TSHMEM: Shared-Memory Parallel Computing on Tilera Many-Core Processors

Bryant C. Lam   Alan D. George   Herman Lam
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering
University of Florida
Gainesville, FL 32611-6200
{blam, george, hlam}@chrec.org

***Abstract*—With many-core processor architectures emerging, concerns arise regarding the productivity of numerous parallel programming tools, models, and languages as developers from a broad spectrum of science domains struggle to maximize performance and maintain correctness of their applications. Fortunately, a partitioned global address space (PGAS) programming model has demonstrated realizable performance and productivity potential for large parallel computing systems with distributed-memory architectures. One such PGAS approach is SHMEM, a lightweight, shared-memory programming library. Renewed interest for SHMEM has developed around OpenSHMEM, a recent community-led effort to produce a standardized specification for the SHMEM library amidst incompatible commercial implementations. This paper presents and evaluates the design of TSHMEM (short for TileSHMEM), a new OpenSHMEM library for the Tilera TILE-Gx8036 and TILE*Pro*64 many-core processors. TSHMEM is built atop Tilera-provided libraries with key emphasis upon realizable performance with those libraries, demonstrated through microbenchmarking. Furthermore, SHMEM application portability is illustrated with two case studies. TSHMEM successfully delivers high performance with ease of programmability and portability for SHMEM applications on TILE-Gx and TILE*Pro* architectures.**

Keywords*—parallel programming; performance analysis; high-performance computing; parallel architectures

I. Introduction

Parallel programming is experiencing explosive growth of demand due to processor architectures shifting toward many processing cores in an effort to maintain performance progression in the face of technological and physical limitations. With the emergence of many-core processors into high-performance computing (HPC), the development of parallel programming models, tools, and libraries is more essential than ever before.

HPC has traditionally focused on models such as message passing with MPI or shared memory with OpenMP, 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 Unified Parallel C (UPC), X10, Chapel, Co-Array Fortran (CAF), Titanium, and SHMEM.

In this paper, we present and evaluate the design of TSHMEM (TileSHMEM), a SHMEM library based on the OpenSHMEM version 1.0 specification [1]. TSHMEM delivers a high-performance many-core programming library that improves developer productivity and enables SHMEM application portability for the Tilera TILE-Gx and TILE*Pro* architectures. With the purpose of leveraging many-core capabilities and optimizations, TSHMEM is built atop Tilera-provided libraries with microbenchmarking employed in order to compare the realizable performance and overhead between those libraries and TSHMEM functionality [2].

The remainder of the paper is organized as follows. Section II provides background on the SHMEM library and standardization efforts via OpenSHMEM, our previous research with GSHMEM (i.e., SHMEM for clusters), and a brief introduction to Tilera's many-core architectures. Section III presents several microbenchmarking results on Tilera TILE-Gx8036 and TILE*Pro*64 processors. Section IV delves into the design of TSHMEM, with performance results and analysis for a subset of the OpenSHMEM specification. Section V presents two application studies with TSHMEM demonstrating portability and performance. Finally, Section VI provides conclusions and directions for future work.

II. 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 partitioned global address space. This section provides a brief background of SHMEM, GSHMEM, and Tilera, which form the foundation of our experience and design for TSHMEM.

A. 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 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 typically at the same offset relative to the start of each symmetric partition.

SHMEM provides a set of 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 provides support for atomic memory operations not available in popular library alternatives until recently (e.g., MPI 3.0).

Due to the lightweight nature of SHMEM, commercial variants have emerged from vendors such as Cray, SGI, and Quadrics. Application portability between variants, however, proved difficult due to different functional semantics, incompatible APIs, or system-specific implementations. This situation had regrettably fragmented developer adoption in the HPC community. Fortunately, SHMEM has recently seen renewed interest in the form of OpenSHMEM, a community-led effort to create a standard specification for SHMEM functions and semantics. Version 1.0 of the OpenSHMEM specification released on January 31, 2012 brings revived hope for widespread adoption. Vendors such as Mellanox are already providing library implementations based on OpenSHMEM [3].

B. GSHMEM

Our prior work with SHMEM involved design and evaluation of an OpenSHMEM library called GSHMEM (GatorSHMEM) [4] atop GASNet [5], a low-level networking layer and communications system with the goal of supporting SPMD parallel programming models, such as PGAS-related models and languages. GSHMEM targeted a draft version of the OpenSHMEM specification in order to evaluate its existing functionality and propose several new additions for future revisions. Built for cluster-based systems, experimental results via microbenchmarking with GSHMEM showed that performance is comparable to 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.

C. Tilera Many-Core Processors

Tilera Corporation, based in San Jose, California, develops commercial many-core processors with emphases on high performance and low power in the cloud computing, general server, and embedded devices 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 dynamic 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.

Table I. BASIC SUBSET OF OPENSHMEM FUNCTIONS.

Category	Example Functions
Environment	
Setup and Initialization	start_pes()
Environment Query	_my_pe(), _num_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()

Group 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()

Our research focuses on the new TILE-Gx8036 (Figure 1) with its predecessor, the TILE*Pro*64 (Figure 2), as baseline 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. These processors consist of the 36-core TILE*Pro*36 and 64-core TILE*Pro*64. The TILE-Gx is Tilera's new generation of 64-bit many-core processors. Differentiated by a substantially redesigned architecture, the TILE-Gx exhibits upgraded processing cores, improved iMesh interconnects, and novel on-chip accelerators. Each 64-bit processing core is now attached to five dynamic networks. The TILE-Gx family currently includes the 16-core TILE-Gx16 (TILE-Gx8016) and 36-core TILE-Gx36 (TILE-Gx3036 and TILE-Gx8036). In addition, TILE-Gx provides 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.

III. 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.

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 with a single TILE-Gx8036 operating at 1 GHz, and the TILEncore*Pro*-64 with a single TILE*Pro*64 operating at 700 MHz. A host machine is required for PCI-card platforms such as the TILEncore*Pro*-64, while it is an option for standalone server platforms such as the TILEmpower-Gx.

A. *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 fulfilled from other tiles in order to keep on-chip as much memory as possible.

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 is more suitable for local or remote homing, but excels for 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.

B. *TMC 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 due to its significant use in one-sided data transfers.

Table II. ARCH. COMPARISON FOR TILE-GX8036 AND TILE*Pro*64.

TILE-Gx8036	**TILE*Pro* 64**
36 tiles of 64-bit VLIW processors 32k L1i, 32k L1d, 256k L2 cache per tile Up to 750 billion operations per second 60 Tbps of on-chip mesh interconnect	64 tiles of 32-bit VLIW processors 16k L1i, 8k L1d, 64k L2 cache per tile Up to 443 billion operations per second 37 Tbps of on-chip mesh interconnect
Over 500 Gbps memory bandwidth 1.0 to 1.5 GHz operating frequency 10 to 55W 2 DDR3 memory controllers	200 Gbps memory bandwidth 700 and 866 MHz operating frequency 19 to 23W @ 700 MHz 4 DDR2 memory controllers
mPIPE for wire-speed packet processing MiCA for crypto and compression	

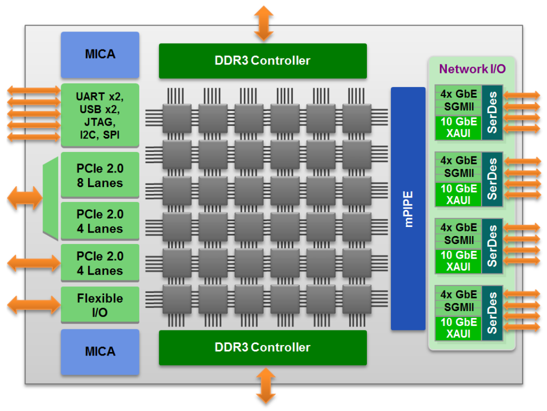


Figure 1. TILE-Gx8036 architecture diagram [6].

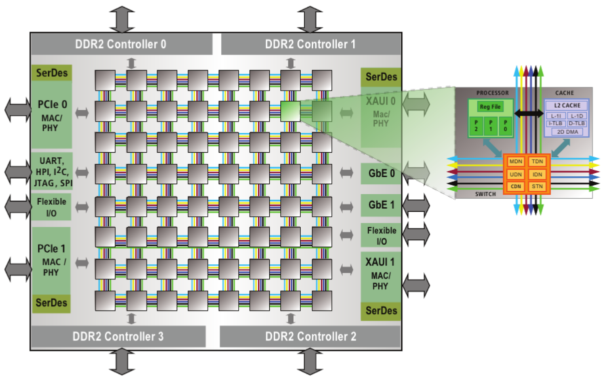


Figure 2. TILE*Pro*64 architecture diagram [7].

Figure 3 shows microbenchmark results for *memcpy()* operations between statically allocated private heap memory and shared-memory segments via TMC common memory. Effective bandwidth on the TILE-Gx36 is much higher than on TILE*Pro*64 for transfers smaller than 2 MB. 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,

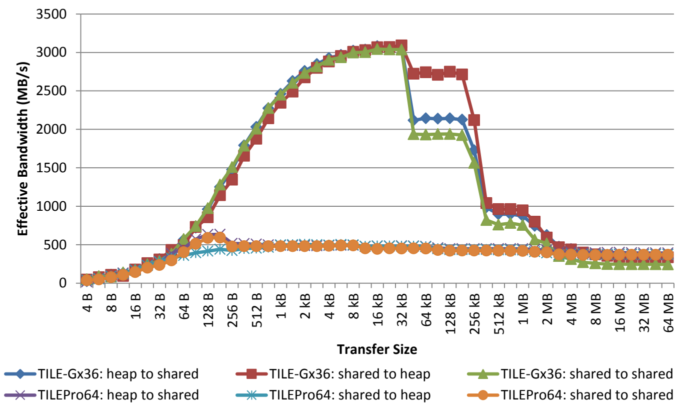


Figure 3. Effective bandwidth for shared-memory copy operations on TILE-Gx36 and TILE*Pro*64.

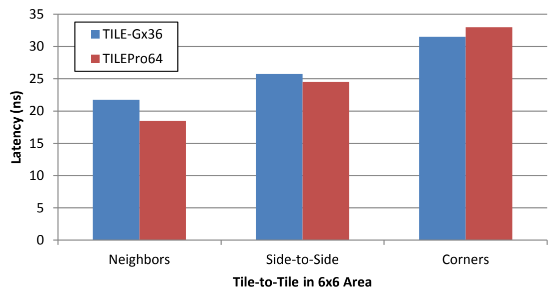


Figure 4. Average one-way latencies on UDN between adjacent tiles (neighbors), tiles across the area (side-to-side), and tiles on opposite corners of the effective area (corners). TILE-Gx36 has higher latency due to setup-and-teardown on a 64-bit switching fabric vs. TILE*Pro*64's 32-bit fabric.

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 significant 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 3100 MB/s, and the L2 cache performance reaches a peak between 1900 MB/s and 2700 MB/s. The third performance transition on TILE-Gx36 is attributed to Tilera's L3 DDC. Effective bandwidth decreases from 1000 MB/s as transfer sizes beyond 1 MB begin exceeding the L2 caches of nearby tiles from DDC, converging at 320 MB/s in memory-to-memory transfers. 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 500 MB/s through the L1d and L2 cache sizes and decreases into memory-to-memory transfers (370 MB/s). Memory-to-memory transfers on the TILE*Pro*64, however, are faster than those on the TILE-Gx36.

C. TMC UDN 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 1-word header to each payload with information about the destination tile and transfer the data packet via the UDN—at a rate of 1 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, *side-to-side* 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 1-word send and a 1-word acknowledgment from the receiver. Average one-way latencies are depicted in Figure 4.

Table III. ONE-WAY LATENCIES ON UDN.

Type (6x6 area)	Direction	Sender	Receiver	Time (ns)	
				TILE-Gx36	TILE*Pro* 64
Neighbors	left	14	13	21	19
	right	14	15	22	19
	up	14	8	22	18
	down	14	20	22	18
	left	28	27	21	19
	right	28	29	22	19
	up	28	22	22	18
	down	28	34	22	18
Side-to-Side	right	6	11	26	25
	left	11	6	25	25
	down	1	31	26	24
	up	31	1	26	24
	right	23	18	25	25
	left	18	23	26	25
	down	33	3	26	24
	up	3	33	26	24
Corners	down-right	0	35	32	33
	up-left	35	0	31	33
	down-left	5	30	31	33
	up-right	30	5	32	33

Individual results in Table III show that each case has consistent latencies with low variance of 1 ns between results, regardless of direction. This data indicates that the average can be interpreted as representative performance. Note that virtual CPU numbers in Table III are used for Sender and Receiver tiles because the physical CPU numbers for these tiles are not the same between the TILE-Gx36 and TILE*Pro*64 due to different chip dimensions. The virtual CPU numbers are equal to the physical CPU numbers on the TILE-Gx36, as the chip dimensions are equal to the test area, but the 6×6 test area is a subset of the 8×8 chip on the TILE*Pro*64. As a result, virtual CPU numbers must be converted to corresponding physical CPU numbers on TILE*Pro*64 (e.g., virtual tile 6 is physical tile 8) if actual tile positions are desired.

Each varying-distance 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 1 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; therefore the estimated setup-and-teardown time is roughly 21 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 slower average latency for the neighbor-to-neighbor and side-to-side cases. These latency tests have focused on

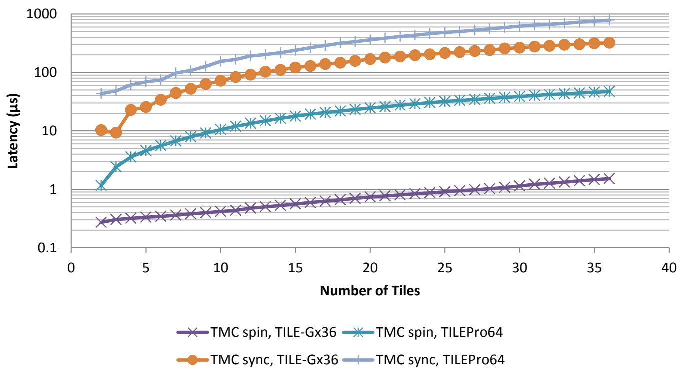


Figure 5. Latencies of TMC spin and sync barriers.

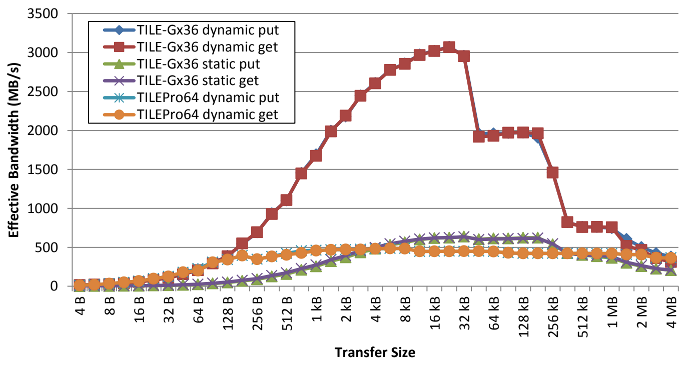


Figure 6. Effective bandwidth of TSHMEM put/get transfers for dynamic-dynamic on TILE-Gx36 and TILE*Pro*64 and static-static on TILE-Gx36.

minimum-sized payloads, but actual data transferred is doubled on TILE-Gx due to a 64-bit switching fabric compared to 32-bit on TILE*Pro*. Effective data throughput for neighbor, side-to-side, and corner-to-corner cases is 2900, 2500, and 2000 Mbps on TILE-Gx36 and 1700, 1300, and 980 Mbps on TILE*Pro*64.

D. *TMC 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.

Latency results for spin and sync barriers are shown in Figure 5. As expected, spin barriers vastly outperform sync barriers due to their polling nature, with latencies of 1.5 µs and 47.2 µs at 36 tiles for the TILE-Gx36 and TILE*Pro*64, respectively, compared to 321 µs and 786 µs. Futhermore, the spin barrier for the TILE-Gx significantly outperforms the TILE*Pro*'s. Since SHMEM focuses on low-overhead, low-latency performance, the substantial effort from Tilera in reducing the barrier latencies when transitioning to the TILE-Gx makes the TMC spin barrier for TILE-Gx an appealing candidate for use in TSHMEM.

IV. 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 currently targets the OpenSHMEM V1.0 specification and implements the functions required by all SHMEM applications. The subsections below are ordered categorically according to Table I, each including design description and performance results. Open issues and proposed extensions to OpenSHMEM are discussed in the final subsection.

A. *Environment Setup and Initialization*

SHMEM implementations typically consist of the library to which applications are linked against and an executable launcher which 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 partition. Memory is kept implicitly symmetric by the constraints imposed when using *shmalloc()*, requiring applications call the routine on all PEs with the same size argument at the same location in the program execution path.

B. *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. Semantics for put operations are non-blocking, returning from the function once the memory request is made and the data transfer is in-flight. Get operations, however, are blocking and will not return until the requested memory is visible to the local tile.

1) Dynamically Allocated Symmetric Objects: Due to the symmetry of each partition, a tile 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.

Figure 6 shows the effective bandwidth for dynamic-dynamic put/get transfers in TSHMEM. Note that put perfor-

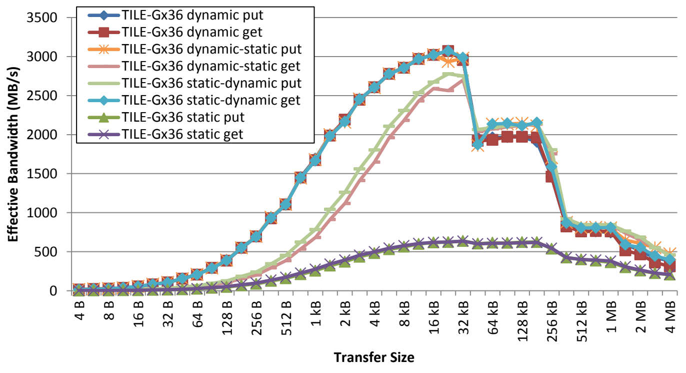


Figure 7. Effective bandwidth of TSHMEM put/get transfers on TILE-Gx36, emphasizing performance with static symmetric variables.

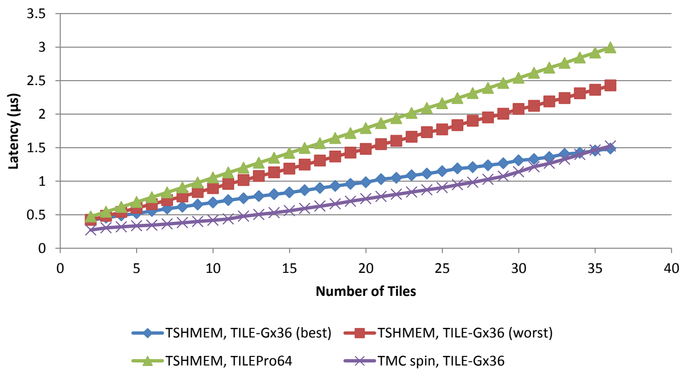


Figure 8. Latencies of TSHMEM barrier.

mance closely aligns with get performance for both the TILE-Gx36 and TILE*Pro*64. The dynamic-dynamic put/get design in TSHMEM demonstrates low overhead as the realizable performance for both devices closely matches the shared-to-shared performance from the common memory microbenchmark in Figure 3. TSHMEM uses the hash-for-home strategy for common memory. Static-static transfers for TILE-Gx36 are also presented in this figure as a comparison to TILE*Pro*64 performance and will be discussed in the next subsection.

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 due to the reality that the virtual addresses of the program heap are identical when parallel processes are instantiated with the same executable. Unfortunately, the heap space resides in private memory of a process and is not directly accessible to other processes.

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 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. In the case when both target and source addresses point to static symmetric variables, neither local or remote tile will be able to service the operation. A temporary shared-memory buffer is created to assist in the transfer, incurring an additional memory copy operation as overhead. Static symmetric variable transfers in TSHMEM are not currently supported on the TILE*Pro* architecture due to lack of support for UDN interrupts.

Figure 7 shows effective bandwidth performance of put/get operations on TILE-Gx36 with different combinations of dynamic and static symmetric variables as either the target or the source array. The precise notation in the legend is *target–source* (e.g., dynamic-static indicates an operation with a dynamic target and static source). For put operations, any source variable may be used (symmetric or otherwise) if the target variable is dynamic. Likewise, get operations may use any target variable provided the source variable is dynamic. As a result, dynamic-static puts and static-dynamic gets exhibit the same performance as their dynamic-dynamic counterparts, which demonstrates low overhead in comparison with the performance of the common memory microbenchmark in Figure 3.

Static-dynamic puts and dynamic-static gets cannot be performed by the local tile. Fortunately, the local tile can interrupt the remote tile and force it to service the request with minor performance degradation. Major performance penalty is incurred only in the static-static case when temporary shared-space allocation is required to aid in the transfer.

C. Synchronization

TSHMEM provides several categories of synchronization: barrier sync; communication sync with fence/quiet; and point-to-point sync (waiting until a variable's value changes).

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 5 illustrate that using sync barriers is not feasible due to 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. Another design was evaluated whereby the start tile broadcasts the *release* signal; however, latencies were two times slower.

The performance of TSHMEM barriers is shown in Figure 8. Since barrier latency is dependent on whether a tile leaves the routine first or last relative to the other tiles, best-case and worst-case barrier results for TILE-Gx36 are shown. For comparison, the microbenchmark results for the TMC spin barrier on TILE-

Gx36 from Figure 5 are also illustrated. Due to the higher clock frequency of TILE-Gx, its barrier outperforms the TILE*Pro*'s barrier. The TILE*Pro*64's TSHMEM barrier with a latency of 3 µs at 36 tiles vastly outperforms its corresponding TMC spin barrier (47.2 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.

2) Fence/Quiet: Since put operations are non-blocking, the communication synchronization routines *shmem_fence()* and *shmem_quiet()* ensure outstanding puts are completed before returning. The *shmem_fence()* routine guarantees put ordering to individual PEs, whereas *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. Additionally, *shmem_fence()* is simply set as an alias of *shmem_quiet()*, providing it the stronger semantics.

D. Collective Communication

SHMEM collective routines provide group-based communication for a subset of tiles. Collective designs and performance results for TSHMEM are discussed below.

1) Broadcast: Broadcast is a one-to-all operation where the active set of PEs obtains data from a root PE. TSHMEM currently has designs and performance results for push-based and pull-based implementations.

The push-based broadcast is performed by having the root PE perform a put operation sequentially to all other PEs. Figure 9 shows the aggregate effective bandwidth, the summation of each participating tile's bandwidth, for push-based broadcasts as transfer size and number of participating tiles increase. Second-column subfigures show performance at various numbers of tiles (up to 36). This design exhibits scalability issues as the performance does not increase as the number of participating tiles are increased.

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 instead of the root PE performing all of the work as is the situation with push-based. Figure 10 shows results from this design, demonstrating low overhead and effectiveness at taking advantage of the abundant iMesh bandwidth. Total broadcast bandwidth on the TILE-Gx36 reaches a maximum of 46 GB/s at 29 tiles and performs up to 37 GB/s at 36 tiles. Bandwidth on the TILE*Pro*64 peaks at 5.1 GB/s for 36 tiles.

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. OpenSHMEM 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 and when 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 11 shows TSHMEM results for a naive fast collect design leveraging pull-based broadcast. All PEs perform a put operation, sending 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 transfer M bytes to the root PE, and (2) root PE broadcasts $(n \times M)$ bytes to n 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 increase because each PE receives a copy of the entire concatenated result containing arrays from all other PEs. This effect is prominently illustrated between Figures 9 and 11 as the push-based broadcast experiences peaks in performance at the same data transfer sizes regardless of the number of tiles, whereas the fast collect performance peaks are shifting toward smaller data sizes as the number of tiles increases.

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 employs a naive reduction design in which a root PE continuously gets data from a remote PE and performs the reduction operation with it and the current reduction outcome values, until all active-set PEs have participated. A pull-based broadcast is then executed to inform all other PEs to get the reduction outcome values.

Results for integer summation reduction are shown in Figure 12. Similar to the push-based broadcast, aggregate bandwidth remains constant regardless of the number of tiles participating in the operation. This performance is due to serialization of data retrieval and reduction processing on one tile, producing a peak aggregate bandwidth of 150 MB/s at 36 tiles on TILE-Gx36. These performance profiles provide baseline behavior for further algorithmic exploration.

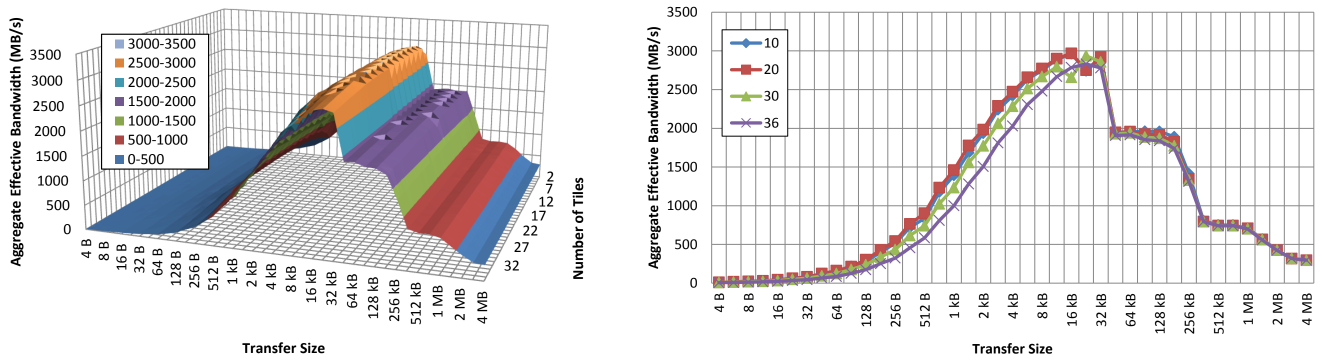
E. Open Issues and OpenSHMEM Extensions

There are several outstanding issues that we plan to resolve in future versions of TSHMEM. Currently, a few operations are missing support for static symmetric transfers. In addition, Figure 8 indicates that TSHMEM barrier latencies may be improved by leveraging the TMC spin barrier for TILE-Gx. Finally, we plan to explore and compare performance for different collective algorithms, such as binomial broadcast and recursive doubling, as we optimize the existing library.

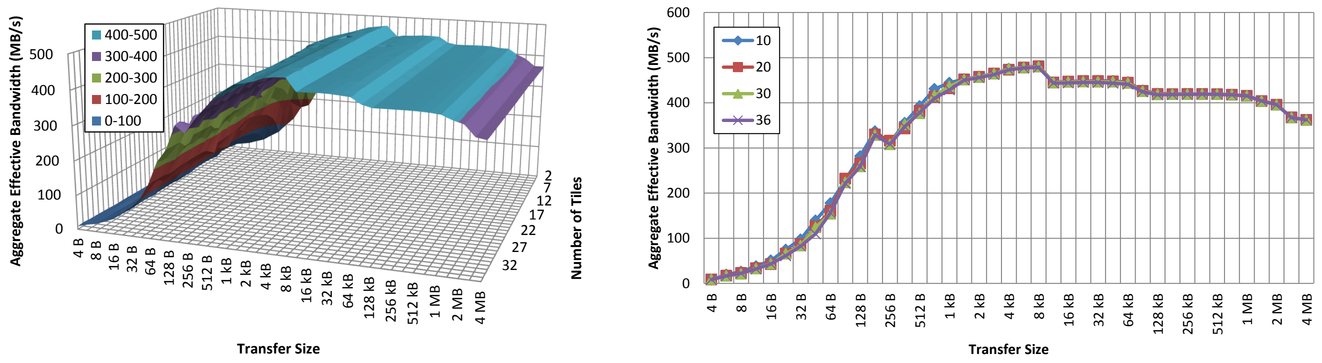
We also propose the inclusion of a *shmem_finalize()* routine into the OpenSHMEM standard in order to properly perform teardown of system resources. The OpenSHMEM standard does not currently provide a way for applications to inform the system to properly clean up and terminate. Instead, SHMEM applications typically rely on resource reclamation by the operating system after program termination. However, in the case of Tilera's UDN, platform instability or lockup may occur if it is not properly disengaged after active use.

V. Application Case Studies

One of the goals of an OpenSHMEM standard specification is to enable application portability across compliant libraries. TSHMEM adheres to this principle and demonstrates many-core performance, portability, and scalability with two application case studies. These applications emphasize equal-scale performance differences between the processor architectures of the TILE-Gx and the previous generation's TILE*Pro*.

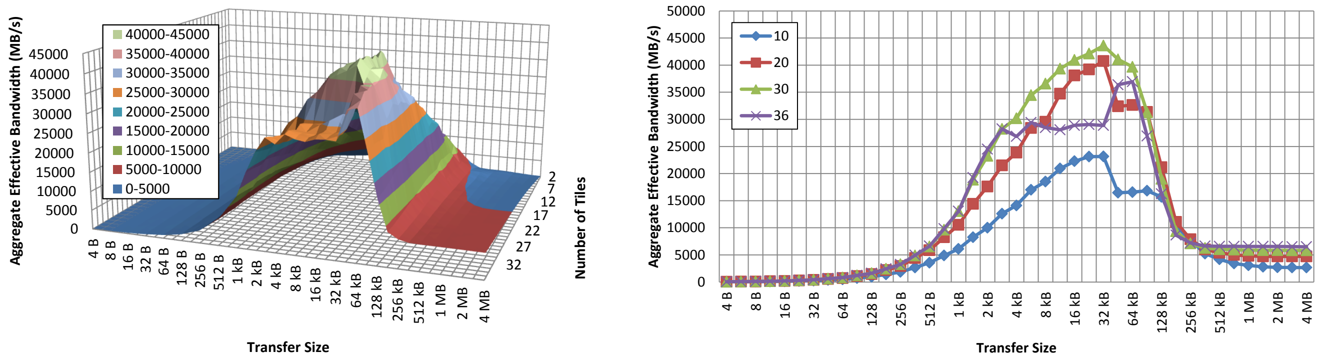


(a)

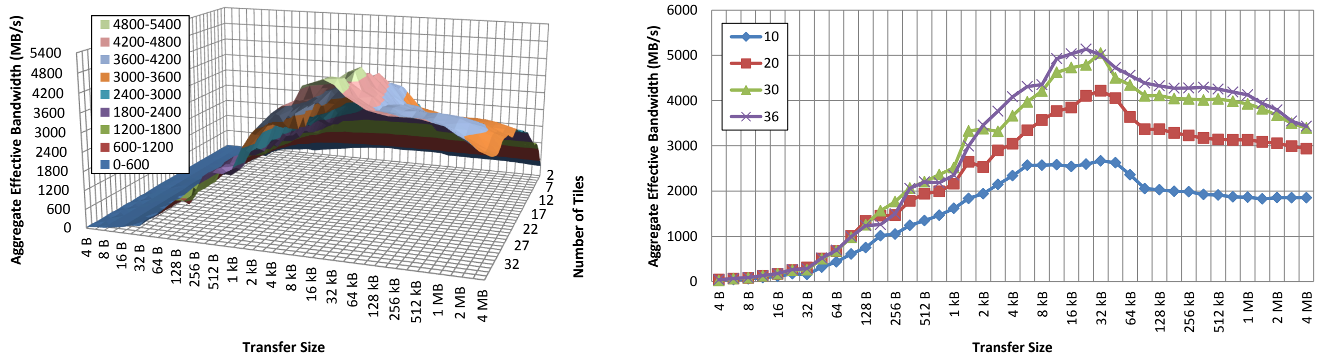


(b)

Figure 9. Push-based broadcast performance on (a) TILE-Gx36, (b) TILE*Pro*64.

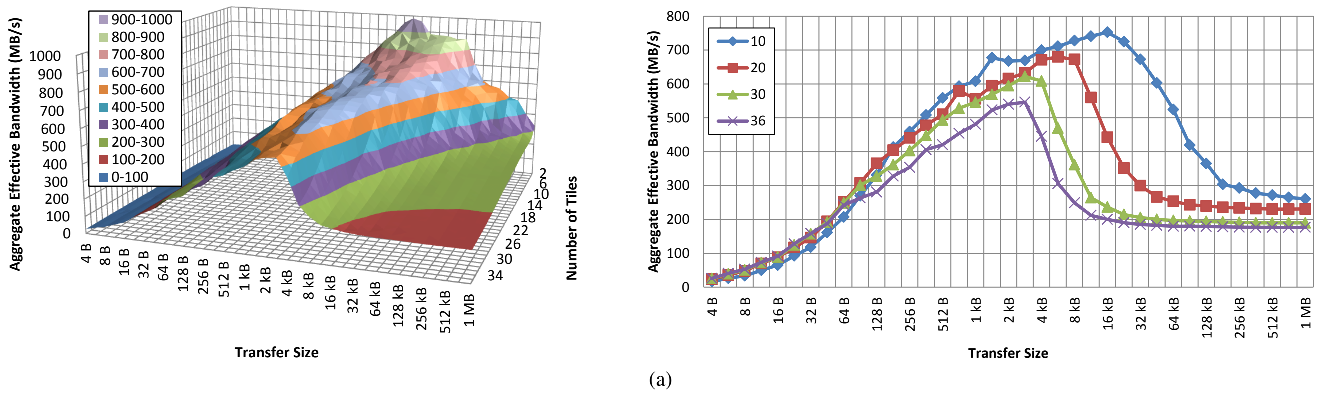


(a)

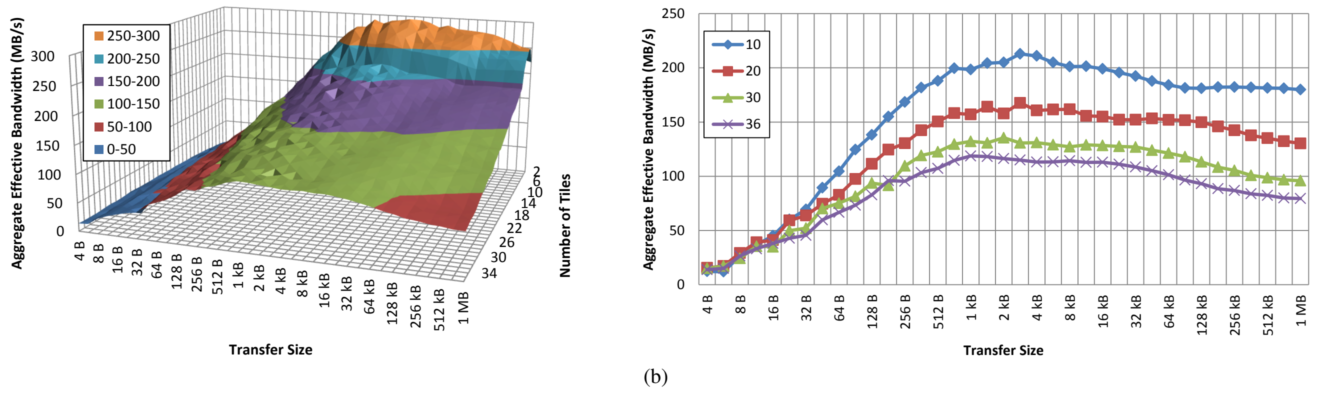


(b)

Figure 10. Pull-based broadcast performance on (a) TILE-Gx36, (b) TILE*Pro*64.

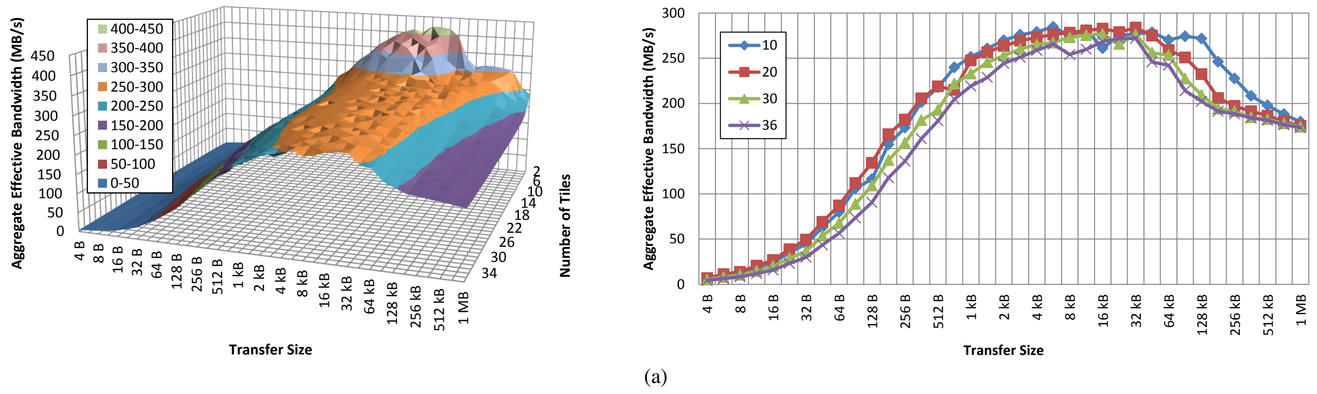


(a)

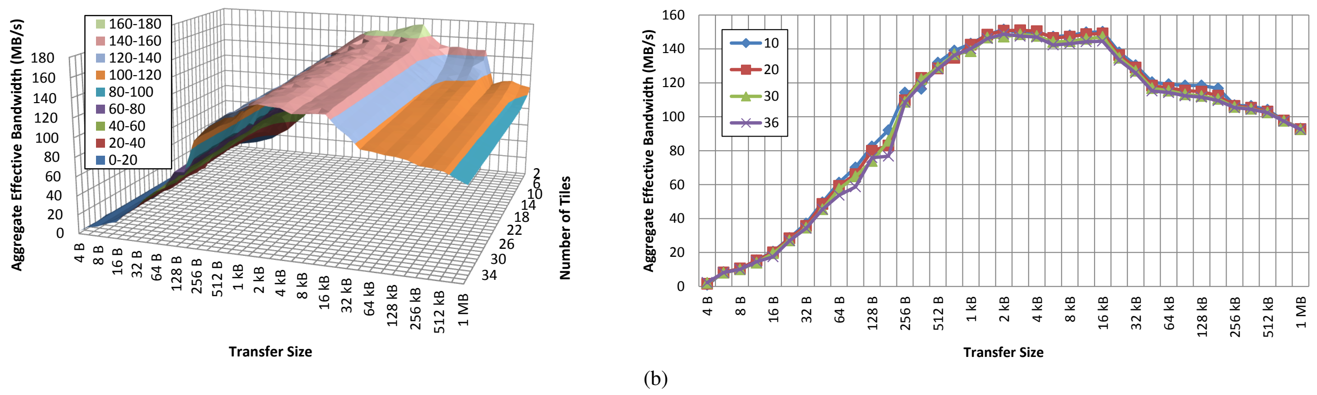


(b)

Figure 11. Fast collection performance on (a) TILE-Gx36, (b) TILE*Pro*64.



(a)



(b)

Figure 12. Integer summation reduction performance on (a) TILE-Gx36, (b) TILE*Pro*64.

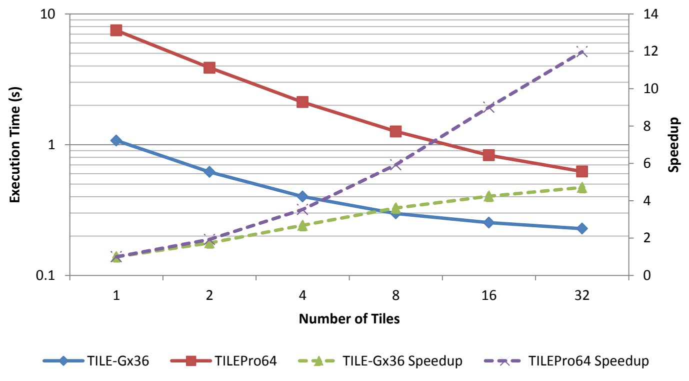


Figure 13. 2D-FFT on 1024×1024 complex floats.

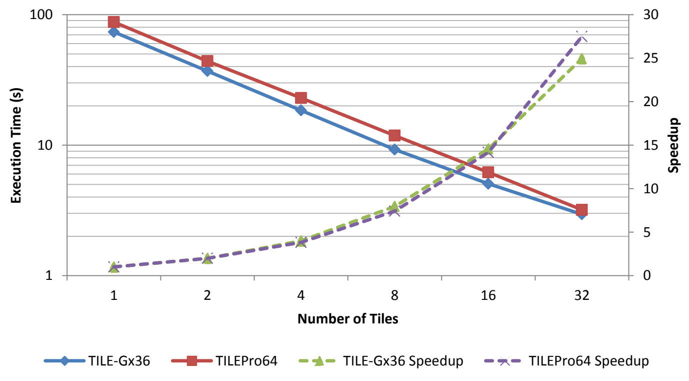


Figure 14. CBIR on 22,000 8-bit images of 128×128.

A. *2D Fast Fourier Transform (FFT)*

Performing a 2D-FFT on an image involves executing 1D-FFT operations over the image's rows, followed by the columns. This parallel 2D-FFT application distributes the image's rows to multiple PEs, with each PE executing a 1D-FFT over its subset of rows. A distributed transpose operation then redistributes the data across all PEs in an all-to-all communication operation, allowing each PE to now perform a 1D-FFT over the image's columns. One final matrix transpose produces the output image.

Figure 13 shows execution times and speedup for this parallel 2D-FFT. Due to computational serialization in the application's final transpose stage, speedup on TILE-Gx begins to level off around 5. Parallelization of this final transpose is left for future work. Execution times are 0.23 and 0.62 seconds at 32 tiles for TILE-Gx36 and TILE*Pro*64, respectively. TILE-Gx36 execution times are much faster (roughly an order of magnitude) than those on TILE*Pro*64 due to improved floating-point performance on the TILE-Gx.

B. *Content-based Image Retrieval (CBIR)*

Content-based image retrieval employs feature vectors to characterize images, facilitating efficient searches on large image databases using the semantics of an image. Figure 14 shows the execution times and speedup for a color-feature-extraction CBIR application based on autocorrelogram [8]. Speedup for both devices increases linearly with the number of tiles until 16 tiles. At 32 tiles, speedup is 25 for TILE-Gx36 and 27 for TILE*Pro*64. Tilera tailored both devices for integer-based workloads and, as expected, the TILE-Gx36 has faster execution times in all cases. These results can be directly compared to our previous CBIR case study on two traditional cluster-based systems with Quadrics and InfiniBand interconnects [4].

VI. Conclusions and Future Work

We have presented and evaluated our software architecture and design for TSHMEM, a high-performance OpenSHMEM library built atop Tilera-provided libraries for their Tilera TILE-Gx and TILE*Pro* many-core architectures. The current TSHMEM design provides for all of OpenSHMEM functionality, excluding static-variable support for a few operations.

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 static symmetric-variable transfers can exhibit low overhead through operation redirection. 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*. Performance, portability, and scalability of SHMEM applications for these many-core devices are also validated with two case studies.

Future work of TSHMEM will focus on library optimizations and extensions, especially pertaining to collectives and memory-homing strategies. Benchmarking will be expanded to include TSHMEM comparisons with other libraries such as OpenMP and MPI. Finally, we plan to leverage novel architectural features of the TILE-Gx such as the mPIPE packet engine as we explore designs for expanding the shared-memory abstraction in TSHMEM across multiple many-core devices.

Acknowledgments

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022. The authors would like to thank Dr. Vikas Aggarwal, Fred Spieler, and Narayana Perla for their contributions to this work. Additional thanks go to Tilera Corporation for help with product technical descriptions.

References

[1] OpenSHMEM, "OpenSHMEM API, v1.0 final," 2012. [Online]. Available: http://www.openshmem.org/

[2] B. C. Lam, A. D. George, and H. Lam, "An introduction to TSHMEM for shared-memory parallel computing on Tilera many-core processors," in *Proceedings of the 6th Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '12. ACM, 2012.

[3] Mellanox Technologies, "Mellanox ScalableSHMEM," Sunnyvale, CA, USA, 2012. [Online]. Available: http://www.mellanox.com/related-docs/prod_software/PB_ScalableSHMEM.pdf

[4] C. Yoon, V. Aggarwal, V. Hajare, A. D. George, and M. Billingsley, III, "GSHMEM, a portable library for lightweight, shared-memory, parallel programming," in *Proceedings of the 5th Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '11. ACM, 2011.

[5] D. Bonachea, "GASNet specification, v1.1," University of California at Berkeley, Berkeley, CA, USA, Tech. Rep., 2002.

[6] Tilera Corporation, "TILE-Gx8036 processor specification brief," San Jose, CA, USA, 2012. [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8036_PB033-02.pdf

[7] ——, "TILE*Pro*64 processor product brief," San Jose, CA, USA, 2011. [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf

[8] J. Huang, S. Kumar, M. Mitra, W.-J. Zhu, and R. Zabih, "Image indexing using color correlograms," in *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, Jun 1997, pp. 762–768.