b) get.

DOI: 10.1002/cpe

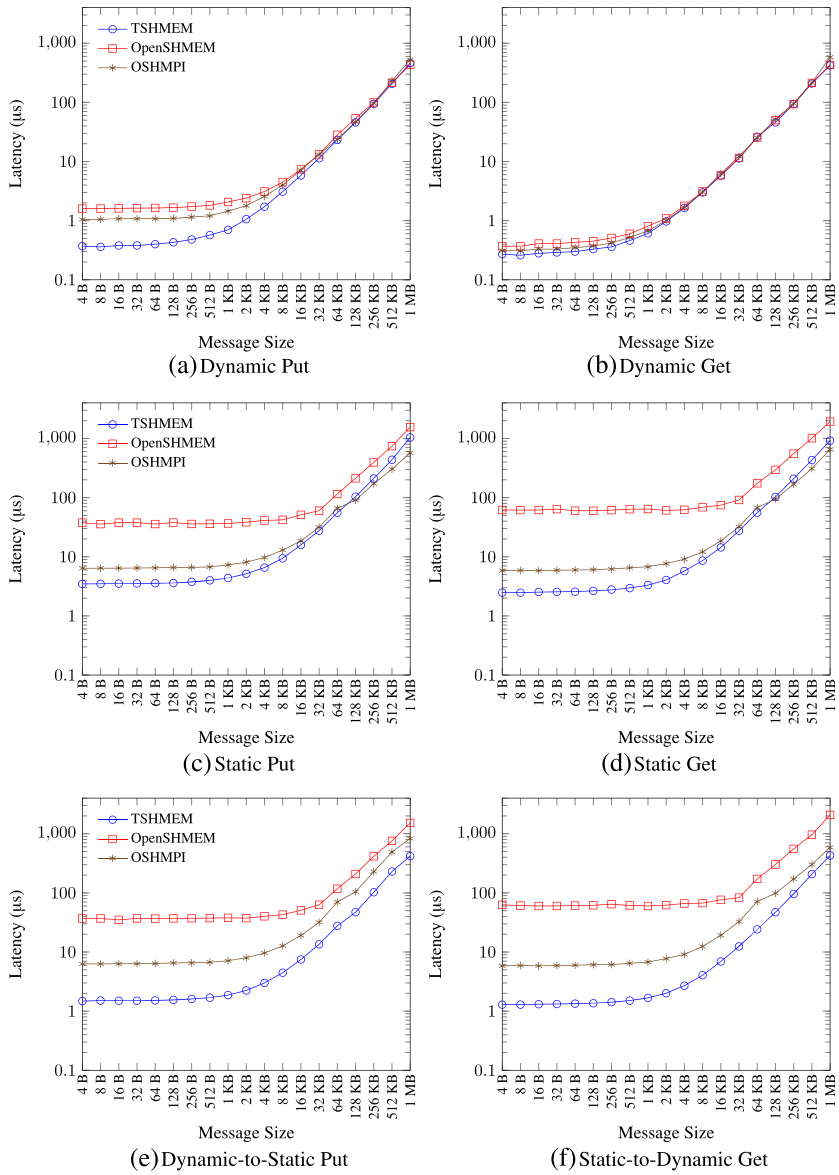


Figure 6. Latencies of SHMEM put/get transfers on TILE-Gx36. (a) dynamic put; (b) dynamic get; (c) static put; (d) static get; (e) dynamic-to-static put; (f) static-to-dynamic get.

Furthermore, TSHMEM includes optimizations for improved performance with dynamic-to-static put operations and static-to-dynamic get operations as seen in Figures 6e and 6f. When the local side of the transfer is represented by a dynamic symmetric variable, the remote tile is able to service the operation with little overhead, instead of the alternative static-to-static case where intermediate buffers are required. With these optimizations, TSHMEM latencies for these two cases are reduced by more than half of the static-to-static latencies. In contrast, both OpenSHMEM and OSHMPI over MPICH relegate these two cases to the static-to-static code path, with comparable performance with the static transfers seen in Figures 6c and 6d. Interestingly, OSHMPI performs slightly worse for transfers at and above 256 KB with these static-to-dynamic put operations compared with the static-to-static case. Intuitively, OSHMPI should perform similar to the statics case, warranting further investigation at possible non-optimal behavior. Note that, by definition, functional semantics for the remaining two cases of static-to-dynamic put operations and dynamic-to-static get operations are equivalent to dynamic-to-dynamic transfers because the remote PE's symmetric variable is dynamically allocated in shared memory and can be directly accessed by the local tile.

DOI: 10.1002/cpe

4.3. Synchronization

The OpenSHMEM specification provides several categories of synchronization: barrier sync; communication sync with fence/quiet; and point-to-point sync (waiting until a variable's value has changed). TSHMEM includes these functions to provide computation and communication synchronization for SHMEM processes.

4.3.1. Barrier synchronization. Barrier synchronization in SHMEM is provided by two routines: `shmem_barrier_all()`, which blocks forward processing until all tiles reach the barrier, and `shmem_barrier()`, which invokes a barrier on a subset of the tiles defined by an active-set triplet of which tile to start at, the stride between consecutive tiles, and the number of tiles participating in the barrier. The microbenchmark results for TMC spin and sync barriers in Figure 4 illustrate that using sync barriers is not feasible because of their high latency, and the spin barrier on TILE*Pro* is significantly slower than the one on TILE-Gx.

Consequently, TSHMEM's barrier design uses the UDN to synchronize between tiles. The start tile in the active set generates an active-set identification for the barrier in order to prevent overlapping barrier calls from returning out-of-order or stalling. The active-set identification is encoded with a *wait* signal and is sent to the next tile and resent linearly until the last tile sends it back to the start, acknowledging that all participating tiles have reached the same execution point in the program. The process is repeated with a *release* signal, allowing the blocking processes to linearly forward the signal before resuming program execution. The number of messages transferred for this operation is $2n$, where n is the number of PEs in the barrier. Interestingly, another design was evaluated whereby the start tile broadcasts the *release* signal instead of having each tile forward it linearly in a chain. Barrier latencies, however, were two times slower for this method.

The performance of `shmem_barrier_all()` is shown in Figure 7 for TSHMEM, OpenSHMEM, and OSHMPI. For comparison and convenience, the microbenchmark results for the TMC spin barrier on TILE-Gx36 in Figure 4 are also illustrated. While not depicted in Figure 7, TSHMEM barriers on TILE*Pro*64 perform with a 36-tile latency of 3 µs, on the same magnitude of performance as TSHMEM barriers on TILE-Gx and vastly outperforming the TMC spin barrier on TILE*Pro*64 (50 µs). The TMC spin barrier on TILE-Gx36, however, outperforms the TSHMEM barrier, opening the possibility of adopting its use for the TILE-Gx version of TSHMEM. Unfortunately, the use semantics for TMC barriers require memory allocation of a state variable to track the number of tasks in the barrier. This allocation would have to occur for each instance of an SHMEM barrier call in order to ensure that PEs that are engaged in multiple barriers do not return from the wrong barrier. One design option is to leverage memoization techniques to alleviate some of the allocation penalty of state variables; however, the added complexity from both memoization management and state-variable management may result in a performance penalty greater than the current performance of TSHMEM barriers over UDN, especially because the current TSHMEM

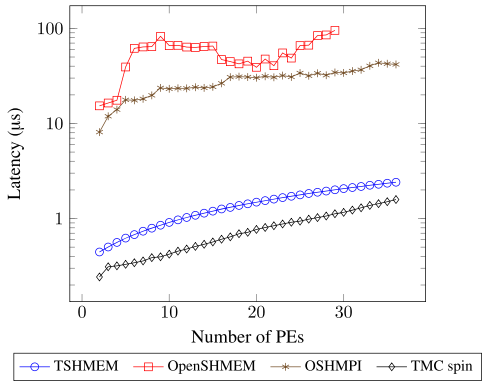


Figure 7. Latencies of SHMEM barrier on TILE-Gx36.

DOI: 10.1002/cpe

barrier design does not depend on state variables nor require memory allocation. We plan to explore memoization and its performance implications in future TSHMEM barrier designs.

Alongside the TSHMEM barrier results, performance for barriers in OpenSHMEM and OSHMPI is also shown. OpenSHMEM barriers demonstrate significant variance and unreliable behavior when scaling up. OSHMPI barriers have minimal variance and scale in performance from 8 to 44 μs at 36 tiles. In contrast, TSHMEM barriers reach 2.4 μs at 36 tiles, over 18 times lower latency than OSHMPI barriers.

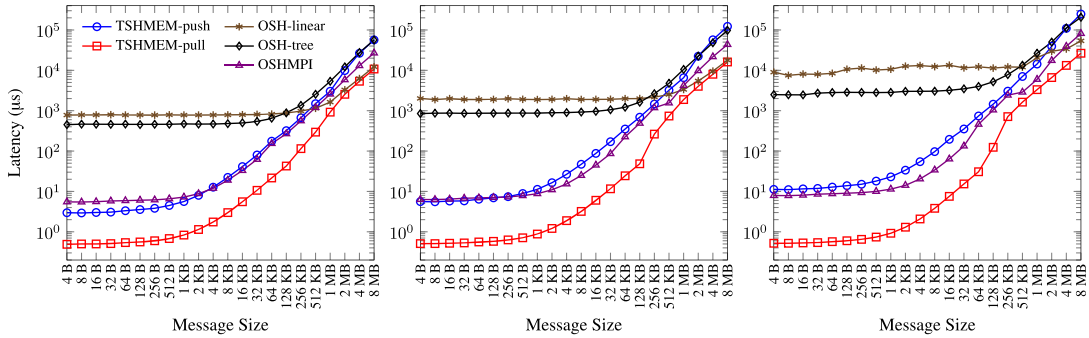
4.3.2. Fence/Quiet. Because put operations do not wait for completion before returning to the calling PE, the communication synchronization routines `shmem_fence()` and `shmem_quiet()` ensure outstanding puts are ordered properly or completed before returning. The `shmem_fence()` routine guarantees put ordering to individual PEs before and after the function call, but does not guarantee completion. In contrast, `shmem_quiet()` is semantically stronger and will block execution until all outstanding puts to all PEs are completed. TSHMEM implements `shmem_quiet()` using `tmc_mem_fence()`, a memory fence operation that blocks until all memory stores are visible. Currently, `shmem_fence()` is set as an alias of `shmem_quiet()`, providing it the stronger semantics until `shmem_fence()` is implemented with its weaker semantics.

4.4. Collective communication

The SHMEM collective routines provide group-based communication for a subset of tiles. Collective designs and performance results for TSHMEM are discussed in the following. While collective algorithms have been explored with greater depth in other parallel environments such as MPI [23–27], the collective algorithms in TSHMEM presented here are intended to explore performance behaviors on the TILE-Gx and its 2D mesh. Results for OpenSHMEM and OSHMPI are also provided as a basis for comparison of both algorithmic performance and runtime/conduit behavior.

4.4.1. Broadcast. Broadcast is a one-to-all operation where the active set of PEs obtains data from a root PE. TSHMEM currently has support for push-based and pull-based implementations of broadcast.

The push-based broadcast is performed by having the root PE perform a put operation sequentially to all other PEs. This algorithm does not fully utilize the mesh fabric or the memory bandwidth of the Tilera processors, and is therefore only used for testing purposes in TSHMEM. In contrast, the pull-based broadcast is performed by having all other PEs in the active set perform a get operation on the data from the root PE. This approach distributes work to all other PEs on the device, instead of the root PE performing all of the work as is the situation with push-based. All other PEs will be issuing concurrent requests to a single memory location. The effective bandwidth is maximized on the TILE-Gx by leveraging its L3 distributed cache (Section 3.1) and storing this repeatedly accessed data within the L2 caches of the tiles, bypassing the need to go to memory. Local tiles on the device observe maximum performance by accessing data directly from cache, but alternative algorithms are



(a) 8 PEs (b) 16 PEs (c) 32 PEs

Figure 8. SHMEM broadcast latencies on TILE-Gx36. (a) 8 PEs; (b) 16 PEs; (c) 32 PEs.

DOI: 10.1002/cpe

preferred for PEs on multi-socket systems that have to access the data through memory and cannot benefit from this optimization.

Figure 8 shows results for push-based and pull-based TSHMEM algorithms, two algorithms within OpenSHMEM, and the performance of the underlying MPI broadcast within OSHMPI. The first OpenSHMEM algorithm is a linear broadcast that is functionally equivalent to the TSHMEM pull-based approach: all PEs other than the root PE issue a get operation on the data. The second algorithm is a binary-tree broadcast, whereby a binary-tree graph is generated to determine which parent PEs transfer data to which children PEs. With the root PE at the tree's root, parent nodes transfer data via put operations until all children receive the broadcasted data. For message sizes less than 128 KB, the tree-based OpenSHMEM algorithm is faster than the linear algorithm, but demonstrates unfavorable performance at large message sizes. Furthermore, the performance difference is significant between the linear algorithm and TSHMEM's pull-based approach despite the functional similarity. TSHMEM-pull outperforms OpenSHMEM-linear for all message sizes, and the linear algorithm is only able to approach the performance of TSHMEM-pull at large message sizes because of amortization of runtime overhead with the higher latency of large data transfers. OpenSHMEM-linear's performance difference can be attributed to the GASNet communication runtime that it uses. In addition to the large overhead from GASNet, we observe system instability and runtime variance using GASNet with large PE counts. As the number of PEs increase, OpenSHMEM experiences higher latency variance with increasing message size, as indicated in Figure 8c. In addition, low performance with OpenSHMEM collectives is not isolated to TILE-Gx; it has also been observed with distributed systems supporting InfiniBand [28] and is an area of improvement for the reference implementation.

The MPI broadcast used by OSHMPI performs similarly to the straightforward push-based approach in TSHMEM. In comparison, the pull-based TSHMEM algorithm is an order of magnitude faster than MPI broadcast, achieving between 0.5 and 0.8 µs of latency for small-message broadcasts. For large message sizes from 4 to 32 MB, OSHMPI stabilizes with approximately three times higher latency than TSHMEM-pull. Finally, TSHMEM-pull exhibits high parallel scalability as the number of PEs increases. For 64-byte transfers, TSHMEM broadcast latencies are from 0.54 (8 PEs) to 0.57 µs (32 PEs), benefiting from the TILE-Gx distributed cache.

4.4.2. Fast collection. Collection is an all-to-all operation that concatenates an array from each PE and distributes the resultant array to all PEs. The OpenSHMEM specification defines two types of collection routines: collect and fast collect (fcollect). General collect allows each PE to supply a different-sized array for concatenation. PEs need to communicate with each other to know how far along the concatenation has progressed as well as where to append their array to the result. In contrast, fast collect has the restriction that each PE must supply the same-sized array, allowing PEs to implicitly know where to append their portion to the resultant array.

Figure 9 shows TSHMEM results for two algorithms: a naive design leveraging pull-based broadcast and a linear design. For the naive fcollect algorithm, all PEs perform a put operation and send

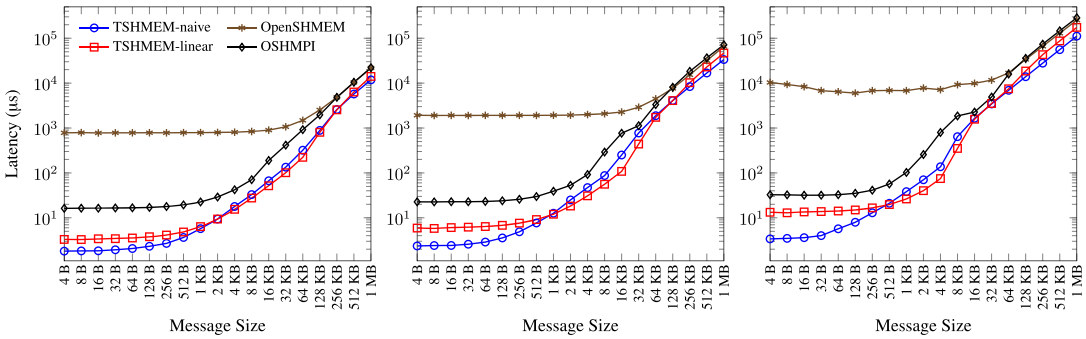


Figure 9. SHMEM fast-collect latencies on TILE-Gx36. (a) 8 PEs; (b) 16 PEs; (c) 32 PEs.

DOI: 10.1002/cpe

their data to a root PE. Once the root PE receives everyone's array, a pull-based broadcast is executed where all other PEs get the newly concatenated result. This fast collect design can be broken into two stages: (1) n PEs (including the root PE) transfer M bytes to the root PE's destination array, and (2) root PE broadcasts $(n \times M)$ bytes to destination arrays on $(n - 1)$ PEs. Treating M as constant, stage 1's total data transferred scales linearly with the number of participating tiles, similar to a broadcast operation. Stage 2, however, scales *quadratically* in total data as the number of tiles increases because each PE receives a copy of the entire concatenated result containing arrays from all other PEs. Summarizing this algorithm, all PEs execute a put operation to the root PE, then all PEs will execute a get operation from the root PE for the result. In contrast, the linear fcollect algorithm has all PEs execute a put operation to each other PE, sending it the portion of its data. This algorithm allows the result to be iteratively built on all PEs as the data arrive. Both TSHMEM and OpenSHMEM implement this linear algorithm, with results illustrated in Figure 9. Within OSHMPI, fcollect is implemented using `MPI_Allgather()`, which performs the same functionality from the MPICH library. On the TILE-Gx, MPICH performance is more favorable than that of GASNet.

As the number of PEs increases, TSHMEM's fcollect performance surprisingly widens in favor of the naive approach for small message sizes. The main performance advantage of the naive approach with a large number of PEs is cache locality. The concatenated array is built on the root PE and then repeated cache reads can distribute the result to the other PEs efficiently via the L3 distributed cache. The linear algorithm only outperforms this naive approach for medium-sized messages. Because small-message or large-message transfers are emphasized in most applications, the default fcollect algorithm in TSHMEM is this naive approach with pull-based broadcast. In comparing the algorithms for large messages, TSHMEM-naive is 1.6 times faster than TSHMEM-linear, 2.3 times faster than OpenSHMEM, and 2.6 times faster than OSHMPI.

4.4.3. Reduction. Reduction is an all-to-all operation that performs an associative binary operation on the array elements from each active-set PE. OpenSHMEM reduction routines are defined by the element type (e.g., short, int, float) and the reduction operation (e.g., xor, sum, min, max).

TSHMEM currently includes three designs for reduction operations: naive, linear, and tree. The design for naive reduction has a root PE continuously get data from each other PE, iteratively performing the reduction operation on the values with the result array. Once all active-set PEs have participated and the final reduction is available, a pull-based broadcast is issued to distribute the results to all other participating PEs. Unlike the naive fcollect whereby each PE was able to put its data onto a root PE concurrently, the naive reduction is bottlenecked with the root PE performing get operations and reducing the results as they sequentially arrive. Therefore, TSHMEM also provides a tree reduction algorithm that sets up a binary-tree communication pattern and reduces the results from the child nodes to each parent node until it reaches the root node (the root PE). Similar to naive, a pull-based broadcast is then executed for each PE to obtain the reduction result.

Finally, TSHMEM and OpenSHMEM both provide a linear reduction algorithm. For linear reduction, each PE iteratively gets the source data from all other PEs and reduces them to a result for the

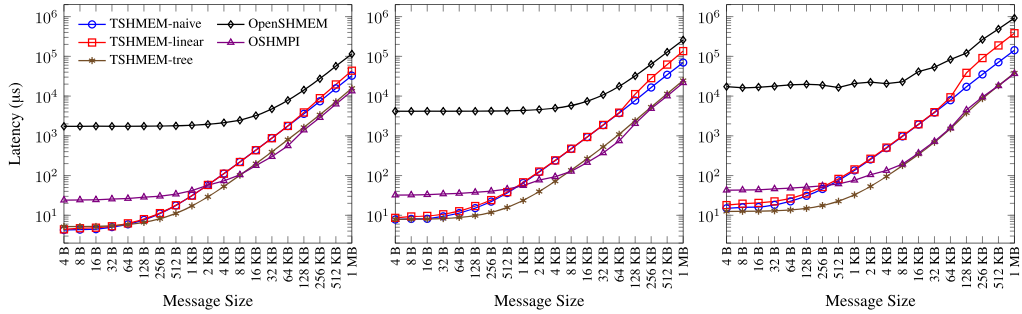


Figure 10. SHMEM float-summation reduction latencies on TILE-Gx36. (a) 8 PEs; (b) 16 PEs; (c) 32 PEs.

DOI: 10.1002/cpe

local PE. Broadcasting the final result is not needed as the result is computed locally on each PE. For OSHMPI, reduction operations are translated to `MPI_Allreduce()` function calls.

Results for summation reduction with single-precision floating-point arrays are shown in Figure 10. We observed similar performance trends for other reduction operations such as integer summation and integer XOR. TSHMEM-linear results surprisingly show similar performance as the TSHMEM-naive method; however, TSHMEM-naive outperforms TSHMEM-linear for larger message sizes. At 32 PEs and for message sizes beyond the L1 cache size (32 KB), TSHMEM-naive is approximately two to three times faster than TSHMEM-linear. For TSHMEM-linear, cache locality significantly affects performance at these larger message sizes since all participating PEs are attempting to get data from all other PEs while accessing their own cache and memory to compute results. This behavior causes numerous concurrent and random memory accesses, whereas TSHMEM-naive experiences better cache locality because of the root PE performing all of the reduction calculations. For 1-MB transfers with 32 PEs, the root PE in TSHMEM-naive experiences half as many local-tile L3 cache reads compared with TSHMEM-linear, and the remaining PEs require only 0.6% as many local-and-remote L3 cache reads to retrieve the reduced dataset from the root PE compared with locally computing it.

At 8 and 16 PEs, TSHMEM-tree is equal to or faster than TSHMEM-naive. For 32 PEs, TSHMEM-tree is faster than TSHMEM-naive for all message sizes. The default reduction algorithm in TSHMEM is the tree approach due to more efficient memory utilization with increasing PE counts. OpenSHMEM performance exhibits similar trends as with broadcast and fcollect. OSHMPI reduction performance is approximately 2.8 times slower than TSHMEM-tree for small messages, and similar in performance for larger messages.

The collective results in this subsection are intended as a case study for the TILE-Gx. A common theme is that distributed collective algorithms can display insufficient performance on shared-memory, many-core devices. Others have reached similar conclusions when experimenting with multi-core systems [29]. In leveraging the device-level microbenchmarking results, we demonstrate that the design of collective communications in TSHMEM offers high performance on the TILE-Gx many-core architecture, while enabling further library exploration toward systems consisting of multiple many-core processors.

5. APPLICATION CASE STUDIES

The SHMEM and OpenMP are highly amenable programming environments for SMP architectures because of their shared-memory semantics. With many-core processors emerging onto the HPC scene, developers are interested in the performance and scalability of their applications for these devices. This section analyzes several applications, written in both SHMEM and OpenMP, on the TILE-Gx8036 [2]. We focus our analysis on showcasing performance differences between OpenMP (provided by the TILE-Gx GCC 4.4.7 compiler) and the three SHMEM implementations: TSHMEM, the OpenSHMEM reference implementation, and OSHMPI atop MPICH. OpenMP serves as a baseline for our performance comparison because of its ubiquity for parallel programming on SMP devices. In comparing TSHMEM with OpenMP, we aim to show that libraries like TSHMEM can offer competitive or higher performance than established language-based solutions.

The applications in this section consist of both custom-developed kernels and example programs from the OpenSHMEM test suite version 1.0d [10]. These applications are presented as follows: exponential curve fitting; OSH 2D heat equation; matrix multiply; OSH matrix multiply; OSH heat image; and a case study in parallelizing the FFTW library. SHMEM applications were ported to OpenMP when it was easily achieved. Specific optimizations were made only when the computational algorithm remained unchanged for both versions of the application. Scalability results are presented with increasing number of PEs, where PEs are either processes in SHMEM or threads in OpenMP and are reported in execution times up to the realistic maximum number of PEs for the TILE-Gx8036 (36 PEs). For OSH heat image, we also present results with increasing problem sizes to illustrate TSHMEM's performance improvement over OpenMP and OSHMPI at full-device scale.

DOI: 10.1002/cpe

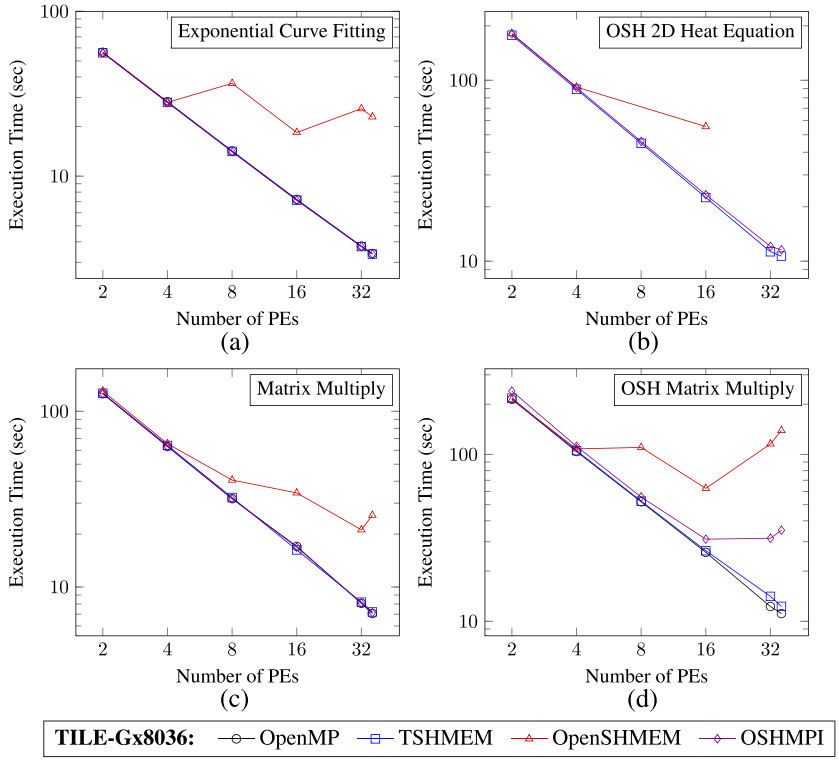


Figure 11. Execution times for (a) exponential curve fitting (100M points, double); (b) OSH 2D heat equation (Jacobi method on a 288 × 288 matrix); (c) matrix multiply (2048 × 2048, double); and (d) OSH matrix multiply (2048 × 2048, double).

5.1. *Exponential curve fitting*

An exponential equation of the form $y = ae^{bx}$ can be represented in linear form with logarithms: $ln(y) = ln(a) + bx$. This form allows us to leverage linear curve-fitting via least-mean-squares approximation and transform the final result back to exponential form with inverse logarithms. The implementation for curve fitting is embarrassingly parallel and consists of a constant number of barriers and reductions. This application serves as a metric for parallel-performance overhead for the runtime environments that we are testing.

The execution times are presented in Figure 11a. OpenSHMEM is the only runtime environment that does not exhibit the expected linear scalability. Scalability of OpenSHMEM is significantly impacted for executions with more than 4 PEs on the TILE-Gx. Further investigation shows that this behavior is a result of generic instrumentation of the GASNet conduits on the TILE-Gx and interoperability issues with GASNet and the TILE-Gx's process scheduler. As a result, GASNet is unable to leverage the TILE-Gx's NUMA (non-uniform memory access) hierarchy in an efficient manner. Attempting to manually set the processor affinity via the Linux scheduler and via `numactl` fails to improve its high-variance behavior. As a result, the performance comparisons between TSHMEM, OpenMP, and OSHMPI in this section are more relevant in demonstrating application behavior on TILE-Gx.

5.2. *OSH 2D heat equation*

The OpenSHMEM (OSH) website has a test suite consisting of benchmarks and applications. One such application is an iterative heat-equation solver for heat distribution in a rectangular (2D) domain via conduction. The provided application supports three iteration methods: Jacobi, Gauss–Seidel, and successive over-relaxation. We benchmark our runtime environments with the Jacobi method on a 288 × 288 rectangular domain, with 288 chosen as the least-common multiple

DOI: 10.1002/cpe

of 32 and 36, such that the domain space is evenly divisible amongst the PEs. SHMEM communication consists of a linear number of put operations, broadcasts, reductions, and barriers. An OpenMP implementation was not tested for this application.

Execution times in Figure 11b show that TSHMEM and OSHMPI performance are similar, with a performance edge to TSHMEM at full-device utilization because of barrier performance. In contrast, OpenSHMEM executions behaved erratically, preventing several PE counts from executing to completion.

5.3. Matrix multiply

The matrix-multiplication algorithm chosen for instrumentation was a partial-row dissemination with loop-interchange optimization for three matrices: $C = A \times B$. Each PE is partitioned a block of sequential rows to compute the partial result. In the case of OpenMP, the A, B, and C matrices are shared among the threads via compiler directives. Because of SHMEM's symmetric heap, the A and C matrices can be easily partitioned among the PEs, but each PE receives a private copy of the B matrix because of the pattern of computation. Consequently, the memory requirements are forced to scale with the number of PEs and the size of the matrix because of the private copies that reside on each PE. There are other parallelization strategies that do not require private matrix copies, but the pattern of computation and communication would have differentiated from the OpenMP version. In addition to row dissemination, loop interchange can easily occur because each matrix element in C has no data dependency with its other elements. By interchanging the inner-most loop with one of its outer loops, locality of reference and cache-hit rates drastically increase.

Execution times for SHMEM and OpenMP matrix multiplication are presented in Figure 11c. For the SHMEM version, communication consists of broadcasting the B matrix to all PEs, unless the data can be accessed directly from the remote partition via `shmem_ptr()`. The OpenMP version has only implicit barriers, as all three matrices are shared via compiler directives and are directly accessible. The execution times for OpenMP, TSHMEM, and OSHMPI are similar with each other and scale consistently to full-device utilization.

5.4. OSH matrix multiply

One of the applications from the OpenSHMEM test suite is a matrix-multiplication kernel. Unlike the previous matrix-multiplication kernel, this kernel implements a block-column distribution for computation and leverages a distributed data structure that divides up the three matrices among the PEs. This data distribution results in more communication time to obtain non-local elements of the B matrix to perform matrix multiplication, but the advantage is substantially lower memory use for increasing number of PEs. As a result, this approach sacrifices some runtime performance, but is more amenable for very large matrices. The communication in this application consists of a quadratic number of barriers and put operations with complexity $O(p \times r)$, where p is the number of PEs and r is the number of rows in the matrices. For further details, source code can be obtained from the OpenSHMEM test suite [10].

The performance of this kernel is shown in Figure 11d for the TILE-Gx. TSHMEM and OpenMP performance scale similarly on the device, with OpenMP showing a slight performance improvement at 32 PEs. This result is attributed to the amount of data movement in the SHMEM version. In the SHMEM version, each PE exchanges data by copying it into another PE's shared partition at the end of each compute iteration. The OpenMP version, however, does not require this step because the data can be accessed directly via data sharing. While the OpenMP approach is more amenable for an SMP device, the SHMEM approach was implemented for operation on a distributed system, as the data cannot be accessed directly and must be transferred with one-sided operations. A different SHMEM implementation would be capable of accessing the data directly, but would only be applicable on SMP devices as a result. Interestingly, OSHMPI performance is consistently worse than OpenMP and TSHMEM even at two PEs and stops scaling after 16 PEs (half-device utilization). This kernel is the only application in this section that exhibits this behavior. This result is attributed to the amount of communication in this application and exposes potential scalability issues with the application itself. The amount of communication depends on p, the number of PEs; therefore,

DOI: 10.1002/cpe

additional PEs will increase both the amount and duration of communication operations. OSHMPI is significantly affected by this behavior with its higher-latency barriers compared with TSHMEM. Finally, the execution times from Figures 11c and 11d show that this implementation of matrix multiplication is 1.5 to 2 times slower than the previous matrix multiplication because of the pattern of computation.

5.5. OSH heat image

This application takes width and height parameters as inputs and solves a heat-conduction modeling problem. Each PE is assigned a block of rows and assists in performing iterative heat-conduction computation in order to generate an output image. The SHMEM communication for this application consists of a linear number of put and barrier operations based on the number of iterations in the modeling problem. The OpenMP version consists of a linear number of barriers and critical-section regions.

Execution times are shown in Figure 12a. OpenMP, TSHMEM, and OSHMPI observe similar performance until 16 PEs. TSHMEM continues to scale,while both OpenMP and OSHMPI exhibit a slight degradation in scaling at 32 and 36 PEs. This application demonstrates favorable performance for TSHMEM at full-device utilization. Because the entire input data are implicitly shared with OpenMP, synchronization operations such as barriers become more numerous and costly than with SHMEM's distributed approach to data partitioning and ownership. Additionally, as the number of PEs increases, the additional synchronization points while iterating on the heat-image model results in this decrease of performance for OpenMP. This slight decrease also applies with TSHMEM, but is less impactful on the overall application performance than in the OpenMP case. For OSHMPI, the majority of the time difference compared with TSHMEM is due to higher-latency barrier synchronization.

We present results for OSH heat image with increasing input sizes in Table III. TSHMEM shows a maximum performance improvement of 30% over OpenMP at inputs of 2048 × 2048, with an improvement of 18% for larger input sizes. This performance improvement is significant and

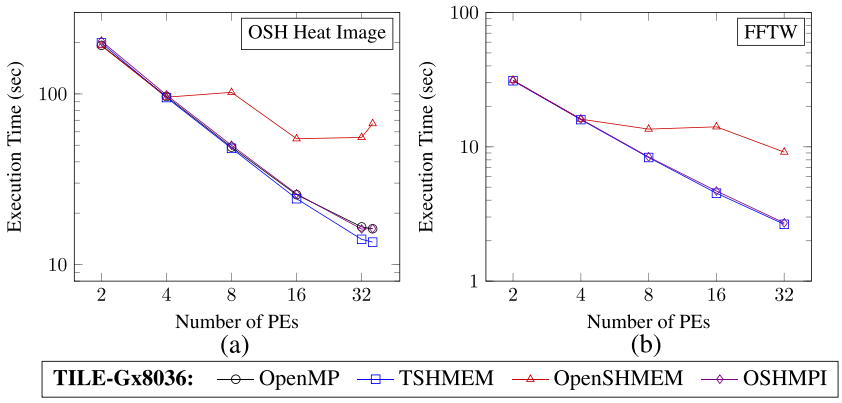


Figure 12. Execution times for (a) OSH heat image (1024×1024 with 5000 iterations) and (b) parallelization of FFTW (8192 FFT operations on 8192-length complex-float arrays).

Table III. Performance of OSH Heat Image at 36 cores for varying problem sizes.

	TSHMEM	Compared to OpenMP		Compared to OSHMPI	
Problem Size	Time (s)	Time (s)	**Improvement (%)**	Time (s)	**Improvement (%)**
1024 × 1024	13.5	16.2	17	16.0	16
2048 × 2048	61.2	87.6	30	67.4	9.3
4096 × 4096	347.6	440.6	21	355.4	2.2
8192 × 8192	1568.8	1918.8	18	1581.1	0.78
16384 × 16384	6313.6	7717.2	18	6371.1	0.90

DOI: 10.1002/cpe

is not intuitively conveyed via the logarithmic-scale graph in Figure 12a. Comparing TSHMEM with OSHMPI, TSHMEM's performance improvement decreases as the problem size increases, but the execution time differences between TSHMEM and OSHMPI show an increasing trend. At 8192×8192, the time difference is 12.3 s in favor of TSHMEM over OSHMPI. The time difference at 16384×16384 is 57.5 s, an increasing trend favoring TSHMEM. This trend indicates that the *rate of growth* for the time difference is positive, but is slower than the rate of growth of the raw execution time for problem sizes less than 16384×16384. As a result, TSHMEM exhibits higher scalability than OSHMPI and each percentage point of improvement becomes more significant as problem sizes increase. Performance improvement of TSHMEM begins to increase at 16384×16384, indicating that the rate of growth for the time difference is now faster than the rate of growth of the execution time.

5.6. Distributed FFT with SHMEM and FFTW

The final application involves the process-based parallelization of a popular fast Fourier transform (FFT) library, FFTW [30]. The application performs a distributed, one-dimensional, discrete Fourier transform (DFT) using the FFTW library, with data setup and inter-process communication via SHMEM. While the FFTW library is already multithreaded internally, this application uses SHMEM instead of MPI to handle inter-process communication via fast one-sided puts to quickly exchange data for a distributed system. An OpenMP implementation was not tested for this application.

The execution times are shown in Figure 12b for the TILE-Gx. This application executes in three phases: (1) DFT operation with twiddle calculations and data exchange, (2) matrix transpose, and (3) DFT operation. All of the SHMEM communication occurs in phase one during data exchange and, for each PE, consists of a linear number of put operations and a computational barrier. TSHMEM and OSHMPI execution times are similar and achieve full-device scaling, with TSHMEM demonstrating a slight performance advantage over OSHMPI because of higher-performance put and barrier operations. OpenSHMEM is able to achieve a moderate amount of scalability, but not to the extent of either TSHMEM or OSHMPI.

6. CONCLUSIONS AND FUTURE WORK

In exploring PGAS semantics for modern many-core processors, we have presented and evaluated our design and analysis of TSHMEM, a high-performance OpenSHMEM library built atop Tilera-provided libraries for the Tilera TILE-Gx and TILE*Pro* many-core architectures. The current TSHMEM design provides for all of OpenSHMEM functionality, excluding static-variable support for atomic operations. Our analysis of TSHMEM serves as an evaluation basis for low-level PGAS semantics and performance on modern and emerging many-core processors with the intent of enabling similar libraries to deliver higher utilization and performance for current-generation and next-generation many-core systems.

Performance of TSHMEM is demonstrated with microbenchmarks of Tilera-library and TSHMEM functions, offering direct validation of realizable performance and any inherited overhead. Results indicate that TSHMEM designs for dynamic symmetric-variable transfers display minimal overhead with underlying Tilera libraries and that numerous SHMEM functions outperform those from the OpenSHMEM reference implementation and from OSHMPI atop MPICH. Additionally, the design for barrier synchronization in TSHMEM is shown to be fast relative to several available Tilera barrier primitives for both the TILE-Gx and TILE*Pro*. In comparing the performance of TSHMEM collectives, the communication algorithms that emphasize cache locality by coalescing results onto a single tile surprisingly performed better than the algorithms that focused on linearly distributed communication.

Performance, portability, and scalability of SHMEM applications for the TILE-Gx are illustrated via numerous application case studies comparing TSHMEM performance with OpenMP, the OpenSHMEM reference implementation, and OSHMPI. Our experiments exhibited application-scalability concerns with the OpenSHMEM reference implementation because of generic instru-

DOI: 10.1002/cpe

mentation for TILE-Gx used by its underlying GASNet communications runtime. As a result, we focus our experiments on analyzing performance behavior with TSHMEM, OpenMP, and OSHMPI. For application scalability, TSHMEM, OpenMP, and OSHMPI exhibit similar trends, but when exploring different problem sizes at full-device utilization, TSHMEM demonstrates a marginal to significant performance improvement. This conclusion provides validation to a bare-metal library design for TSHMEM on many-core devices.

Future work for TSHMEM will include further library optimizations in conjunction with exploration of extensions for the OpenSHMEM standard specification. With the resurgence of interest in SHMEM and OpenSHMEM, proposed extensions such as threading support [31] merit investigation with impact on performance and API semantics. Finally, we plan to diversify our PGAS design exploration with TSHMEM to include devices such as the Intel Xeon Phi many-core coprocessor [32], and expand our design on the TILE-Gx with novel architectural features such as the mPIPE packet engine as we explore the shared-memory abstraction in TSHMEM across multiple many-core devices.

ACKNOWLEDGEMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022.

REFERENCES

1. Lam BC, George AD, Lam H. TSHMEM: Shared-memory parallel computing on Tilera many-core processors. In *Proceedings of 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13. IEEE Computer Society: Washington, DC, USA, 2013; 325–334.
2. Lam BC, Barboza A, Agrawal R, George AD, Lam H. Benchmarking parallel performance on many-core processors. In *Openshmem and Related Technologies. Experiences, Implementations, and Tools*, vol. 8356, Poole S, Hernandez O, Shamis P (eds)., Lecture Notes in Computer Science. Springer International Publishing: Cham, Switzerland, 2014; 29–43.
3. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 1996; **22**(6):789–828.
4. Dagum L, Menon R. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science Engineering* 1998; **5**(1):46–55.
5. Silicon Graphics International Corp. SHMEM API for parallel programming. (Available from: http://www.shmem.org/) [Accessed on 17 May 2015].
6. Barriuso R, Knies A. SHMEM user's guide for C. *Technical Report*, Cray Research Inc., 1994.
7. Message Passing Interface Forum. MPI: a message-passing interface standard, version 3.0, 2012. (Available from: http://www.mpi-forum.org/docs/mpi-3.0/) [Accessed on 17 May 2015].
8. Chapman B, Curtis T, Pophale S, Poole S, Kuehn J, Koelbel C, Smith L. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of 4th Conference on Partitioned Global Address Space Programming Models*, PGAS '10. ACM: New York, NY, USA, 2010; 2:1–2:3.
9. OpenSHMEM. OpenSHMEM API, v1.0 final. (Available from: http://www.openshmem.org/) [Accessed on 17 May 2015].
10. University of Houston. OpenSHMEM source releases. (Available from: http://openshmem.org/site/Downloads/Source) [Accessed on 17 May 2015].
11. The Ohio State University. MVAPICH2-X: Unified MPI+PGAS communication runtime over OpenFabrics/Gen2 for exascale systems. (Available from: http://mvapich.cse.ohio-state.edu/overview/) [Accessed on 17 May 2015].
12. Hammond JR, Ghosh S, Chapman BM. Implementing OpenSHMEM using MPI-3 one-sided communication. In *Openshmem and Related Technologies. Experiences, Implementations, and Tools*, vol. 8356, Poole S, Hernandez O, Shamis P (eds)., Lecture Notes in Computer Science. Springer International Publishing: Cham, Switzerland, 2014; 44–58.
13. Portals OpenSHMEM implementation. (Available from: https://code.google.com/p/portals-shmem/) [Accessed on 17 May 2015].
14. Coti C. POSH: Paris OpenSHMEM, a high-performance OpenSHMEM implementation for shared memory systems. *Procedia Computer Science* 2014; **29**:2422–2431.
15. Cray Inc. Software for the Cray XK7 System. (Available from: http://www.cray.com/products/computing/xk-series?tab=software) [Accessed on 1 June 2015].
16. Mellanox Technologies. Mellanox ScalableSHMEM. (Available from: http://www.mellanox.com/related-docs/prod_software/PB_ScalableSHMEM.pdf) [Accessed on 17 May 2015].
17. Bonachea D. GASNet specification, v1.1. *Technical Report*, University of California at Berkeley: Berkeley, CA, USA, 2002.

DOI: 10.1002/cpe

18. Yoon C, Aggarwal V, Hajare V, George AD, Billingsley, III M. GSHMEM, a portable library for lightweight, shared-memory, parallel programming. *Proceedings of 5th Conference on Partitioned Global Address Space Programing Models*, PGAS '11, Galveston, TX, USA, 2011; 1–9.
19. Tilera Corporation. TILE-Gx36 multicore processor. (Available from: http://www.tilera.com/products/?ezchip=585&spage=621) [Accessed on 17 May 2015].
20. Tilera Corporation. TILE*Pro*64 processor family.
21. MPICH: high-performance portable MPI. (Available from: http://www.mpich.org/) [Accessed on 17 May 2015].
22. The Ohio State University. OSU micro-benchmarks. (Available from: http://mvapich.cse.ohio-state.edu/benchmarks/) [Accessed on 17 May 2015].
23. Ma T, Bosilca G, Bouteiller A, Dongarra JJ. Kernel-assisted and topology-aware MPI collective communications on multicore/many-core platforms. *Journal of Parallel and Distributed Computing* 2013; **73**(7):1000–1010. Best Papers: International Parallel and Distributed Processing Symposium (IPDPS) 2010, 2011 and 2012.
24. Graham RL, Shipman G. MPI support for multi-core architectures: optimized shared memory collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 5205, Lastovetsky A, Kechadi T, Dongarra J (eds)., Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2008; 130–140.
25. Mamidala AR, Kumar R, De D, Panda DK. MPI collectives on modern multicore clusters: performance optimizations and communication characteristics. *8th IEEE International Symposium on Cluster Computing and the Grid, 2008 ccgrid '08*, Lyon, France, 2008; 130–137.
26. Chan E, Heimlich M, Purkayastha A, van de Geijn R. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 2007; **19**(13):1749–1783.
27. Thakur R, Rabenseifner R, Gropp W. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications* 2005; **19**(1):49–66.
28. Jose J, Zhang J, Venkatesh A, Potluri S, Panda DK. A comprehensive performance evaluation of OpenSHMEM libraries on InfiniBand clusters. In *Openshmem and Related Technologies. Experiences, Implementations, and Tools*, vol. 8356, Poole S, Hernandez O, Shamis P (eds)., Lecture Notes in Computer Science. Springer International Publishing: Cham, Switzerland, 2014; 14–28.
29. Nishtala R, Yelick KA. Optimizing collective communication on multicores. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar '09. USENIX Association: Berkeley, CA, USA, 2009; 18–18.
30. Frigo M, Johnson SG. The design and implementation of FFTW3. *Proceedings of the IEEE* 2005; **93**(2):216–231.
31. Ten Bruggencate M, Roweth D, Oyanagi S. Thread-safe SHMEM extensions. In *Openshmem and Related Technologies. Experiences, Implementations, and Tools*, vol. 8356, Poole S, Hernandez O, Shamis P (eds)., Lecture Notes in Computer Science. Springer International Publishing: Cham, Switzerland, 2014; 178–185.
32. Intel Corporation. Intel Xeon Phi coprocessor 5110P. (Available from: http://ark.intel.com/products/71992/) [Accessed on 17 May 2015].

DOI: 10.1002/cpe