PGAS 2006: Presentation Abstract

Efficient Point-to-point Synchronization in UPC

Dan Bonachea, Rajesh Nishtala, Paul Hargrove, Katherine Yelick
UC Berkeley / LBNL http://upc.lbl.gov

This work proposes an extension to the UPC language designed to efficiently support point-to-point synchronizing data transfers. Microbenchmarks demonstrate how the prototype implementation significantly outperforms alternative mechanisms available in UPC. The extension is used to achieve significant speedups in a UPC-level implementation of collective operations and a Sparse Matrix Vector Multiply (SPMV) application kernel.

The traditional approach to parallel scientific computing in a distributed-memory environment utilizes two-sided message-passing interfaces such as MPI, whereby application code on both sides of a communication operation must take explicit and specific action on behalf of each data transfer in order to accomplish communication (ie each message send operation is matched by a message receive operation at the target). This model encapsulates both data transfer and implicit synchronization within each message, because by definition the completion of a message receive at the target indicates that initiator has previously produced the payload and issued the matching send operation. Messaging interfaces (including MPI) often imply additional forms of synchronization, such as point-to-point ordered delivery of messages.

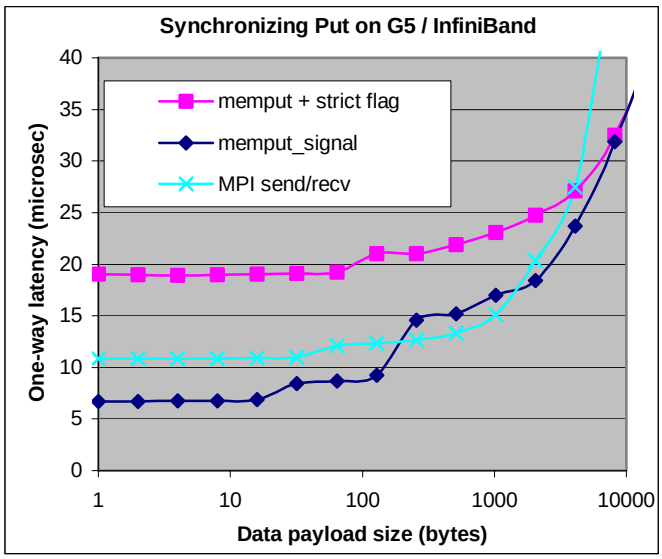
Partitioned Global Address Space (PGAS) languages such as UPC expose a one-sided communication paradigm to the application level, whereby communication is accomplished through assignments and dereferences that operate on potentially-remote data locations with no implicit synchronization semantics. PGAS languages decouple data transfer from thread synchronization; therefore necessary synchronization is usually accomplished using explicit mechanisms such as barriers or locks. This relaxation is primarily motivated by programmability concerns (the shared-memory-style semantics relieve the programmer from the error-prone tedium of two-sided message passing), and the one-sided semantics additionally allow for more aggressive optimization of communication within the parallel compiler and runtime system.

However, there is a common class of application scenarios where it is sometimes desirable to execute a data transfer and additionally allow a remote thread to synchronize on the completion of that transfer – programs with point-to-point producer/consumer data dependencies being one prominent example. The naïve way to synchronize a producer/consumer data transfer in UPC is for the producer to write shared data, followed by both threads executing a barrier, and subsequently the consumer reads the data. This method is effective and may be optimal for some classes of bulk-synchronous codes, but it's also overly heavyweight – it requires collective cooperation from all threads (which may be impractical in codes with irregular or data-dependent communication patterns) and imposes the cost of a global barrier where all that's required is a one-way point-to-point synchronization.

A more sophisticated mechanism available in UPC is for the producer to write the payload to shared memory and then write a flag variable (both variables usually having affinity to the consumer), and the consumer waits for the flag to change before consuming the payload. This mechanism (hereafter referred to as the memput + strict flag approach) is effective, but requires careful coding to guarantee correctness in the presence of aggressive reordering – for example, the producer must ensure the payload write is globally committed (using a fence or UPC strict operation) before issuing the flag write, and similarly the consumer must prevent the flag read from being optimized away or reordered with respect to payload accesses. When applied correctly in appropriate algorithmic settings, this mechanism can significantly reduce costs relative to barrier synchronization. However, the explicit use of program order on the issuing thread to construct the synchronizing data transfer generally implies an end-to-end latency of 1½ round trips on distributed-memory systems, and furthermore can inhibit overlap opportunities on the initiating thread for much of that time. There is also a get-based dual of this algorithm which uses a flag write by the producer followed by a payload get by the consumer – this variant similarly imposes an end-to-end latency of 1½ round trips and will not be considered further. Approaches explored in the context of other PGAS languages include issuing a payload write and notification message back-to-back and relying on point-to-point ordered transport to deliver them in the correct order – however such approaches are only efficiently implementable in certain contexts (and cannot be expressed with guaranteed correctness in UPC), and furthermore remain suboptimal because they still impose the CPU overhead of injecting and retiring at least two packets in the underlying transport. In order to achieve optimal performance on a number of interconnects, we desire a new semantic interface for a synchronizing data transfer that permits implementation using a single underlying transport operation that transmits both the data payload and necessary synchronization information.

This work proposes a library extension to UPC that efficiently supports point-to-point synchronizing data transfers, which we have prototyped in the Berkeley UPC compiler. The extension consists of a new semaphore shared data type `bupc_sem_t`, POSIX-like functions to create and manipulate the semaphores, and post/wait primitives that provide pairwise thread synchronization functionality. Finally, the extension includes a new synchronizing memput operation `bupc_memput_signal`, which behaves like a regular `upc_memput` but additionally signals a semaphore at the target upon completion of the payload transfer. The talk introduces this interface and explains how `bupc_memput_signal` can be used to efficiently implement producer/consumer codes in UPC. `bupc_memput_signal` provides the needed synchronizing data transfer primitive, while remaining faithful to the one-sided communication model – unlike message passing, the payload destination is specified by the initiator and therefore the operation can be retired without explicit interaction from the target thread. Furthermore, the application-level semantics are sufficiently abstract to allow a wide range of implementation strategies, providing the flexibility required to achieve the best performance on a variety of machines. On many networks the operation can be implemented using a single transport packet (for small payloads), allowing it to outperform the other available UPC-level mechanisms and match the end-to-end latency performance of MPI message passing. Figure 1 provides a one-way latency comparison of the mechanisms on two very different clusters, showing that in both cases `bupc_memput_signal` significantly outperforms the latency of signaling using memput + strict flag

and is competitive with MPI message passing latency. This trend continues across other distributed memory systems (as will be shown in the talk), and motivates the adoption of this extension.



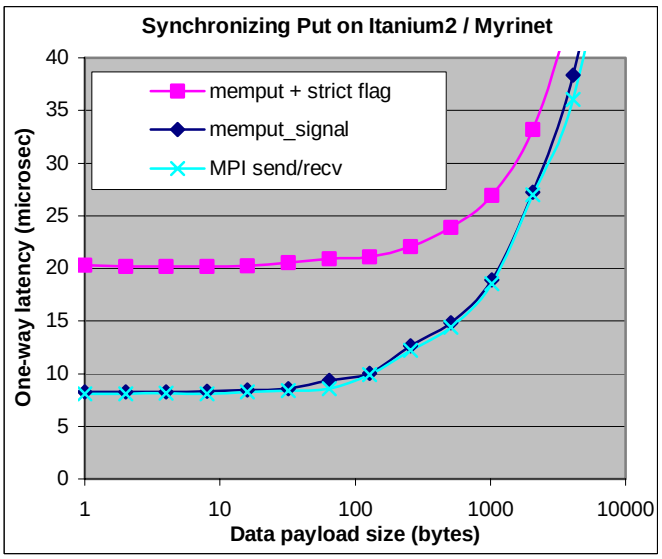


Figure 1: Microbenchmark comparison of mechanisms for synchronizing data transfer on two clusters.

As further motivation for the proposed library extension, we've written a prototype version of the UPC collectives library which utilizes a variety of communication patterns and performs synchronization using `bupc_memput_signal` or memput + strict flag. Figure 2 compares the small-payload broadcast latency of both versions for moderate thread counts – in each case the performance reported is the best over all communication patterns using that mechanism, including 1-ary to N-ary trees and a binomial tree. The figure demonstrates that across a wide variety of machines, the signaling put extension enables significant efficiency gains, allowing it to rival the performance of the MPI broadcast implementation (and significantly outperform it on one system). The talk will include similar performance results for additional collective operations and payload sizes to demonstrate the value of the `bupc_memput_signal` extension.

In order to provide application-level motivation for the extension, we've also implemented an SPMV kernel that uses synchronizing data transfers to push source vector updates to the required threads during each iteration. Figure 3 shows the performance breakdown for this kernel on an Opteron/InfiniBand cluster using either synchronization mechanism for a 2-D 9-point stencil matrix on a 1024x1024 grid with a 4x4 thread layout. This algorithm happens to be dominated by local computation, but the version using `bupc_memput_signal` achieves a 36% reduction in the synchronous communication time (time the processor is busy initiating or completing communication operations) and manages to provide an 10% reduction in total running time. The talk will present similar comparisons on other systems, and also performance results from a Conjugate Gradient kernel implemented using `bupc_memput_signal`-based reduction operations.

Future work includes refining the implementation of the library extension, incorporating its use into additional application benchmarks, and lobbying for the inclusion of this extension in an upcoming revision of the UPC specification.

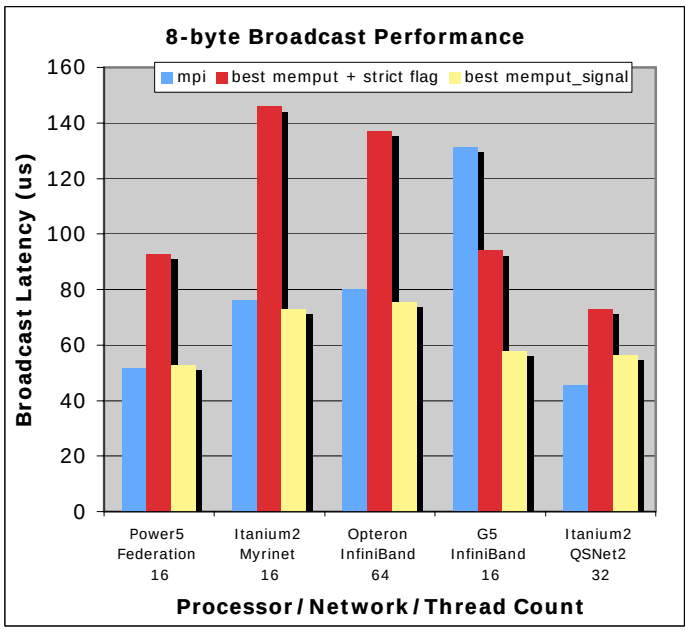


Figure 2: Performance comparison of MYSYNC broadcast using various mechanisms to implement synchronization.

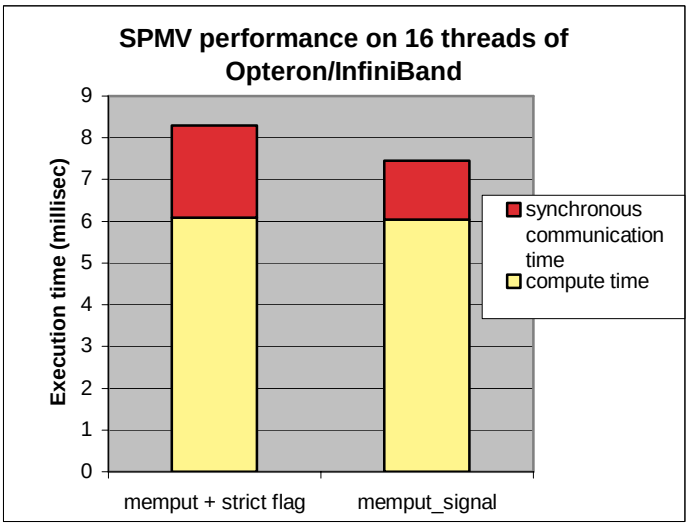


Figure 3: Performance breakdown of two UPC implementations of SPMV for a 9-point stencil matrix. The `bupc_memput_signal` version achieves a 36% speedup in communication time and 10% speedup overall.