Automatic Nonblocking Communication for Partitioned Global Address Space Programs

Wei-Yu Chen[1,2]
wychen@cs.berkeley.edu

Dan Bonachea[1,2]
bonachea@cs.berkeley.edu

Costin Iancu[2]
cciancu@lbl.gov

Katherine Yelick[1,2]
yelick@cs.berkeley.edu

[1] University of California at Berkeley
[2] Lawrence Berkeley National Laboratory

ABSTRACT

Overlapping communication with computation is an important optimization on current cluster architectures; its importance is likely to increase as the doubling of processing power far outpaces any improvements in communication latency. PGAS languages offer unique opportunities for communication overlap, because their one-sided communication model enables low overhead data transfer. Recent results have shown the value of hiding latency by manually applying language-level nonblocking data transfer routines, but this process can be both tedious and error-prone. In this paper, we present a runtime framework that automatically schedules the data transfers to achieve overlap. The optimization framework is entirely transparent to the user, and aggressively reorders and aggregates both remote puts and gets. We preserve correctness via runtime conflict checks and temporary buffers, using several techniques to lower the overhead. Experimental results on application benchmarks suggest that our framework can be very effective at hiding communication latency on clusters, improving performance over the blocking code by an average of 16% for some of the NAS Parallel Benchmarks, 48% for GUPS, and over 25% for a multi-block fluid dynamics solver. While the system is not yet as effective as aggressive manual optimization, it increases programmers' productivity by freeing them from the details of communication management.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – Optimization

General Terms

Design, Experimentation, Performance

ICS'07, June 18-20, Seattle, WA, USA.

Keywords

UPC, PGAS, nonblocking

1. INTRODUCTION

As high end computing systems continue to scale in computational power of individual compute nodes and overall node count, optimization techniques that hide communication latency through overlap of communication with computation have proven important [8, 12, 37], and recent work has demonstrated the value of overlap in problems limited by bisection bandwidth [3, 21]. Partitioned Global Address Space (PGAS) languages, such as UPC [34], Titanium [35], and Co-Array Fortran [28]), provide direct access to lightweight one-sided communication. In these languages, data transfer and synchronization are decoupled, minimizing software overheads and making communication overlap more viable. In [3], the authors showed that the performance of the NAS FT benchmark [1] could improve by 90% with aggressive communication and computation overlap in a UPC implementation. At the hardware level, communication routines are split-phase by nature; an *init* call initiates the operation, and a subsequent *sync* call blocks to await operation completion. By separating the initiation of a remote memory access as far away as possible from its completion, its latency can be hidden through the overlapping of communication and computation as well as message pipelining.

Today parallel programmers typically achieve communication overlap by manually applying nonblocking communication primitives provided by the programming model, and explicitly inserting synchronization calls for the nonblocking accesses. This manual scheduling of nonblocking communication creates additional obstacles on the already difficult task of parallel programming. Manual optimizations are cumbersome to code, and programmers typically only schedule transfers but do not apply more sophisticated optimization techniques. If one synchronizes the nonblocking accesses too early, performance suffers due to insufficient amount of overlap. On the other hand, synchronizing too late (or not at all) can result in incorrect program behavior that is often difficult to track down. In a sense, manual communication scheduling is analogous to manual memory management, a practice now generally held to be error-prone and detrimental to productivity.

Therefore, optimizations that automatically schedule communication to achieve overlap would be highly desirable.

Ideally, the compiler should statically determine the placement of the initiation and synchronization calls. In practice, however, compiler communication optimizations are often limited by the inherently static nature of the approach: 1) data transfer sizes are usually not statically known or vary dynamically for each communication operation; 2) the manner in which the application uses the transferred data might be difficult or impossible to determine due to factors such as the presence of indirect accesses (e.g., $a[b[i]]$) or third party library functions; and 3) when scheduling communication operations, the optimizer may not have a good estimate of the amount of overlap available. Thus, runtime support is often required in order for the compiler to successfully exploit communication overlap while not violating data dependencies.

In this paper, we describe a runtime framework for automatic nonblocking communication optimizations. Our system intercepts blocking communication calls and aggressively reschedules and transforms these operations. Remote puts are synchronized on-demand when a synchronization operation or other conflicting accesses are encountered. Remote gets are automatically prefetched by the runtime based on past access history. To further improve performance, the runtime automatically performs aggregation and selects the most efficient communication primitives available for special patterns such as strided accesses. We also present a number of techniques that help reduce the amount of runtime overhead associated with conflict checking as well as message initiation and synchronization.

We have implemented the system in the runtime layer of an optimizing UPC compiler. The optimization framework is transparent to the user, and through the GASNet communication system [5] supports a wide range of contemporary high-speed networks. Experimental results show that our approach removes a significant amount of communication overhead on clusters, speeding up eight benchmarks by over 19% on average. We have also tested the system on a large computational fluid dynamics (CFD) application written in UPC and achieved a 25% speedup. While our runtime optimizations are not yet as effective as aggressive manual transformations, our approach has the benefit of increasing programmer productivity by freeing them from the details of communication management.

The rest of this paper is organized as follows. In Section 2, we present background information on the language and compilation environment. Section 3 describes our runtime optimization framework that automatically overlaps blocking bulk communication, and summarizes our efforts on further tuning the performance of our system. Section 4 presents a detailed analysis of the experimental results on a supercomputing cluster. Section 5 surveys related work and Section 6 concludes the paper.

2. BACKGROUND

The approach described in this paper has been implemented in the Berkeley UPC compiler [10]. UPC is a parallel extension of the ISO C programming language aimed at supporting high performance scientific applications. The language adopts the single-program-multiple-data (SPMD) programming model. In addition to each thread's private address space, UPC provides a shared memory area to facilitate communication among threads. While a private object may only be accessed by its owner thread, all threads can read or write data in the shared address space. For more details about the language see [34].

An interesting UPC feature relevant to our optimizations in this paper is its hybrid memory consistency model. Every shared variable access in UPC is semantically categorized as either "strict" or "relaxed" by the language specification, based on a combination of type qualifiers and pragmas. The strict memory model is analogous to sequential consistency in that it requires the execution of the accesses on each thread to appear in program order from the perspective of all threads, while relaxed accesses only need to preserve local data dependencies on the initiating thread. As a consequence, communication ordering needs to be preserved only for strict accesses while relaxed memory accesses can generally be reordered. The difference between the two models is visible only in a program with a *data race*, which occurs when two threads access the same memory location with no ordering constraints between them, and at least one of the accesses is a write [27].

The Berkeley UPC compiler is divided into three main components: the UPC-to-C translator, the UPC runtime system, and the GASNet communication system. During the first phase of compilation, the Berkeley UPC compiler translates UPC programs into C code in a platform independent manner, with UPC-related parallel features converted into runtime library calls. The translated C code is next compiled using the target system's C compiler and linked to the runtime system, which performs initialization tasks such as thread generation and shared data allocation. The Berkeley UPC runtime delegates communication operations such as remote memory accesses to the GASNet communication layer, which provides a uniform interface for low-level communication primitives on all networks. GASNet provides efficient point-to-point put/get communication primitives that operate on contiguous memory regions as well as higher level operations that perform aggregation of operations targeting disjoint memory regions.

The runtime optimization framework presented in this paper is designed specifically for UPC programs. Thus, there are distinctions between read and write operations and the memory consistency model is of great assistance in the implementation. However, the same approach should be applicable to other parallel programming models that use one-sided communication and permit non-conflicting high-latency accesses to be reordered.

3. RUNTIME COMMUNICATION OPTIMIZATIONS

The basic principle of communication optimizations that exploit overlap is to issue the initiation of a communication operation as early as possible in the program schedule and the completion of the operation as late as possible. The constraints on the placement and scheduling of these operations are determined by the application data dependencies. The candidates of our optimizations are the **upc_memget** and **upc_memput** communication calls. Both calls are part of the standard UPC library and perform semantically blocking memory-to-memory operations with relaxed memory consistency; the former copies (remote) shared data into the thread's private address space, while the latter updates (remote) shared data with contents from its private buffer. The **upc_memcpy** call is also handled in the special case

where the source or destination is local, by rewriting it to the appropriate **upc_memget** or **upc_memput** call at runtime. Our optimizations operate on relaxed memory accesses, which according to the UPC memory model can be reordered as long as local data dependence is preserved.

At runtime, any program path between two synchronization events becomes an optimization region. Most UPC programs use barriers to perform synchronization, but any statements that imply a strict memory access (e.g.,**upc_lock** library call) are also considered synchronization events. We monitor the sequence of communication operations (source, destination, size) within one optimization region. For any blocking communication call within a region, we decouple the initiation of the operation from its completion. Initiations of put operations are executed in the same program order, while their completions are dynamically delayed in the execution path until a synchronization event or conflicting operation is encountered. Initiation of get operations is speculatively executed at the synchronization point directly preceding the operation in the execution path, while their completion is executed in program order. For the case of gets the operation of our runtime is equivalent to speculative prefetching.

Data consistency and dependence is preserved by a combination of dynamic conflict checks and double buffering. Communication aggregation is performed for special communication patterns such as strided accesses, and the system is carefully tuned to reduce the overhead for programs that do not benefit from overlap. This is achieved by a combination of compiler support and run-time profitability analysis. Similarly, our system reduces the overhead of prefetch initiation and metadata maintenance by using flow control heuristics and by overlapping them with communication.

3.1 Optimizing Puts

The upward mobility of the initiation of **upc_memput** operations is limited by control dependencies, so we focus on delaying its completion. The completion of a put operation can be postponed only until either a synchronization event or a memory access that conflicts with the put is encountered. Ideally such code-motion inhibiting statements should be identified at compile time so that a **sync** call of the put could be placed right before them. However, in practice static analysis alone results in overly conservative synchronization placements, due to the imprecision of the alias analysis, especially for C-based languages.

Automatic nonblocking transformation of **upc_memput** is thus best done dynamically in a demand-driven style by the runtime system, when the exact dependence information becomes available. When **upc_memput** is called, it is converted to a nonblocking call and added to a runtime-managed list of outstanding puts (a tuple of $\langle handle, remote\_addr, src\_addr, nbytes \rangle$). Completion of any outstanding put operation is necessary at the following program points:

1. Synchronization events: all outstanding put operations are completed.

2. Statements that read or write the remote destination: any outstanding put operation whose target overlaps the remote memory region involved in the operation has to be completed, in order to maintain local dependencies. At these entry points, our runtime checks for conflicts and retires the operation involved, if any.

3. Statements that modify the local source: any outstanding put whose source is modified has to be completed. Since we do not rely on static program analysis, any local write could potentially fall into this category, and checking for local dependence would pose scalability problems. To address this problem, we eliminate local conflicts by using a nonblocking put call in our communication layer that allows the local source memory to be modified once the initiation call returns. Depending on the transfer size, this may result in the source memory being copied into a temporary buffer inside the communication layer prior to sending.

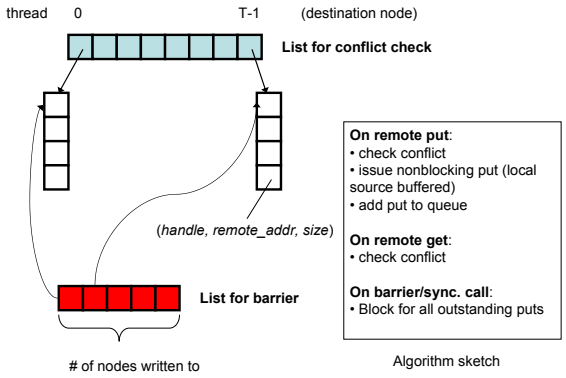

Figure 1: Runtime structure for nonblocking puts

Figure 1 depicts the runtime data structure for nonblocking put management. The outstanding puts are organized into an array of lists indexed by destination node, so that conflict checks will only be performed against puts with the same destination. The lists store and capture the FIFO order of the outstanding operations. To facilitate scalable completion at synchronization points of all outstanding operations, a separate array of pointers to the lists containing active operations is maintained. Source buffering is performed inside the communication layer and thus does not require special support.

Our approach requires minimal compiler support, as it no longer needs to guarantee code motion safety but only needs to estimate the profitability of the nonblocking transformation. At each **upc_memput** site, the translator performs a forward traversal; if it encounters a communication call, a loop, or a function call before reaching a barrier or other synchronization events, it assumes that sufficient overlap exists and marks the **upc_memput** as a candidate for nonblocking optimization.

The dynamic conflict checking required by our approach constitutes a source of runtime overhead that we try to minimize. Since operations are maintained in per-target lists, the overhead of conflict checks grows linearly with the number of outstanding requests to a specific node. However, our application experience indicates that very often programs issue communication requests to memory regions whose address varies monotonically (e.g., traversing through a remote array in a loop). This observation is also validated on a

number of numerical applications in [29]. Therefore, with each list of outstanding accesses we dynamically maintain a bounding box of the remote memory regions involved. If the target address falls outside the bounding box, clearly no conflict exists; otherwise, the runtime reverts to the slower but precise list scanning.

In programs that issue memory accesses monotonically, this technique reduces the conflict checking overhead to O(1) complexity. For programs that do not exhibit this property, a more sophisticated data structure such as balanced search trees could be used to reduce the asymptotic complexity of conflict checking[1]. However, as Section 4 indicates, even with the simple approach conflict checking does not introduce significant overhead in the benchmarks we have examined.

In order to further reduce list traversal costs, during a conflict check the runtime also queries the networking layer to test if the earliest nonblocking put is complete, and removes it from the list accordingly. The runtime also imposes a tunable limit on the maximal put list length; when the maximum is reached, the runtime synchronizes on all the puts on the list before proceeding. Besides metadata maintenance, the biggest source of runtime overhead is the buffering required inside the networking system. Section 4.1 presents some performance results.

3.2 Optimizing Gets

The downward mobility of a **upc_memget**'s synchronization is limited by any use of the local destination buffer. Therefore we would instead like to move the message initiation up, effectively prefetching the remote data. The challenge here is that the runtime needs to have knowledge about future accesses; the translator could help determine the earliest safe place to initiate a nonblocking get, but its effectiveness is severely restricted by the inherent imprecision of static analysis.

Our solution for this problem is to exploit the structured nature of most parallel code. A large class of SPMD parallel programs exhibit spatial and temporal locality in their communication pattern. While the data values being communicated are usually updated by local computation from one program phase to the next, the communication structure (size, source, and destination address of the **upc_memget** operations) often remains unchanged. For example, in a stencil algorithm like the multigrid method, the communication ghost regions are allocated once and reused in subsequent phases. Similarly, studies on the MPI NAS benchmarks suggest a significant portion of the communication calls are *dynamically analyzable*, with constant parameters at runtime [19]. Thus, we contend that for an important class of applications, one can use past history to predict future access patterns by analyzing the communication when a phase is first executed, and automatically prefetching the gets for the phase in its subsequent executions.

Figure 2 describes the design for our nonblocking get optimization. We consider a program phase to be the set of statements executed by a thread after any synchronization event and before reaching the next synchronization event. Each static source level instance of a synchronization event is assigned a unique identifier by the compiler.

[1]The stored intervals are guaranteed to be non-overlapping by virtue of the conflict-checking invariants, allowing them to be unambiguously sorted based on start address.

When a given thread reaches a synchronization statement for the first time, we create a data structure to store the prefetch candidates for the subsequent phase. Subsequent gets are recorded and associated with the event. This operation is overlapped with the execution of the original get and does not add any visible runtime overhead. On the first execution of a phase no speculative actions are performed.

For subsequent executions of a phase, prefetch calls are issued by each thread immediately upon entrance (e.g., right after a barrier), to maximize the amount of available overlap. To avoid local dependency violations or spurious updates to user data on a mispredict, the prefetched data are stored into runtime-allocated temporary buffers. When a **upc_memget** call is encountered, if it matches one of the prefetches, the runtime synchronizes the outstanding transfer and copies the desired data from the prefetch buffer into its destination. Otherwise, the runtime adds this previously unseen get to the prefetch list and issues the get as usual.

At the end of a phase, the runtime synchronizes and deletes any unused prefetches, to avoid performing useless communication in the future[2]. Any put operation within a phase requires conflict checks against the outstanding gets, to ensure that the application does not read stale values that violate data dependencies. This means that no conflict checking is required when issuing the prefetches.

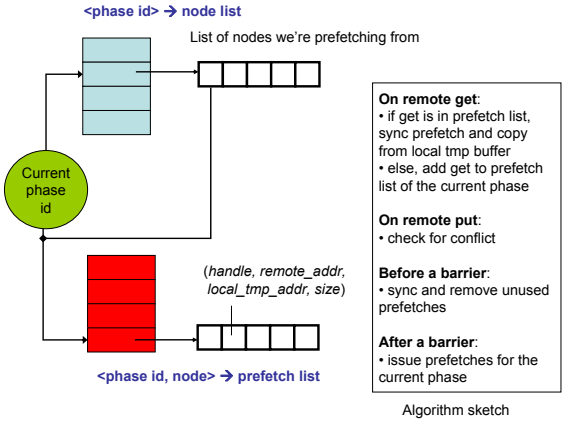

Figure 2: Runtime structure for get prefetching

The runtime structures for nonblocking gets consist of two hash tables. One maps the current phase to a list of nodes that the thread has been prefetching from, while the other maps $\langle phase\_id, node\_id \rangle$ to a list of prefetches. Similar to the case for puts, the compiler algorithm complexity is greatly reduced, since the use of prefetching temporary buffers and runtime conflict checking eliminates the need for static data dependence analysis. The get optimization notably assumes that remote addresses appearing in a **upc_memget** operation remain valid for the lifetime of the program, such that subsequent speculative get operations can be safely issued without danger of causing a runtime fault (for example, if the relevant remote object has been freed). In Berkeley UPC this is already guaranteed because

[2]Mispredicted gets must be retired before their target buffers can be safely reused for subsequent prefetches, to prevent a race between two outstanding gets to the same buffer.

the shared memory allocator never unmaps memory pages.

To avoid prefetching a get whose copy overhead outweighs its available overlap, each get's profitability is estimated before its addition to the prefetch list. Profitability is determined dynamically by comparing the amount of overlap (time of **upc_memget** - time of phase start) to the memory copy overhead. The estimation overhead can be completely overlapped by the cost of the remote get.

3.3 Automatic Communication Aggregation

Thus far in our framework, the gets and puts are issued individually to hide their communication latencies through overlapping. Many SPMD programs, however, have an alternating phase structure with remote accesses grouped into a phase separate from local computation. The accesses in a communication phase are generally non-contiguous - however, combining puts and gets between the same pair of node into larger transfers is often profitable, as it amortizes the high per-message overheads of cluster network hardware over larger data payloads, thereby achieving a higher effective transfer bandwidth. Manual packing and unpacking of the non-contiguous accesses can be tedious and error-prone, however, and departs from the one-sided communication model since they typically require the cooperation of the remote thread. An optimization that automatically detects and aggregates the communication bursts would therefore be very useful.

We augment the framework to perform communication aggregation by targeting the non-contiguous remote access methods called the **VIS** (vector/indexed/strided) functions in the GASNet communication layer [6]. The VIS calls accept a list of non-contiguous put/get as arguments, and the communication algorithm is selected at runtime based on network characteristics and transfer parameters. The VIS calls perform message aggregation using GASNet Active Messages, packing non-contiguous data at the source into large packets and unpacking it at the destination. In Section 4.3 we present some micro-benchmark results on the performance of VIS calls.

When the translator detects at compile time a potential burst of gets or puts (e.g., a **upc_memget** inside a loop), it inserts special **begin_aggregate** and **end_aggregate** runtime calls to mark the *aggregation region*. When the runtime encounters a remote access inside the region, instead of issuing the access immediately (for puts) or adding it to the prefetch list (for gets), it stores it into an aggregation queue, which is again organized based on the remote node. Upon exiting the aggregation region, the runtime issues the communication using a single VIS call per remote node, letting the network decide the best way to combine and schedule the accesses. Special patterns such as strided accesses and accesses with identical size are recognized and supported using the more efficient VIS calls with reduced metadata overhead. Conflict checks proceed as usual inside an aggregation region, and a conflict terminates the aggregation region prematurely by switching to the default behavior described earlier.

In the current design, no buffering is done for the accesses in an aggregation region, and the translator must guarantee that local memory used by the gets and puts is not modified by other code falling within the aggregation region. As most communication phases (the candidates for our aggregation regions) are short and contain no computation code, so far our translator has been successful in proving that the **upc_memget** and **upc_memput** operations inside aggregation regions can safely be reordered without violating data dependences. We are planning to add a runtime check function that the translator can issue when it cannot verify if a local access may be in conflict with the **upc_memget** or **upc_memput** calls.

4. EXPERIMENTAL RESULTS

The last section presented the basic design for our automatic nonblocking communication framework. In this section, we provide a performance analysis for our framework. Our optimizations are primarily designed for cluster architectures where remote communication latency is high relative to the processor speed. For space constraints we only present experimental results from the cluster listed in Table 1, though our findings are applicable to any cluster systems where communication overlap is available (i.e., message overhead is smaller than the network latency). The get and put time in the table refer to the cost of a blocking 8-byte remote access.

4.1 Buffering Overhead

The put initiation cost depends on the message startup overhead of the network, as well as the overhead imposed by the communication layer's buffering of the local source. Figure 3 compares the cost of issuing a nonblocking put without source buffering and with source buffering; the semantic difference between the two is that the former poses the additional requirement that the local source memory cannot be safely modified until the nonblocking put completes. The latency of a blocking put is also included for comparison. For very small puts (up to 72 bytes on this GASNet network), the transfer is automatically performed using PIO and thus incurs no buffering overhead. Larger puts incur a cost for copying the source to a bounce-buffer, and beyond about 1KB this memory copy begins to affect the nonblocking put initiation time, and grows roughly linearly with the transfer size. Even with buffering, however, the cost of issuing a nonblocking put is still significantly less than the latency of a blocking put. It should also be noted since the buffering is done at the discretion of the communication layer, a nonblocking buffered put should never perform worse than a blocking put.

Two main sources of overhead exist in the case of get operations. If the get has been prefetched, there may be a cost for the synchronization of the matching prefetch, which should never be higher than that of the original blocking get. Additionally, since the prefetch stores the remote data into a temporary buffer, the runtime needs to pay a copy overhead. Figure 4 measures the copy overhead by comparing the execution time of a local memcpy to that of a blocking get. While the copy overhead is negligible for small gets, it could equal about 30% of the communication latency for large transfers; this motivates our profitability analysis described in Section 3.2.

An additional potential performance penalty for buffering is that it may increase memory pressure by evicting live data from the caches. We have not observed this effect in our experiments, however. Without our optimization, the data fetched by a get sits in memory after a remote direct memory access (RDMA), and the application would need to pull it into cache on demand on first use. Since it is likely

Processor	Network	Software	Get	Put
2.2GHz Opteron	Infiniband 4X	Linux 2.6.5, pathcc 2.4	11.6us	8.4us

Table 1: Machine summary

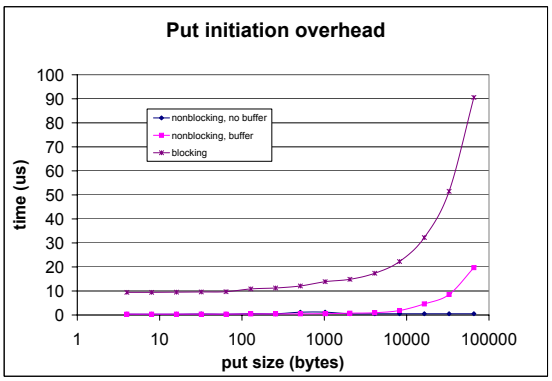

Figure 3: Put initiation overhead

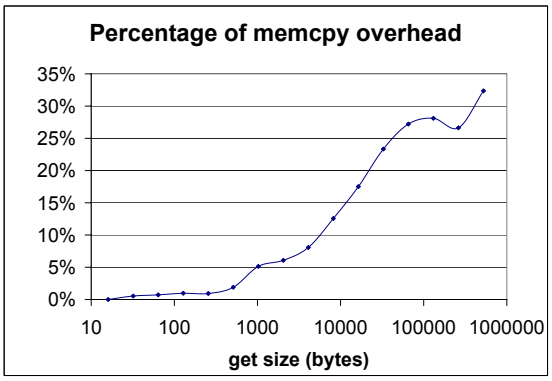

Figure 4: Memory copy overhead for prefetched gets.

that the application will immediately use the data after a get, our copy operation has the effect of streaming the data from the prefetch buffer into cache. Finally, the penalties imposed by get prefetch "misses" in an RDMA system are almost entirely constrained to NIC resources and memory bus bandwidth. Mistakenly prefetched data should never occupy space in cache.

4.2 Communication-Related Overhead of Speculative Prefetch

The costs of prefetch synchronization before a barrier is affected by the number of mispredicted prefetches in the phase, as the useful ones would have been synchronized earlier when their matching **upc_memget** call was encountered. To improve prefetching accuracy, the runtime removes from the list prefetches that are never used in the phase. Furthermore, the prefetch clearing time can be overlapped with the barrier latency by using *split-phase barriers*. In a split-phase barrier, a thread first notifies other threads that it has reached the barrier, then waits until all threads have executed the notification call. By inserting prefetch synchronization code between the notify and the wait call, we can effectively overlap its overhead with the barrier latency. Micro-benchmark results on our target platform suggest that even in the absence of load imbalance, a barrier still takes $55us$ for 16 nodes and up to $87us$ for 64 nodes, well above the round trip latency for small- and medium-sized gets. We therefore expect the overhead of list clearing to be completely hidden by barrier latency for most applications[3].

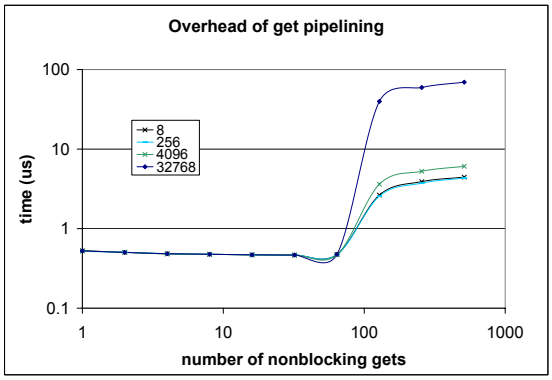

Figure 5: Prefetch initiation overhead, for each individual get.

The prefetch initiation overhead is primarily determined by the network's ability to issue consecutive nonblocking gets, specifically the message gap parameter in the LogP model [2]. Since the initiation occurs after a barrier and before user code executes, it falls in the critical path and we cannot easily hide this synchronous overhead. Figure 5 measures the initiation overhead of a sequence of nonblocking gets on our target platform. Four different message sizes are measured: 8 bytes, 256 bytes, 4KB, and 32KB. While the per-get overhead of initiating 64 nonblocking gets is the same as that of one get, at 128 gets performance begins to suffer dramatically for all message sizes, due to the (tunable) network queue depth being exceeded. Once the network queue depth is exceeded, subsequent initiation operations stall until the head of the queue is retired, and the stall time is primarily determined by the message size and the network bandwidth.

We implement a simple flow-control mechanism to prevent a flood of messages during prefetch initiation and avoid these

[3]This optimization does not apply to nonblocking puts, because UPC's memory model requires a thread to globally complete its put operations before issuing the notify operation. The reordering is legal on the prefetches, however, because the fetched data is never used.

costs of exceeding the network queue depth. Instead of issuing all prefetches immediately after the barrier, the runtime issues them in 64-element chunks. When a **upc_memget** is encountered, our system checks (without blocking) if the previously issued prefetches to the same destination node have finished; their completion indicates that the network is likely not busy, and the system issues the next chunk of prefetches. Since the array of prefetches for each node is kept in FIFO order, our system will also first prefetch for **upc_memget** operations that occur earlier in program execution. This technique thus would work well for iterative code with locality in communication schedule, and these applications are exactly the kind we are targeting for the get prefetching optimizations.

4.3 Effectiveness of Communication Aggregation

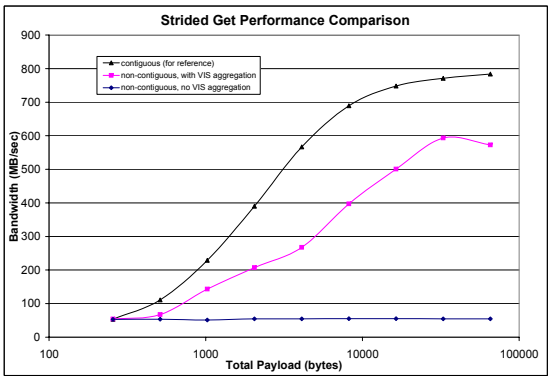

Figure 6: Strided get performance microbenchmark

Figure 6 compares the effective bandwidth of performing a logically non-contiguous get operation using a single aggregated VIS call versus a flood of individual (non-aggregated) get operations that achieve the same data movement. The microbenchmark specifically measures the bandwidth for a 1-d strided get operation with 256-byte elements and a stride of 850 bytes at both the source and destination, while varying the number of elements to vary the total transfer size – this setup was chosen because it closely matches the most common access pattern for the BT benchmark examined in section 4.4.

The figure demonstrates that performing the non-contiguous get using message aggregation provides a huge bandwidth advantage over the non-aggregated approach; the latter has consistently poor bandwidth due to the high message count and relatively heavy per-message overheads. The non-aggregation curve is flat because in the experiment all NIC-level messages are 256 bytes independent of the total payload for the higher-level strided operation (shown on the x-axis). Thus, we observe a constant bandwidth equal to the flood bandwidth for 256-byte messages. For comparison purposes, the figure also includes the raw bandwidth for fully contiguous transfers of the given total payload size (where no aggregation is necessary), to represent the theoretical maximal bandwidth for a get of the given payload size. The non-contiguous, aggregated transfer pays CPU and memory system overheads for gathering and scattering the payload from the non-contiguous source and destination locations to contiguous transfer buffers at the network layer, which explains the degradation relative to the raw contiguous transfer performance.

4.4 Benchmarks

Table 2 lists the benchmarks used in our performance evaluation. BT, IS, MG, and SP are derived from the MPI NAS benchmarks, and their implementations are described in [14, 18]. The FT benchmark is designed to aggressively overlap communication with computation [3], and FT-pencils is a variant of the benchmark that issues smaller messages for better overlap. The implementation of CG is described in [4], and gups is a version of the HPCS RandomAccess benchmark that uses bulk communication. Cfd is an application that solves the time dependent Euler equations for computational fluid flow in a rectangular computational domain, with the high level data structures and algorithms implemented in UPC. Class B input size is used for the NAS benchmarks, while the gups benchmark executes eight million updates. Communication ratio refers to the percentage of execution time spent inside **upc_memget** and **upc_memput** calls, and thus represents a upper bound on the performance improvements from our optimizations. Collectively the benchmarks cover a wide range of communication patterns, from a large number of small messages (BT, SP) to a small number of large transfers (FT, IS).

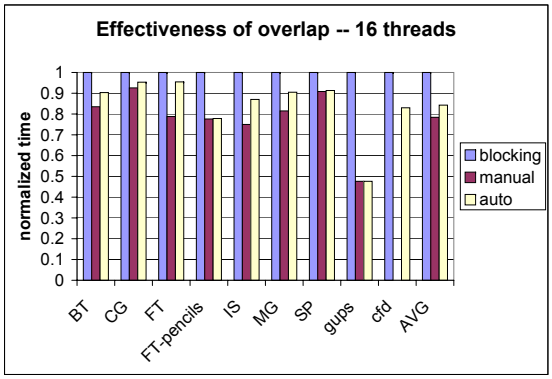

Figure 7: Optimization speedup for 16 threads.

Figure 7 presents the optimization speedup of our optimization framework. Three configurations are compared: a *blocking* version that uses fully blocking communication, a *manual* version where the communication calls are manually converted to nonblocking in a way that maximizes overlap, and finally an *auto* version with our optimizations. Since the cfd program contains more than 10,000 lines of UPC code, we have not manually converted the communication calls to be nonblocking; this underscores the importance of optimizations that could automatically generate nonblocking communication. Sixteen threads are used in the experiments, with one thread per node. As expected, nonblocking communication is an effective method for hiding communi-

Name	Get or Put	Comm. Ratio(%)	Get/Put count (thousands)	Avg/Max Size (KB)
BT	both	27.1	2100	0.15 / 1
CG	get	8.2	34	33 / 38
FT	put	22.3	5	131 / 131
FT-pencils	put	22.3	164	4 / 4
IS	get	50.9	0.352	239 / 512
MG	put	30.3	19	2.6 / 135
SP	put	9.8	636	1.7 / 216
Gups	put	61.9	8	0.5 / 1
Cfd	get	23	123	0.1 / 0.25

Table 2: Benchmark characteristics. The results were collected on 16-thread runs, using data from thread zero.

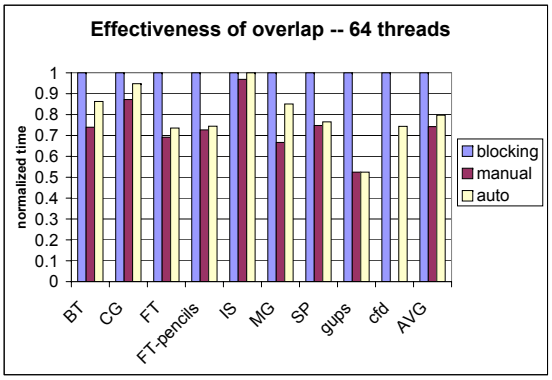

Figure 8: Optimization speedup for 64 threads.

cation latencies, with the manual version speeding up the benchmarks by 22% on average; for some benchmarks (e.g., FT and SP) the communication latency is almost entirely eliminated. Our automatic optimization framework is somewhat less effective and removes 16% of the communication overhead overall, and is faster than the blocking version for all benchmarks. BT and cfd benefit the most from communication aggregation due to their large number of small-sized gets.

The largest performance discrepancy between the manual and the auto version occurs on the FT benchmark, whose large transfer size induces a high local buffering overhead that negates most of the advantages from overlapping. Disabling network buffering for puts (this happens to be correct for this benchmark) brings the auto version's performance to within 3% of that of manual blocking, but has no effects on the other benchmarks. Thus, while our framework is effective for small to medium transfer sizes, a zero-copy consistency checking algorithm could further benefit applications with large-sized puts.

As mentioned in previous sections, conflict checks represent pure overhead introduced by our system. To quantify its potential impact on performance, we compute the ratio of conflict check time versus the speedup time achieved by our system (i.e., time for *blocking* - time for *auto*); this ratio approximates the additional improvement expected from our system if such checks could be disabled. The ratio is under one percent for all of the benchmarks except BT, suggesting that conflict checking is virtually free. For get-based benchmarks (CG,IS) and gups, which pack communication to the same node into a single transfer, conflict checks occur rarely and thus incur no overhead. The other put-based benchmarks issue the accesses with monotonically increasing remote addresses, and our bounding box optimization is effective at reducing their conflict check overhead.

To test the scalability of our framework, we run the benchmarks with the same input size using 64 threads in Figure 8. Manual tuning improves performance by 25%, while our system achieves a 19% speedup overall. Nonblocking communication is effective on all benchmarks with the exception of IS, which fails to benefit from either automatic or manual optimizations. The communication part of the IS benchmark is implemented as a collective all-to-all exchange, with the total number of messages in the program increasing quadratically as more threads are added. In the absence of independent computation to be overlapped, the resulting network contention reduces the effectiveness of pipelined communication. Compared to 16-processor runs, our automatic optimization becomes much more effective on the FT benchmark due to a reduction in message size; because the input size is fixed, quadrupling the thread count decreases transfer size by the same ratio, and therefore results in a much smaller buffering overhead.

5. RELATED WORK

Overlapping communication and computation has long been recognized as an important technique for hiding communication latencies [8, 12, 20, 25]. Consequently, there have been ongoing efforts in promoting nonblocking communication as a standard feature in the PGAS language community. Nonblocking array copy operations have recently been introduced into Titanium's standard library [17], and a similar proposal is currently awaiting approval by the UPC consortium. The Rice Co-Array Fortran compiler allows programmers to explicitly create nonblocking communication via the use of a nonblocking region [13], and Split-C [16] also supports explicit split-phase operations. Manual communication scheduling improves performance, but also exposes the complexity of nonblocking communication to the programmer.

Chen et al. [11] described an SSA-based optimization framework that uses static analysis to generate split-phase remote accesses. While their algorithm is effective for fine-grained irregular accesses, static scheduling of bulk communication

is more difficult because it is not always possible to statically determine the memory locations that a communication or computation statement will access, especially for a language like C. Iancu et al. [21] attempted to overcome the limitation of compiler analysis by utilizing virtual memory support. When a nonblocking communication is issued, the local memory pages involved in the communication are marked as having no access through the `mprotect` system call, so that synchronization will happen on-demand when the computation statements attempt to access those pages and receive a memory protection error. The scheme maximizes the amount of overlap, but may not be portable; the POSIX standard [30] requires that the protected memory be obtained from the `mmap` call, but the local memory involved in nonblocking communication generally either lives on stack or comes from malloc, which may not use mmap to obtain new memory.

Su and Yelick [33] developed an array prefetching algorithm for irregular array accesses in Titanium using inspector-executor techniques. Their optimization supports loops with indirect accesses ($A[B[i]]$) and not bulk communication in general. Kamil et al. [24] developed compiler analysis techniques to reduce the number of memory fences required for enforcing sequential consistency, and used the analysis to automatically convert blocking array copies in Titanium into nonblocking operations. Their optimization significantly improves the performance of two matrix vector multiply benchmarks.

Intelligent run-time systems have received renewed interest in recent years and show very promising potential in terms of performance, scalability and programmer productivity. The approach most closely related to ours is the Charm++ [9] runtime. Charm++ provides for latency hiding through an abstract execution model based on processor virtualization and message-driven execution. Charm++ decomposes the computation and achieves overlap by rescheduling threads that block for communication. In a sense, our approaches are converses – Charm++ schedules computations while our optimizations aggregate and schedule communication. Sorensen and Baden [23] present a data-driven programming model and run time library that manages communication pipelining and scheduling through task graph, actor-like execution.

Software caching of remote memory has been studied extensively in the context of distributed shared memory (DSM) systems [7, 22, 26, 32], and is available in a number of UPC compilers [15, 31]. While remote reference caching can be very effective for shared memory style code which is oblivious to data locality [36], it may not help programs for which the user has manually replaced fine-grained accesses with bulk communication. Most well-tuned PGAS applications use bulk communication to amortize the high remote access latencies on clusters, and our optimizations are specifically designed to further improve the performance for these kind of programs.

6. CONCLUSION

Effective use of communication networks is critical to the scalability of parallel applications. Partitioned Global Address Space languages have proven effective at utilizing modern networks because their one-sided communication is a good match to underlying network hardware. These languages also provide the means to leverage communication overlap for latency hiding, however the use of split-phase communication operations has primarily been manually applied by programmers. We have presented a runtime algorithm for automatically optimizing PGAS programs by transparently converting blocking remote data transfers into nonblocking ones in a way that maximizes the overlap of communication with computation, while still maintaining the memory consistency guarantees of the language. Data dependence is preserved via runtime conflict checks and temporary buffers, with a number of techniques applied to lower the metadata maintenance overhead. The system also recognizes special access patterns and automatically aggregates communication to further improve performance.

We have implemented the system in an optimizing UPC compiler. Our experimental results show that the automatic optimizations can be quite effective at hiding communication costs for a wide variety of UPC programs. Average speedups from our techniques are 16% on 16 processors and 19% on 64, relative to the blocking communication specified in the source program. While the automatically optimized versions do not yet match the performance of the versions with manually-optimized nonblocking communication (which use application-specific knowledge to resolve dependencies), the system is transparent to users and thus frees them from the details of communication management. Additionally, since the system does not depend on the SPMD model or other specific parallelism constructs in UPC, the principles are applicable to other parallel programming models that use one-sided communication and permit non-conflicting accesses to be reordered.

7. REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An Evaluation of Current High-Performance Networks. In *the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-sided Communication and Overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[4] K. Berlin, J. Huan, M. Jacob, et al. Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

[5] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[6] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet. Technical Report LBNL-56495, Lawrence Berkeley National Lab, October 2004.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.

[8] S. Chakrabarti, M. Gupta, and J. Choi. Global Communication Analysis and Optimization. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, 1996.

[9] CHARM++ project web page. Available at http://charm.cs.uiuc.edu.

[10] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003.

[11] W. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-grained UPC Applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[12] S.-E. Choi and L. Snyder. Quantifying the Effects of Communication Optimizations. *ICPP*, 00:218, 1997.

[13] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. A Multi-platform Co-Array Fortran Compiler. In *the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT 2004)*, 2004.

[14] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, et al. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 36–47, 2005.

[15] Compaq UPC version 2.0 for Tru64 UNIX. http://h30097.www3.hp.com/upc/.

[16] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing (SC1993)*, 1993.

[17] K. Datta, D. Bonachea, and K. Yelick. Titanium Performance and Potential: an NPB Experimental Study. In *18th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, 2005.

[18] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *Supercomputing2002 (SC2002)*, November 2002.

[19] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In *14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, November 2002.

[20] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, July 1996.

[21] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the overlap. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[22] L. Iftode and J. P. Singh. Shared Virtual Memory: Progress and Challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[23] Jacob Sorensen and Scott Baden. A Data Driven Model for Tolerating Communication Delays. In *Proceedings of the 12th SIAM Conference on Parallel Processing for Scientific Computing* , 2006.

[24] A. Kamil, J. Su, and K. Yelick. Making Sequential Consistency Practical in Titanium. In *Supercomputing 2005 (SC'05)*, November 2005.

[25] M. T. Kandemir, A. N. Choudhary, P. Banerjee, J. Ramanujam, and N. Shenoy. Minimizing Data and Synchronization Costs in One-Way Communication. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1232–1251, 2000.

[26] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, 1994.

[27] R. Netzer and B. Miller. What are race conditions? some issues and formalization. *ACM Letters on Programmming Languages and Systems*, I(1), March 1992.

[28] R. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[29] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and Precise Array Access Analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[30] POSIX standard. http://www.opengroup.org/onlinepubs/009695399/.

[31] J. Savant and S. Seidel. MuPC: A Run Time System for Unified Parallel C. Technical Report CS-TR-02-03, Department of Computer Science, Michigan Techincal University, September 2002.

[32] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, 1996.

[33] J. Su and K. Yelick. Array Prefetching for Irregular Array Accesses in Titanium. In *Sixth Annual Workshop on Java for Parallel and Distributed Computing*, 2004.

[34] UPC language specifications, v1.2. Technical Report LBNL-59208, Berkeley National Lab, 2005.

[35] 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. *Concurrency: Practice and Experience*, 10:825–836, 1998.

[36] Z. Zhang and S. Seidel. Benchmark Measurements of Current UPC Platforms. In *4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, April 2005.

[37] Y. Zhu and L. J. Hendren. Communication Optimizations for Parallel C Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, pages 199–211, 1998.