Hybrid PGAS Runtime Support for Multicore Nodes

Filip Blagojević, Paul Hargrove, Costin Iancu, Katherine Yelick

Lawrence Berkeley National Laboratory
{fblagojevic, phhargrove, cciancu, kayelick}@lbl.gov

Abstract

With multicore processors as the standard building block for high performance systems, parallel runtime systems need to provide excellent performance on shared memory, distributed memory, and hybrids. Conventional wisdom suggests that threads should be used as the runtime mechanism within shared memory, and two runtime versions for shared and distributed memory are often designed and implemented separately, retrofitting after the fact for hybrid systems. In this paper we consider the problem of implementing a runtime layer for Partitioned Global Address Space (PGAS) languages, which offer a uniform programming abstraction for hybrid machines. We present a new process-based shared memory runtime and compare it to our previous `pthreads` *implementation. Both are integrated with the GASNet communication layer, and they can co-exist with one another. We evaluate the shared memory runtime approaches, showing that they interact in important and sometimes surprising ways with the communication layer. Using a set of microbenchmarks and application level benchmarks on an IBM BG/P, Cray XT, and InfiniBand cluster, we show that threads, processes and combinations of both are needed for maximum performance. Our new runtime shows speedups of over 60% for application benchmarks and 100% for collective communication benchmarks, when compared to the previous implementation. Our work primarily targets PGAS languages, but some of the lessons are relevant to other parallel runtime systems and libraries.*

1. Introduction

To meet the growing demand for computing capability in an era where power density limits processor speed increases, modern systems rely on multicore processors as their building block. As the number of cores per chip grows, and memory grows more slowly, support for parallel programming models that provide shared memory abstractions becomes essential.

On large-scale parallel systems, most people are looking to hybrid programming models that combine the popular MPI [25] library with a shared memory model, such as OpenMP [11]. Meanwhile, Partitioned Global Address Space (PGAS) languages such as Titanium [29], Co-Array Fortran [24], Unified Parallel C (UPC) [6], X10 [13] and Chapel [4], offer the possibility of a single programming model that runs well across the shared and distributed memory features of the machine. The PGAS languages offer a uniform approach, but for performance and memory scaling, their runtime systems need to use a hybrid that takes advantage of shared memory when it exists. At the software level this primarily translates into a choice of mapping language-level task to either pthreads or processes. Threads offer the more natural and popular alternative, because threads share a single address space, while processes have disjoint address spaces by default, which means data transfers go through a network loopback.

In this paper we present an implementation of the Berkeley UPC runtime [3] that uses processes with shared memory bypass for intra-node communication. We evaluate the performance of the process approach when compared to a previous implementations based on POSIX threads (`pthreads`). We evaluate HPC systems with Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) nodes across three networks: InfiniBand, Cray XT5 and IBM BG/P. Our workload is representative for the scientific computing domain and it contains microbenchmarks, application benchmarks implemented with fine-grained communication, implementations of the NAS Parallel Benchmarks [21] using bulk communication, as well as optimized implementations of collective operations.

The choice of processes or `pthreads` requires different runtime software engineering techniques, affects application performance and poses different requirements when programming model or parallel library interoperability is desired. Contemporary high performance networking APIs multiplex requests (with mutual exclusion for thread safety) generated by `pthreads` within a process while allowing more isolation between multiple processes. Our results indicate that process based implementations are essential for achieving high performance due to subtle interactions with the networking software stack: at the application level this manifests in better bandwidth achieved for small and medium message sizes. Process based implementations have a lower message injection overhead and are capable of sustaining a higher message injection rate. As shown by our results, high message injection rates can lead to performance degradation and therefore process based implementations are likely to require additional levels of communication throttling. We present microbenchmark results for the attainable network bandwidth that are highly correlated with the observed application behavior and given knowledge of the application characteristics can be used to choose the

appropriate runtime implementation. This study makes the following contributions:

- We designed and developed support for inter-process communication using shared memory in the Berkeley UPC runtime. To the best of our knowledge this is the first runtime implementation that can seamlessly and *efficiently* support combinations of process and `pthreads` on clusters.
- We categorize the most important design and performance related tradeoffs between the process-based and the `pthreads`-based implementations of a PGAS runtime.
- We show that for best performance and performance portability, hybrid implementations that support multiple processes per node with multiple `pthreads` per process are required.

In Section 3 we describe the design and implementation of the process-based shared memory communication in Berkeley UPC and discuss the performance tradeoffs in Section 3.1 and 4. The new process based implementation we present improves both the performance and the interoperability of the existing Berkeley UPC runtime: process based runtimes have fewer restrictions when combined with non-thread safe external libraries. These results are of interest to implementors of runtimes for large scale systems, as well as application developers, independent of the programming model: MPI, PGAS, X10 or Chapel.

2. Experimental Platforms

Table 1 presents our experimental platforms. *Ranger* [26] is a Sun Constellation Linux cluster containing NUMA quad-socket, quad-core AMD Opteron nodes connected by InfiniBand with a 1 GB/sec unidirectional point-to-point bandwidth. We also use a two-node InfiniBand cluster with quad-socket, quad-core Intel Tigerton UMA processors. The Cray XT5 cluster, (*Hopper* [22]) contains dual-socket quad-core AMD Opteron nodes connected with a Seastar2 Interconnect with a peak bidirectional bandwidth of 9.6 GB/s. The system runs a modified Linux operating system called Compute Node Linux (CNL) and the low level networking API exposed to applications is Portals. The IBM BG/P [18] contains quad-core PowerPC 450 nodes connected by multiple specialized networks. The nodes run a modified Linux kernel: Compute Node Kernel (CNK) and the low level networking API is the IBM DCMF.

The OpenIB and Portals low level APIs provide thread safety: any locking inside these libraries is beyond application or third party runtime control.

3. Shared Memory Communication in the UPC Runtime

PGAS languages have demonstrated [5, 23, 28] increased productivity due to high level control over data locality and layout and increased performance due to better exploitation of non-blocking communication and Remote Direct Memory Access (RDMA) support. These languages provide the abstraction of global shared memory: while any language-level task is allowed direct access to a global heap, each owns a part of this address space and it can access it without using the network.

For clusters of multicore processors, achieving good performance requires exploiting the shared memory within a node: the interaction between two tasks residing on the same node should bypass the network device. The current parallel programming models achieve this by either mapping language level tasks within one node to `pthreads`, or to processes with shared memory segments used for communication in the MPI case. No existing PGAS or MPI implementations allow hybrid mapping approaches (combinations of processes and `pthreads`) that exclusively use shared memory inside the cluster nodes. The MPI implementations do provide shared memory communication between processes on a node and place the communication bounce buffers within this region. In contrast, PGAS implementations have to expose a much larger region of memory and therefore face bigger implementation challenges due to restricted OS support.

Prior to this work, intra-node shared memory in the BUPC runtime was provided by mapping language-level tasks to `pthreads` within a single process. We have designed and implemented the Process SHared Memory (PSHM) mechanism to provide shared memory communication among processes that reside on the same node: besides completely bypassing the networking layer, PSHM allows hybrid execution models, where UPC level tasks are mapped to a mix of processes and `pthreads`. This paper argues that the mixing of processes and `pthreads` is required for performance and performance portability when implementing runtimes for modern cluster programming paradigms. In the rest of this study, we refer to the UPC shared memory process based runtime configuration as UPC-pshm, to the UPC `pthread` configuration as UPC-`pthreads`, and to the hybrid configuration as (multiple processes and `pthreads`) as UPC-hybrid.

3.1 PSHM Design

The Berkeley UPC implementation [3] uses a layered approach: language level abstractions are provided by a runtime which delegates communication and synchronization to the GASNet [2] layer. Several implementations of PGAS languages use GASNet as their communication layer: UPC, Titanium, Co-Array Fortran and Chapel.

The new PSHM implementation resides in GASNet and it provides: (1) shared memory communication through POSIX shared memory segments, and (2) a shared memory network abstraction for Active Messages [14] support. Existing MPI implementations may use SYSV, POSIX, or a disk file mapped via `mmap` to provide shared memory communication. Our initial implementation was based on the POSIX shared memory due to fewer restrictions on the amount of

	Processor	Clock GHz	Cores	NUMA	Network	Bandwidth
Tigerton	Intel Xeon E7310	1.6	16 (4x4)	no	Mellanox InfiniHost III Lx	10 GB/s
Ranger	4 quad-core AMD	2.3	16 (4x4)	socket	InfiniBand	1GB/s - unidirectional
Hopper	2 quad-core AMD	2.4	8 (2x4)	socket	Seastar2	9.6 GB/s - bidirectional
BG/P	4 PowerPC 450	0.85	4 (4x4)	no	Custom	5.1 GB/s

Table 1. *Architectural configuration of systems tested.*

allocated space: MPI needs to provide only buffer space for communication while PGAS languages need to export a significant portion of the address space. Currently, we also support the SYSV shared memory and a shared disk file mapped via *mmap*.

Most of our experimental results are obtained on Linux based systems which impose fewer restrictions on shared memory allocation. The Compute Node Kernel (CNK) installed on the IBM BG/P systems provides POSIX shared memory from a reserved pool under control of an environment variable. The Compute Node Linux (CNL) OS installed on the Cray XT series systems does not provide POSIX shared memory. On Cray XT we can use shared files mapped via `mmap`, or the SYSV shared memory. Note that the shared memory provided through a disk file can experience high startup overhead due to the time needed to allocate a file on disk. Also, some OSes might force the changes in the disk-shared file to be occasionally committed to the disk, but we did not detect this behavior on CNL (SYSV and POSIX shared memory do not experience these problems).

In PSHM each process allocates a shared memory segment and the global UPC heap is composed of the shared segments contributed by the individual processes. Current OSes impose limits on the amount of shared memory allocated by an individual process and we ensure space scalability using a distributed allocation mechanism. For portability and scalability reasons, the starting addresses of the shared memory segments are not aligned across individual process address spaces and the PSHM implementation contains per process data structures to store the starting addresses of the segments contributed by all PSHM processes. This data is needed to implement the UPC mandated pointer-to-shared arithmetic semantics associated with blocked data layouts.

GASNet contains an Active Messages (AM) [14] layer that provides both software portability and the mechanisms used in efficient implementations of synchronization, memory allocation, locks, collective and scatter/gather operations: building a lightweight AM communication layer is a requirement for achieving good scalability. Note that MPI implementations use internal mechanisms very similar to Active Messages.

The UPC-pshm implementation contains a separate PSHM-AM network layer to handle Active Messages traffic. The memory allocated for the PSHM-AM network is separated in two regions, one for request AMs and the other for reply AMs. Each region is split into multiple segments and each segment contains a number of queues equal to the total number of PSHM processes as illustrated in Figure 1. A

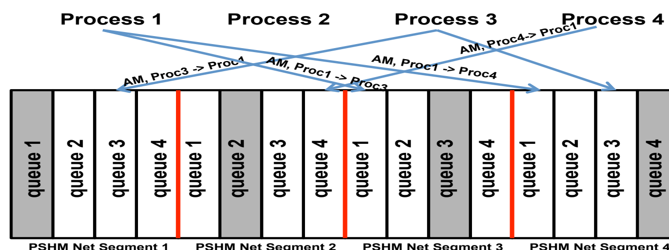


Figure 1. PSHM Request/Reply Structure, example with 4 processes. Each set of 4 queues represent incoming queues for one of the processes. All other queues are the outgoing queues for the same process.

segment corresponds to a single PSHM process, and is used for storing the incoming AM queues for that process. In the example presented in Figure 1, the message "*Process 1 to Process 3*" is written in the first queue of Segment 3. Message "*Process 3 to Process 1*" is written in the third queue of Segment 1. Every time an application-level task makes a runtime call, the implementation contains code that polls the AM network data structures for arrival of new messages.

The AM network data structures require $O(n^2)$ memory space, where n is the total number of UPC-pshm processes on *one* node. This space is required to store the AM header information and requires only 20B per header (though padded to a multiple of cache line size). The buffer space for each AM payload varies by network, but scales only *linearly* with the number of cores per node. AM performance under load is determined by the availability of payload space and at the current and near future core concurrency this term dominates the memory consumption of the implementation. For reference, on a 16 core machine we pre-allocate tens of MB of AM buffer space. The total amount of space is controlled by environment variables and we also provide flow control mechanisms when this space is temporarily exhausted. This design with statically pre-allocated buffers, also present in MPI implementations, provides a tradeoff between memory consumption and speed of operation. Based on our audit of GASNet and UPC constructs that rely on AMs, we believe that two buffer entries per process are enough to avoid deadlock. For example, for a possible future 100 core node, this translates into minimum space requirements of $\approx$ 20MB of memory. For much higher node concurrency, static AM buffer management might have to be replaced with dynamic management.

For inter- and intra-node communication, GASNet uses separate network layers: PSHM and the external network. The decision about which network will be used is performed inside GASNet and the communication data structure used in the inter-node case is network dependent; in most cases it

is encapsulated inside the low level communication library, e.g. the InfiniBand library or the OS kernel. The GASNet implementation polls on both inter- and intra-node networks in order to allow progress and avoid deadlock.

The efficiency of high level programming abstractions, as well as the interaction among various layers of the software stack depends on the OS-level execution contexts that are used to map language level threads. Using the UPC-pshm, hybrid and `pthreads` runtime configurations, we examine the performance tradeoffs with focus on the interaction of the PSHM-AM network with `pthreads`, the efficiency of AM processing for UPC-pshm and UPC-`pthreads`, interaction of `pthreads` and processes with the networking layer, and the performance of global locking and synchronization operations.

3.2 AM Performance

Several UPC language level constructs use Active Messages in their implementation: locks, barriers and optimized implementations of scatter/gather operations. AMs are also used in the implementation of proposed UPC language extensions such as remote atomic. operations, semaphores and remote invocations. Other modern programming language constructs such as the asynchronous activities (`async`) in X10 can also be implemented using AMs.

The AM layer implements a simplified Remote Procedure Call (RPC) paradigm, e.g. after a AM-Send operation a handler is executed within the execution context of the receiver. We measure AM latency with a microbenchmark that sends a large number of empty AMs (no data payload) between two UPC tasks. The results are presented in Table 2. The UPC-pshm executables are compiled without `pthreads` support and do not perform any locking for thread safety. The UPC-`pthreads` executables are built without PSHM support and perform locking for thread safety but use only one internal network, i.e. do not add additional overhead for polling the PSHM AM network. The UPC-hybrid executables are compiled with both PSHM and `pthreads` support.

On *Ranger* and *Hopper*, the UPC-pshm configuration (2 procs) improves AM latency by up to 3 times when compared to the network loopback approach previously employed. The customized network on BG/P uses shared memory for intra-node communication and the UPC-pshm AM latency is equal to the loopback latency.

In a pure `pthreads` configuration, all threads share the same address space and any datum is directly accessible to any thread. In this implementation, AMs amount to direct function calls within the originating thread and exhibit very low overhead. In the process based implementations, the AM handler has to be executed in the context of the "receiving" task and data movement and synchronization occurs within this operation. The latency in this case is much higher than the `pthreads` case: $\approx 2\mu s$ on *Ranger*, $\approx 1.2\mu s$ on *Hopper* and $\approx 6\mu s$ on BG/P. A pure process based implementation requires no mutual exclusion on the runtime data structures: adding the ability to run with multiple `pthreads` per process to the runtime requires adding thread safety to the runtime data structures and additional locks for mutual exclusion. As illustrated, adding thread safety to a pure process based configuration increases the AM latency by up to a factor of 1.8 on *Ranger*, 1.46 on *Hopper*, and 1.54 on BG/P.

On shared memory (single node), the capability of "in-line" processing of AM handlers for operations within the node is a strong advantage of `pthreads` that share a single address space. The pure `pthreads` based implementation with *one* process per node has the lowest latency when compared with configurations using multiple processes per node. This is an intrinsic difference in behavior that vanishes when considering multiple nodes where the AM communication is performed across the physical network. Note also that in-line processing of AMs has implications on load balancing: a `pthreads` based implementation might suffer from originator imbalance, while a process based implementation will exhibit recipient imbalance.

3.3 Communication Performance

Two network performance metrics are usually used when developing application level optimizations: 1) overhead of message injection and 2) bandwidth (or inverse bandwidth). The former is important for optimizations that employ non-blocking operations, such as overlapping communication with other independent activities, while the latter is also used in message aggregation optimizations.

To measure networking performance we use a communication intensive microbenchmark where any UPC thread communicates outside its node. We measure: (1) *blocking* communication - a thread issues only one transfer at a time and waits for its finish; and (2) *non-blocking* communication - a thread issues 1024 outstanding operations before waiting for the completion of the first one in the sequence.

Figure 2 presents the message injection cost on the *Ranger* system: two 16 AMD Opteron core nodes connected by InfiniBand. All cores within the node are active and we vary the number of `pthreads` per process, e.g. the line labeled 2T-8P illustrates the performance of a configuration where we run 8 processes each containing 2 `pthreads`. Low level networking APIs, such as InfiniBand[1] or Portals, expose to their clients (e.g. GASNet) specific data structures to describe and manage communication. Usually, two processes will use disjoint data structures, while two pthreads are multiplexed on the same data structure and require additional locking for mutual exclusion. As illustrated, the injection cost rises as the number of `pthreads` per process increases. The UPC-pshm implementation ("1T-16P" in Figure 2) exhibits the least contention and it outperforms the UPC-`pthreads` implementation ("16T-1P") by up to a factor of eight for medium sized and large messages.

On *Ranger* when a process (with multiple pthreads) spans multiple sockets we observe a severe performance degradation for small message sizes: for reference, the mes-

[1] In the IB terminology, Queue Pairs (QP) and Completion Queues(CQ).

Round Trip Latency (μs)	2 pthreads	2 procs	2 procs + thread-safety	2 procs + loopback	2 procs + thread safety + loopback
Ranger	0.085	1.989	3.598	6.082	6.675
Hopper	0.095	1.220	1.793	4.830	5.430
BG/P	0.305	6.203	9.569	6.229	9.512

Table 2. *AM latency comparison between different runtime configurations.*

sage injection overhead of UPC-`pthreads` for eight byte messages is $\approx 27\mu$s. The InfiniBand software stack is thread safe and when a process spans sockets in a NUMA system lock contention, coherency traffic and non-local lock accesses will cause this degradation. In contrast, the results on a 16 core UMA Intel Tigerton InfiniBand system do not exhibit this behavior. The UPC-pshm implementation also exhibits much lower message injection overhead than UPC-`pthreads` on both the Cray XT5 and IBM BG/P, the results are omitted for brevity. For reference, UPC-pshm has an injection overhead up to six times lower than UPC-`pthreads` on *Hopper*. On all systems, the message injection overhead decreases (e.g. by $\approx 0.4\mu s$ on *Ranger*) when increasing the number of active nodes.

The right hand side of Figure 2 presents the aggregate bandwidth between two nodes on *Ranger* when using bidirectional traffic and blocking communication. For medium and small message sizes, the UPC-`pthreads` implementation and "16T-1P" configuration achieve the lowest bandwidth due to its high message injection overhead. Note that the differences in message injection overhead presented on the left hand side of Figure 2 explain only partially the performance ordering observed in this experiment and do not fully account for the bandwidth differences between the multiple configurations.

We attribute the bandwidth differences between the different runtime configurations and implementations to the network (NICs and switches where applicable) response under load: 1) message injection rate and 2) communication topology. UPC-pshm and UPC-`pthreads` have intrinsically different message injection overheads: UPC-pshm has a lower overhead and can sustain a higher message injection rate than UPC-`pthreads`. Under load, the networks employ their own flow control and throttling mechanisms that impact the achievable bandwidth.

Figure 3 presents the aggregate bandwidth between eight InfiniBand nodes in a setting where each node communicates only with one other node. The labels contain the number of pthreads per process, the number of processes per node and the microbenchmark implementation, e.g. NB stands for the non-blocking version. When using *blocking* communication, the configurations with the lowest message injection overhead (UPC-pshm) achieve the best bandwidth for small and medium sized messages, up to five times better than the UPC-`pthreads` bandwidth, as shown in the left hand side of Figure 5. For large message sizes, all implementations achieve similar bandwidth. Figure 4 presents the aggregate bandwidth between two nodes on the Cray system, similar results are observed on IBM BG/P. For blocking communication, the differences between the implementations are less pronounced, with UPC-pshm again achieving the best bandwidth.

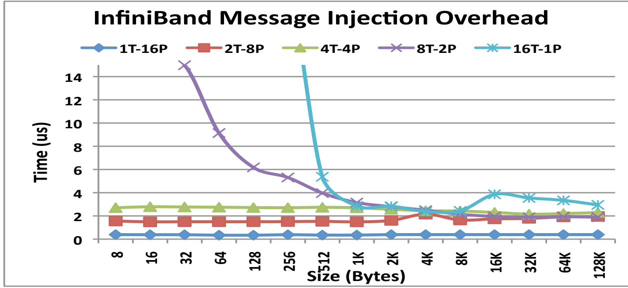
The performance degradation when applications have a high sustained message injection rate is illustrated by the behavior of the *non-blocking* communication benchmark. In this case, the implementations with higher injection overheads achieve better aggregate bandwidth: in particular, the UPC-`pthreads` implementation achieves up to five times better bandwidth than UPC-pshm, as shown in Figure 3.

Low messaging overhead allows for high injection rates and our non-blocking communication benchmark is designed to inject a high volume of traffic into the network. GASNet provides software mechanisms for throttling[2] the communication load at the source by limiting the number of allowed in-flight messages. The results in Figure 3 for non-blocking communication on Ranger correspond to the setting tuned for the UPC-`pthreads` implementation. In Figure 4 we present the impact of throttling the number of outstanding communication requests on the Cray system, e.g. "NB=32" denotes the setting where we allow only 32 outstanding messages per node. As shown, the lower[3] the number of outstanding messages allowed, the better the bandwidth (for the non-blocking communication). Of particular interest are the lines labeled "NB=32" where the UPC-`pthreads` implementation matches the bandwidth achieved when using blocking communication. For this best setting, UPC-pshm still achieves only half of the UPC-`pthreads` bandwidth for large messages, furthermore there is no setting that improves the UPC-pshm bandwidth beyond the reported numbers. On *Ranger*, throttling improves the performance (results not shown) when using non-blocking communication and for large message sizes UPC-pshm achieves the same bandwidth as UPC-`pthreads`. This experiment illustrates the benefits of additional levels of communication throttling performed transparently inside the runtime.

On all systems, changing the communication topology exercised by the microbenchmark does not change the trends reported. Figure 5 presents results for the case where one task communicates only with one other task (P2P) and the case where one task changes the destionation of each message (P2M). While the P2P benchmark setting, where a node sends multiple messages to another node, captures the behavior of hand optimized applications using bulk transfers, the P2M setting captures the behavior of applications implemented (written in shared memory style) using fine-grained communication. Changing the communication topology *does not affect* the performance ordering of the

[2] Lower level APIs also provide these mechanisms.

[3] NB=240 was the default setting before the experiments in this paper.



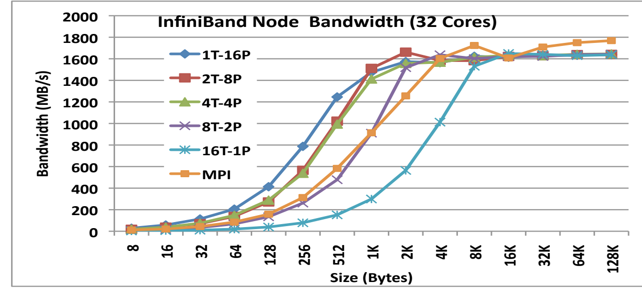


Figure 2. Message injection overhead and inter-node bandwidth on *Ranger*.

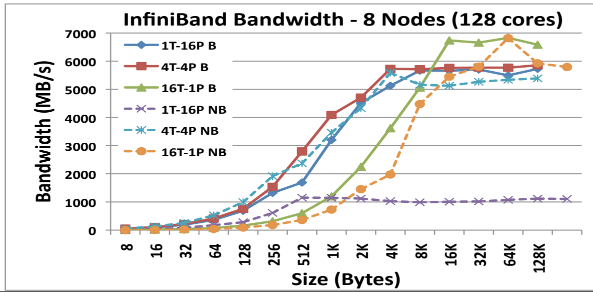


Figure 3. Aggregate bandwidth for eight nodes (128 cores) on *Ranger*. "1T-16P B" denotes 1 pthread per process with blocking (NB=nonblocking) communication .

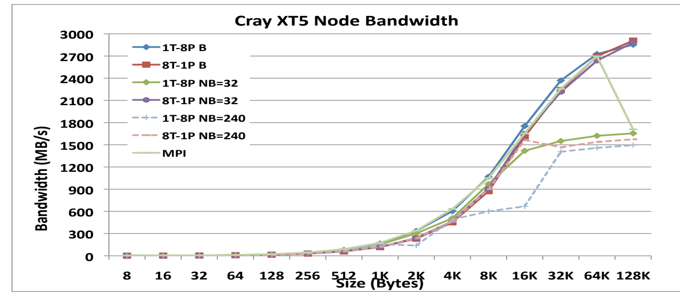
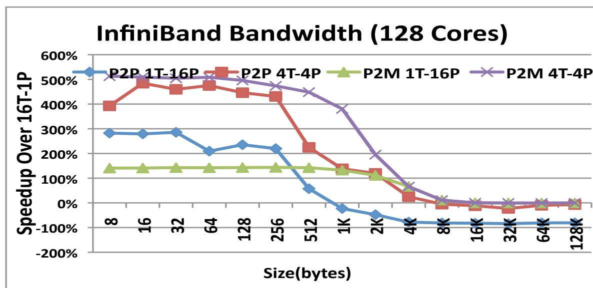


Figure 4. Node bandwidth on Cray XT5



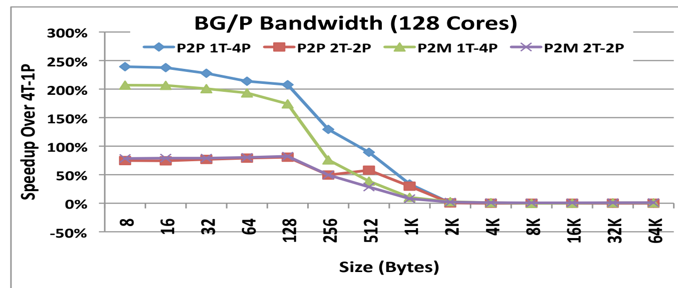


Figure 5. Bandwidth improvements of different comfigurations over UPC-pthreads(16T-1P on InfiniBand, 4T-1P on BG/P). P2P: each process has only one communication partner. P2M: each process uses a different destination process for each message.

implementations, it affects only the magnitude of the differences.

UPC-pshm and UPC-pthreads have different message injection overheads due to unavoidable implementation differences. On all systems, these differences translate into the UPC-pshm or UPC-hybrid always achieving better bandwidth than UPC-pthreads for *small to medium* sized messages. For both blocking and non-blocking communication, no single combination of pthreads and processes is capable of delivering the highest bandwidth for every transfer size and every communication type. When using non-blocking communication, each implementation requires different tuning with respect to the allowed number of in-flight messages: UPC-pthreads allows for a larger number of in-flight messages in this case. On *Ranger* a tuned UPC-pshm can match the bandwidth of UPC-pthreads for *large* messages. On the Cray, UPC-pshm provides roughly half of the bandwidth of UPC-pthreads for large messages, irrespective of additional throttling levels. Our tuning efforts also indicate that in order to further improve performance, active traffic management solutions (individual message throttling) might be required, as opposed to coarse grained node management.

4. Application Performance Evaluation

The workload contains the UPC NAS Parallel Benchmarks (UPC NPB) [21] version 2.4 augmented by optimized versions of LU, BT and SP [19], fine-grained application kernels (GUPS, MCOP and Sobel) and optimized implementations of collective operations. All NPB implementations (EP, CG, IS, MG, LU, FT, BT, SP) use bulk communication. The EP, MG, FT and LU implementations also contain critical sections and some operations implemented with fine-grained communication. All benchmarks are compiled with the Berkeley UPC 2.10.0 compiler. We conduct our experiments on the platforms described in Section 2 and report the average performance across at least five runs.

4.1 Shared Memory Performance

For the shared memory performance evaluation we use the UMA (*Tigerton*) and NUMA (*Barcelona*) 4x4 systems. Our

results indicate that for the workload considered there is little or no observable performance difference between the three configurations. On the NUMA system, the configurations where the number of pthreads per process is lower than the number of cores per socket have sometimes a very slight performance advantage for some of the benchmarks.

4.1.1 Fine-Grained Communication Benchmarks

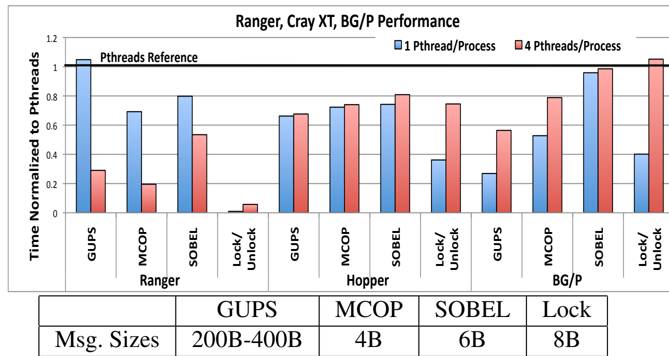
The fine-grained benchmarks, GUPS [16], MCOP [10] and Sobel [15] are designed to reflect the behavior of the communication pattern that occurs in applications during data structure initializations, dynamic load balancing, or remote event signaling. The communication pattern in GUPS also captures the behavior of stand-alone graph based applications. In addition, we present the performance of contended lock acquire and release operations.

The GUPS benchmark performs read/modify/write accesses to random locations in a large distributed array. This is a common operation in parallel hash table construction. The amount of work is static and evenly distributed among threads at execution time. The read/modify/write loops in the benchmark are designed to allow computation/communication overlap. The MCOP benchmark solves the matrix chain ordering problem, which is a combinatorial problem that captures the characteristics of a large class of parallel dynamic programming algorithms. The matrix columns are distributed across UPC threads, and communication occurs when UPC threads access elements in the same row. Sobel performs edge detection with Sobel operators (3X3 filters). The application is parallelized by partitioning the source image across threads into equal contiguous chunks of rows. Communication is performed when a UPC thread accesses bordering rows; remote accesses to the next thread rows are also performed. For the locking performance we measure the duration of a `upc_lock`/`upc_unlock` sequence of operations when all threads are operating on the same lock object.

Figure 6 shows the message sizes and it compares the timings of all process/`pthreads` combinations to the UPC-`pthreads` implementation on two nodes on all systems. Values below one indicate better performance than the UPC-`pthreads` implementation. The execution time comparison for the fine-grain communication benchmarks is consistent with the microbenchmarks behavior: e.g. due to the irregular fine-grained communication, the hybrid of processes and `pthreads` outperforms UPC-pshm and UPC-`pthreads` by up to 85% on *Ranger*. For the locking benchmark, the `pthreads` implementation is two orders of magnitude slower. On *Hopper* and BG/P, the differences between UPC-pshm and -hybrid are smaller, with UPC-pthreads always attaining the lowest performance.

4.1.2 UPC NPB Performance

Figure 7 presents the performance of the NPB implementations on *Ranger*, *Hopper* and IBM BG/P. The execution time is normalized to the UPC-`pthreads` implementation, values



	GUPS	MCOP	SOBEL	Lock
Msg. Sizes	200B-400B	4B	6B	8B

Figure 6. Performance of fine grain communication benchmarks. For each configuration, the results are normalized to 16 `pthreads` per process.

below one signify performance better than UPC-`pthreads`. All implementations perform blocking communication; Table 3 shows the message sizes.

The results show that the UPC-`pthreads` implementation is not the best option, and a process based implementation often performs better. For the majority of benchmarks, the best performance is attained by the UPC-pshm or hybrid implementations where the number of `pthreads` per process is less than or equal to the number of cores per socket. Prior to this work, the underlying assumption in the community of implementors of parallel programming models was that providing `pthreads` implementations is sufficient to exploit the shared memory within a node.

The EP benchmark contains one critical section during which a reduction is performed. The overhead of the *upc_lock()* operations, as well as the small size messages, cause a performance degradation of up to 80% when the UPC-`pthreads` implementation is used on InfiniBand. IS and FT perform all-to-all communication, and for the message sizes used in these applications, hybrid and `pthreads` based implementations achieve a slightly better bandwidth on InfiniBand (according to Figure 5). This translates into roughly 10% for IS and 5% for FT communication (bar labeled Bulk) performance improvements for UPC-hybrid and UPC-`pthreads` when compared to UPC-pshm. FT also contains a critical section which causes the large performance degradation with UPC-`pthreads` on InfiniBand.

For CG, UPC-pthreads outperforms UPC-pshm and -hybrid on InfiniBand, due to the large message sizes used in this benchmark (again, according to Figure 5). MG uses message sizes of various granularities, with most messages falling into the small to medium range. On *Ranger*, the UPC-hybrid configurations have a slight (3%-4%) performance advantage over UPC-pshm and UPC-`pthreads`. Most of the MG performance degradation on InfiniBand with UPC-`pthreads` is due to a critical section.

In LU, BT and SP tasks exchange small to medium messages, which translates in UPC-pshm and UPC-hybrid performance gains (up to 6% on InfiniBand) over UPC-`pthreads`, due to better bandwidth. They all implement a pipelined algorithm with point-to-point synchronization,

128 UPC Threads	EP	CG	IS	MG	LU	FT	BT - 256	SP - 256
Class B	8B	37KB	8B, 6.5KB - 11KB	24B - 34KB	200B - 480B, 50KB, 100KB	32KB	1KB, 6KB - 11KB, 50KB	2KB - 8KB, 50KB
Class C	8B	75KB	8B, 20KB - 40KB	24B - 130KB	160B - 800KB, 250KB	128KB	4KB, 20KB - 40KB, 120KB	5KB - 20KB, 120KB

Table 3. UPC NAS Benchmarks - Message sizes and number of barriers for 128 UPC Threads.

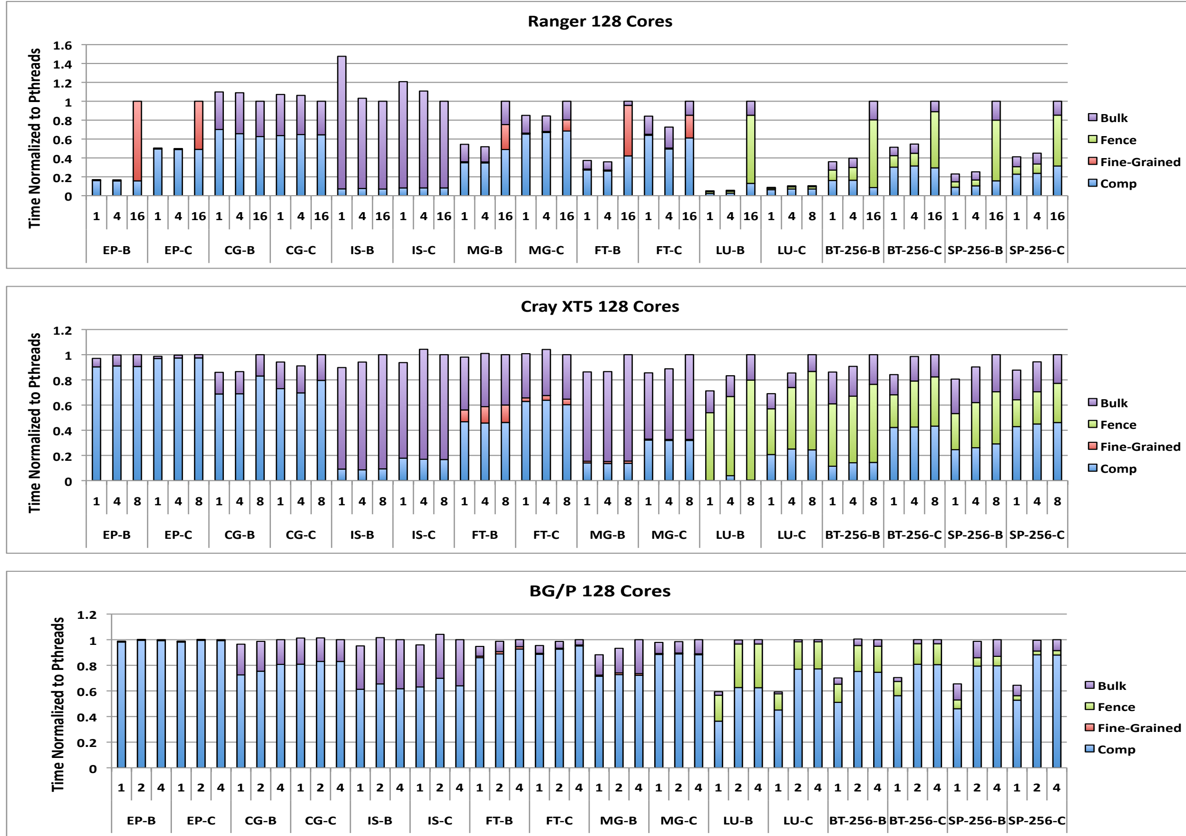


Figure 7. UPC NPB on *Ranger*, *Hopper*, and BG/P, using 128 cores. The charts present the performance of various UPC threads process/`pthreads` mapping, relative to the UPC- `pthreads` configuration.

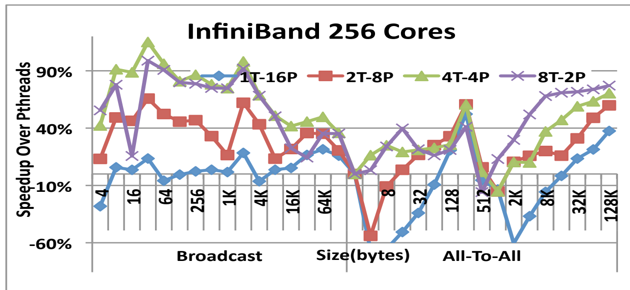
highly optimized for communication/computation overlap. The implementation uses the `upc_fence` operation to quiesce the network, which results in frequent polling operations. On InfiniBand, the UPC-`pthreads` implementation suffers the highest overhead due to lock contention in the low level networking stack. On the BG/P, a significant difference is observable in computation time across various configurations. When `pthreads` support is added to the runtime configuration, file scope variables need to be replicated and are accessed indirectly through pointers: the IBM XLC compiler generates code that runs slower in this case. This behavior is also observed by Duell [12].

On all platforms, the process based configurations perform better than a pure `pthreads` based configuration. On the Cray and IBM systems, the pure process based configuration provides best performance, while on the InfiniBand system a hybrid configuration where processes do not span sockets provides the overall best performance.

4.1.3 Collective Operations

The BUPC runtime provides optimized collective operations that use the best intra-node synchronization mechanisms, optimal communication topologies (trees) and non-blocking communication for overlap. These implementations are selected by an installation time autotuning stage. `Pthreads` within a process perform all the intra-node operations within one step, followed by inter-node (inter-process) communication: this step is performed by only one `pthread` within a process. Thus, a pure process based implementation will have the highest degree of inter-node communication parallelism, while in a `pthreads` based implementation the communication is "serialized".

Figure 8 presents the speedup of different runtime configurations over the performance of UPC-`pthreads` for a broadcast and an all-to-all operation. In the broadcast operation the communication is uni-directional with a number of active tasks determined by the tree topology. In the all-to-all operation communication is bi-directional and all tasks are active. As illustrated, on InfiniBand best performance is again attained by a hybrid configuration which provides performance improvements of up to 130%. Overall, for collective operations a hybrid configuration where a processes spans a socket (4T-4P) provides best performance. For small message sizes, the UPC-pshm implementation which exhibits the highest communication parallelism pro-



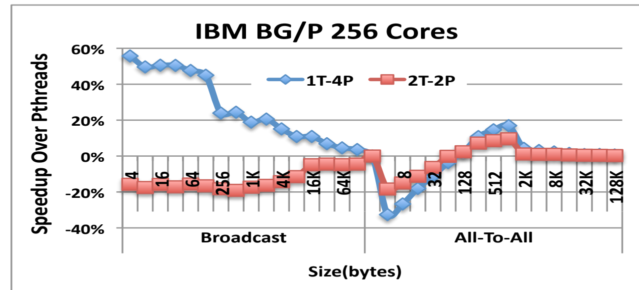


Figure 8. Broadcast and all-to-all performance comparisons. On InfiniBand, we compare against 16T-1P, on BG/P against 4T-1P.

vides slightly lower performance than the UPC-pthreads configuration. For large message sizes a process based configuration outperforms the pure pthreads based implementation.

On the IBM BG/P the pure process based configuration usually provides the best performance. On the Cray XT system, the pure process based implementation provides significantly lower performance than any combination containing pthreads. We are still investigating the cause of this behavior, which we believe to be caused by a lack of proper tuning of the collectives implementation in the process case.

5. Related Work

The initial implementation of shared memory regions between OS processes in the BUPC runtime was performed by Duell [12]. He implemented only a shared memory runtime without support for inter-node communication and we extend his work beyond a single node across multiple low level networking APIs.

A large amount of work has been performed on inter-node communication optimizations in PGAS and MPI runtimes. Several studies focus on manual and automatic communication/computation overlap using one-sided communication provided by the PGAS languages [7, 8, 17]. Our work is complementary to these studies. Bhatelé et al. [1] compare the communication performance of three supercomputer architectures: IBM BG/P, Ranger and Cray XT5. They find network contention to cause a low aggregate bandwidth when multiple MPI processes simultaneously initiate inter-node communication. This observation confirms the results presented in our study, and strengthens the argument in favor of a runtime that allows mixing of processes and pthreads. Underwood et al. [27] propose a CPU-based remote address computation, without the usage of *E-registers* on Cray T3E and report that only a few cores can saturate modern Network Interface Cards.

Several application-level studies compare the performance of processes and pthreads. Madduri et al. [20] compare pthread and MPI implementation of GTC on various shared memory platforms. While the authors observe significant speedup achieved by pthreads over the MPI implementation, they also suggest that the straightforward partitioned grid approach will not scale linearly beyond 16 threads due to the increased contention when the shared grid is updated. Antonopoulos et al. [9] explore fine-grain pthreads-based parallelization in Parallel Constrained Delaunay Mesh generation software. They also find that the locking overhead introduced by pthreads outweighs the potential benefits and propose better performing implementations using hardware locking mechanisms. Our work shows that when taking into account the interactions with third party software communication libraries pure pthreads based implementations are not capable of providing best performance due to unavoidable software interactions.

6. Conclusions

In the multi- and many-core era, implementations of programming models for scientific computing need to provide both shared and distributed memory performance. In this paper we consider the problem of mapping language (or execution model) level tasks onto OS execution abstractions: processes or pthreads. We have developed PSHM, a new process-based shared memory implementation of the Berkeley UPC runtime and explored the most efficient mapping of PGAS tasks to cores on clusters of multicore processors. We consider multiple strategies: mapping PGAS tasks to pthreads, mapping them to processes with OS-supported shared memory between them, or some hybrid of the two. We believe this is the first PGAS language implementation that allows a hybrid mapping of language threads to both pthreads and processes using shared memory for intra-node communication. By combining PSHM with pthreads, fine-grain communication benchmarks, as well as implementations of the NAS Parallel Benchmarks and collective operations, experienced speedups of more than 60%.

We discuss and isolate some of the likely factors for these performance differences, which include locking overhead in the thread version and network contention in the process version. This behavior is unavoidable and captures a *fundamental* difference when building runtimes for parallel computing: sharing of data structures requires mutual exclusion and affects network message injection rates. We present microbenchmark results that can be used to select the runtime configuration able to provide best performance and discuss the tuning required by different configurations. Our results reveal that inter-node communication, as well as communication dependent language-level constructs, are heavily influenced by the processes/pthreads choice and a hybrid approach where processes do not span sockets is likely to provide best performance.

References

[1] A. Bhatele, L. Wesolowski, E. Bohm, E. Solomonik, and L. V. Kale. Understanding Application Performance on Three Predominant Supercomputer Architectures: Intrepid, Ranger and Jaguar, using Micro-benchmarks. http://charm.cs.uiuc.edu/papers/, 2010.

[2] D. Bonachea. GASNet Specification, v1.1. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002.

[3] Berkeley UPC. Available at http:upc.lbl.gov.

[4] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS04*, pages 52–60, 2004.

[5] F. Cantonnet, Y. Yao, M. Zahran, and T. El-ghazawi. Productivity Analysis of the UPC Language. In *In IPDPS 2004 PMEO workshop*, 2004.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, and K. W. E. Brooks. Introduction to UPC and Language Specification, 1999.

[7] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick. Automatic Nonblocking Communication for Partitioned Global Address Space Programs. In *ICS '07*, pages 158–167, New York, NY, USA, 2007. ACM.

[8] W.-Y. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *PACT '05*, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Christos D. Antonopoulos and Xiaoning Ding and Andrey Chernikov and Filip Blagojevic and Dimitrios S. Nikolopoulos and Nikos Chrisochoides. Multigrain Parallel Delaunay Mesh Generation: Challenges and Opportunities for Multithreaded Architectures. In *ICS '05*, pages 367–376. ACM Press, 2005.

[10] T. Cormen, C. Leiserson, and R. Rivset. Introduction to Algorithms. *The MIT Press*, 1994.

[11] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[12] J. Duell. Pthreads or Processes: Which is Better for Implementing Global Address Space Languages? . *Masters Report, Computer Science Division, UC Berkeley*, 2007.

[13] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access. In *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.

[14] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[15] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: a NPB Experimental Study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002.

[16] B. R. Gaeke and K. Yelick. GUPS (Giga-Updates per Second) Benchmark .

[17] C. Iancu, W. Chen, and K. Yelick. Performance Portable Optimizations for Loops Containing Communication Operations. In *ICS '08*, pages 266–276, New York, NY, USA, 2008. ACM.

[18] IBM. Overview of the IBM Blue Gene/P Project. *IBM J. Res. Dev.*, 52(1/2):199–220, 2008.

[19] H. Jin, R. Hood, and P. Mehrotra. A Practical Study of UPC with the NAS Parallel Benchmarks. *The 3rd Conference on PGAS Programming Models*, 2009.

[20] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, and K. Yelicky. Memory-Efficient Optimization of Gyrokinetic Particle-to-Grid Interpolation for Multicore Processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[21] The GWU NAS Benchmarks. http://www.gwu.edu/∼upc.

[22] National Energy Research Scientific Computing Center. http://www.nersc.gov/nusers/systems/hopper.

[23] R. Nishtala, G. Almasi, and C. Cascaval. Performance Without Pain = Productivity: Data Layout and Collective Communication in UPC. In *PPoPP '08*, pages 99–110, New York, NY, USA, 2008. ACM.

[24] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[26] Texas Advanced Computing Center. http://www.tacc.utexas.edu/resources/hpcsystems/.

[27] K. D. Underwood, M. J. Levenhagen, and R. Brightwell. Evaluating NIC Hardware Requirements to Achieve High Message Rate PGAS Support on Multi-Core Processors. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[28] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and Performance Using Partitioned Global Address Space Languages. In *PASCO '07*, pages 24–32, New York, NY, USA, 2007. ACM.

[29] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *In ACM*, pages 10–11, 1998.