AM++: A Generalized Active Message Framework

Jeremiah J. Willcock
Indiana University
150 S. Woodlawn Ave
Bloomington, IN 47401, USA
jewillco@cs.indiana.edu

Torsten Hoefler
University of Illinois at Urbana-Champaign
1205 W. Clark St.
Urbana, IL 61801, USA
htor@illinois.edu

Nicholas G. Edmonds
Indiana University
150 S. Woodlawn Ave
Bloomington, IN 47401, USA
ngedmond@cs.indiana.edu

Andrew Lumsdaine
Indiana University
150 S. Woodlawn Ave
Bloomington, IN 47401, USA
lums@cs.indiana.edu

ABSTRACT

Active messages have proven to be an effective approach for certain communication problems in high performance computing. Many MPI implementations, as well as runtimes for Partitioned Global Address Space languages, use active messages in their low-level transport layers. However, most active message frameworks have low-level programming interfaces that require significant programming effort to use directly in applications and that also prevent optimization opportunities. In this paper we present AM++, a new user-level library for active messages based on generic programming techniques. Our library allows message handlers to be run in an explicit loop that can be optimized and vectorized by the compiler and that can also be executed in parallel on multicore architectures. Runtime optimizations, such as message combining and filtering, are also provided by the library, removing the need to implement that functionality at the application level. Evaluation of AM++ with distributed-memory graph algorithms shows the usability benefits provided by these library features, as well as their performance advantages.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Performance

Keywords

Active Messages, Parallel Graph Algorithms, Parallel Programming Interfaces

1. INTRODUCTION

High-Performance Computing (HPC) has long been synonymous with floating-point-intensive scientific computation. However, there has recently been an emergence of large-scale applications that do not match the traditional HPC profile. The human genome project, for example, was an early and highly visible example of a nontraditional HPC problem that required levels of performance and storage typically only available in high-end supercomputing systems. Nontraditional HPC problems continue to grow in importance and in their computational requirements in such diverse areas as bioinformatics, data mining, search, knowledge discovery, network analysis, etc.

Developing high-performance software for nontraditional HPC problems is complicated by the fact that HPC hardware, software, and systems have been designed and optimized for traditional HPC applications. In addition to not being floating-point-intensive, nontraditional HPC problems differ from traditional HPC problems by being unstructured, dynamic, and data-driven. These characteristics of nontraditional HPC problems present particular challenges for parallelization. The standard parallelization approach for scientific applications follows the SPMD model which partitions the problem data among a number of processes and then uses a bulk-synchronous computation and communication pattern (typically with message passing) to effect the overall computation. However, this approach to parallelization is not well-suited to nontraditional HPC problems [15].

An alternative mechanism for parallelizing nontraditional HPC applications is to use an active messaging (AM) model, where all computation is driven by message handlers. Active messages were originally developed as part of the Split-C project [29] but are widely used today in a number of different areas. Systems such as Unified Parallel C (UPC), Co-Array Fortran, a number of MPI implementations, and object-based programming systems all rely on active messaging for their low-level communications. Unfortunately, using active messages directly in an application is difficult because the interfaces to existing active message systems are too low-level.

To address these issues, we have developed AM++, a new library for active message programming, intended for use by nontraditional HPC applications. AM++ is targeted at a "middle ground" between low-level active message libraries such as GASNet [2] and object-based runtime systems such as Charm++ [13]. Particular features of AM++ include the following:

- Allowing message handlers to themselves send arbitrary messages, simplifying the expression of some applications.
- A multi-paradigm implementation, including object-oriented and generic programming, to balance runtime flexibility with high performance.

PACT'10, September 11–15, 2010, Vienna, Austria.

- Type safety provided for messages.
- A modular design enabling configurability by the user without substantial application modification.
- A design enabling compiler optimizations such as inlining and vectorization of message handler loops, and multi-threaded message handlers.
- Flexible message coalescing and redundant message elimination.
- Performance competitive with existing AM systems. Graph algorithms written using AM++ have higher performance than those using the previous, ad hoc AM framework in the Parallel Boost Graph Library [9].

2. RELATED WORK

Previous libraries supporting active message or remote procedure call semantics can be broadly grouped into two categories: low-level and high-level. Low-level systems are meant for other libraries and run-time systems to build upon, or to be used as a target for compilers; performance is the most important goal, with flexibility less important. These libraries do not provide type safety for messages and often have arbitrary, frequently system-specific, limits on the number and types of parameters to an active message, actions that can be performed in active message handlers, etc. As these libraries assume a sophisticated user, features such as message coalescing are not always provided. Examples of these libraries include IBM's Deep Computing Messaging Framework (DCMF) [14], IBM's Low-level Application Programming Interface (LAPI) [21], Myrinet Express (MX) [8], and GASNet [2]. GASNet's manual states that it is not intended for direct use by applications [2, §1.2], while DCMF and LAPI are intended for such use. GASNet is the most portable interface and can use the native InfiniBand interface on our system; thus, we used it in our comparisons as an example of a fast low-level library.

On the other hand, libraries in the remote procedure call (RPC) tradition are designed for usability, often at the expense of performance. These libraries tend to be more flexible, and more likely to support heterogeneous systems, interface discovery (including sophisticated interface definition languages), and security; run-time dispatch is used heavily, but type checking is usually provided. Some RPC systems expose individual functions for remote access, while others expose objects and their methods. Examples of traditional RPC systems include Java RMI [30], CORBA [20], and ONC RPC [26].

Some authors have implemented systems based on RPC principles that are intended for high-performance computing applications. These systems trade off some of the flexibility of fully general RPC in exchange for higher performance. They also include techniques such as message coalescing to reduce message counts, and rely on asynchronous method calls to hide latency. These systems include Charm++ [13, 22] and ARMI [25]. The ParalleX system is also based on active message/RPC principles, with sophisticated methods for locating possibly-migrated remote objects and non-message-based synchronization methods [12]. Optimistic Active Messages [31] are a technique that allows more flexibility in handlers even within a low-level active messaging system; it runs active message handlers in an interrupt context (for example) until they do forbidden operations (such as operations that could block), at which point they are delayed for execution in a separate thread.

Our work differs from these systems in that it does not use objects as the target of remote accesses; we instead use functions registered as handlers. Our messages target nodes, not single objects. Because we allow C++ function objects, methods could be used as AM++ handlers as well by adding an explicit parameter representing the object being addressed, but that is not the intended use case. We provide the flexibility of arbitrary actions (including active message sends) in message handlers like in RPC systems, but use a fully asynchronous model: a message is not guaranteed to be sent until an explicit synchronization operation. We additionally avoid run-time dispatch for our higher-level interfaces using the techniques of generic and generative programming; the compiler is able to resolve that an entire buffer of coalesced messages is all intended for the same handler, and optimize accordingly. We therefore are in the middle, between the low-level AM implementations with their focus on performance and the higher-level systems with their focus on ease of use; our goal is a mix of both.

The Message Passing Interface (MPI) standard defines *one-sided* operations [18, §11] which are in some sense similar to active messages with predefined handlers. Those operations allow users to access remote memory (put and get) and perform calculations on remote memory (accumulate). The latter is probably closest to the concept of active messages; however, MPI specifies a fixed set of possible operations and does not currently support user-defined accumulations.

Partitioned Global Address Space (PGAS) languages, such as Unified Parallel C (UPC) [27] and Co-Array Fortran [19], are comparable to AM++ in that they are both trying to provide user-level interfaces to AM and one-sided messaging. The Berkeley UPC compiler's runtime system uses GASNet for communication [1]. However, these languages primarily use active messages and one-sided operations for data movement, not for the implementation of user algorithms. AM++ exposes the ability to use active messages for algorithm implementation.

3. LIBRARY DESIGN

AM++ is designed to be especially flexible, without sacrificing performance. The essential features include a mix of runtime and compile-time configurability in order to trade off runtime flexibility for performance overhead. Another feature is extensive type safety: message data being sent, as well as the handlers receiving messages, use well-defined types to enable both better compile-time error checking and the potential for compiler optimizations. Part of this potential is that message handlers for individual messages within a coalesced block can be inlined into the loop over the whole block (with further optimizations possible), a feature that is difficult to achieve with indirect calls through function pointers registered at runtime. Message coalescing itself can be customized as well: for example, when a particular kind of message is idempotent, duplicate messages can be eliminated by the library—saving the user the effort (and code complexity) of removing them in their application. AM++ also allows handlers to invoke arbitrary actions, including sending active messages (to an arbitrary depth), as well as allocating and freeing memory. One consequence of nested messages is that detecting when all messages have been handled is more difficult (the termination detection problem); we allow the user to specify the algorithm to use and the level of nesting required to trade off flexibility (in nesting depth) for performance (simpler algorithms for limited depths).

AM++ is built with a layered design, as shown in Figure 1. To blend performance and flexibility, the lower-level communication layers use an object-oriented design for greater configurability at runtime (for example, an individual subroutine can add message handlers for its own use, with those handlers removed when the subroutine ends). The assumption at the lowest level is that data blocks are typically large (due to the coalescing of messages) and so the overheads of such a design will be amortized across a large number of messages. The upper layers of AM++, however, han-

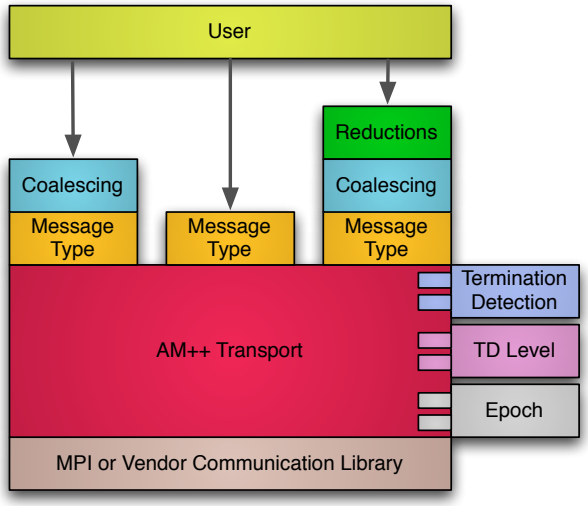


Figure 1: Design of AM++.

dle the individual messages; an application will send many of those small messages, making performance much more important. These layers use C++ generic programming techniques to allow compile-time configuration without a loss of performance. The compiler can then use static knowledge of the configuration of these levels to optimize the code that handles individual messages more effectively.

An AM++ program incorporates the following steps, also shown in Figure 2:

1. Creating a transport to provide low-level communication interfaces (Section 3.1).
2. Creating a set of message types—type-safe handlers and send operations—within that transport.
3. Optionally constructing coalescing layers to group messages for higher throughput (Section 3.2).
4. Beginning a message epoch (Section 3.3).
5. Sending active messages within the epoch (triggering remote operations in those messages' handlers).
6. Ending the message epoch, ensuring all handlers have completed.

3.1 Low-Level Transport Layer

At the lowest level, active messages are sent and received using *transports*. These are abstractions that represent sets of handlers, with the ability to add and remove handlers dynamically. The transport manages termination detection and epochs, as described in Section 3.3. Transports also provide various utility functions, such as accessing the number of nodes and the current node's rank.

Several transports can be active at one time, but they will not interact; waiting for a message in one does not test for messages received by the others, which may lead to a deadlock. In this way, transports are somewhat similar to MPI communicators, but more independent. MPI communicators interact in the sense that they all belong to the same MPI implementation, and thus share message progression; AM++ transports are completely independent and so progress on one does not necessarily imply progress on others. Section 3.4 contains more details of progress semantics in AM++.

AM++'s current low-level transport is built atop MPI for portability. However, nothing in the model is tied to MPI or its semantics. Transports could also be built to use lower-level AM libraries such as GASNet [2] or direct network interface libraries such as InfiniBand verbs (OFED) or Myrinet Express (MX). The MPI transport is thread-safe assuming thread-safety in the underlying communication libraries. In our MPI-based implementation, we use MPI datatypes to support heterogeneous clusters without needing explicit object serialization; serialization could be added later to handle more complicated user-defined data types and communication libraries that do not support heterogeneity.

In order to send and handle active messages, individual *message types* must be created from the transport. A transport, given the type of data being sent and the type of the message handler, can create a complete message type object. That message type object then stores the handler and provides a type-safe way to send and handle messages. The interface between the message type object and the underlying transport is internal; the user only sees the methods provided by the message type and its calls to user-defined handlers. When a message is sent, its type is checked to ensure it is valid for that message type object. Messages are also type-checked and necessary conversions are performed before their handlers are called so that handlers do not need to perform type conversion internally. Message type objects are required to be created collectively (i.e., by all processes in the transport at the same time) across nodes, and so no type checking is done between different nodes. Any allocation of message ID numbers or similar objects is done internally by the message type objects; users do not need to do anything to avoid message ID conflicts, even across separate libraries using the same transport. An example of setting up an AM++ transport, creating a message type (built by a coalescing layer), and sending a message is shown in Figure 3.

At the transport level, message coalescing and other combining techniques are not applied, and so messages consist of buffers (arrays) of small objects. Sending messages is fully asynchronous to enable overlap of communication and computation, and so the application (or upper messaging layer) passes in a reference-counted pointer that represents ownership of the data buffer. When the buffer can be reused, the reference count of the pointer is decremented, possibly calling a user-defined deallocation function contained in the pointer. This function is effectively a callback indicating when it is safe to reuse the buffer. Buffers for received messages are automatically allocated and deallocated by the message type object using a memory pool; the user is thus required to declare the maximum message size for each message type.

3.2 Message Set Optimization

The modularity of AM++ enables several layers to be built atop the basic message type objects. These layers form generalizations of message coalescing, and thus are important to increase message transfer rates. Because the layers work on single, small messages that are sent very frequently, compile-time configuration (expressed using C++ templates) is used to define the compositions of components. Increasing the amount of compile-time knowledge of the application's messaging configuration provides more information to the compiler, allowing a greater level of optimization and thus lower messaging overheads. Here, we describe three layers that can be built atop the basic message types: message coalescing, vectorization and optimization of message handlers, and duplicate message removal or combining.

3.2.1 Message Coalescing

Message coalescing is a standard technique for increasing the rate at which small messages can be sent over a network, at the cost of increased latency. In AM++, a message coalescing layer can be applied atop a message type; the basic message type does

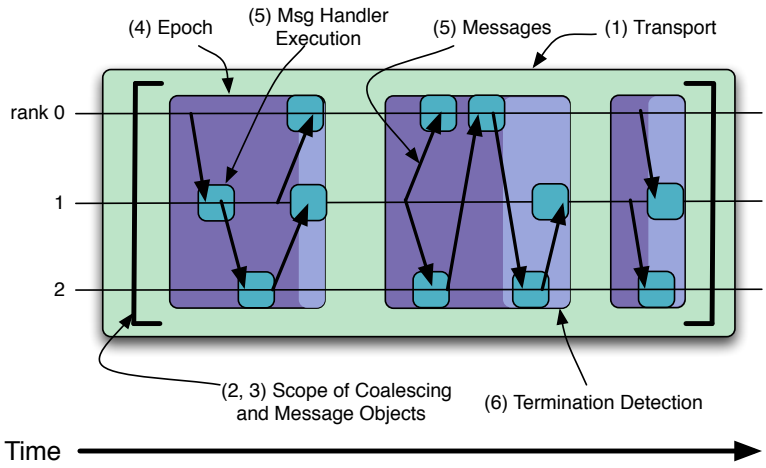


Figure 2: The overall timeline of a program using AM++. Parenthesized numbers refer to the preceding list of steps.

```
// Create MPI transport
mpi_transport trans(MPI_COMM_WORLD);

// Build coalescing layer (the underlying message type object is created automatically)
basic_coalesced_message_type<my_message_data, my_handler, mpi_transport>
  msg_type(trans, /* Coalesced message size */ 256);

// Set message handler
msg_type.set_handler(my_handler());

// Declare that handler will not send messages (0 is nested message level)
scoped_termination_detection_level_request<mpi_transport> td_req(trans, 0);

{ // The block represents a single epoch
  scoped_epoch<mpi_transport> epoch(trans);
  if (trans.rank() == 0) msg_type.send(my_message_data(/* data */ 1.5), /* destination */ 2);
}
```

Figure 3: Example usage of AM++.

not need to know whether (or what kind of) coalescing is applied. Coalescing can be configured separately for each individual message type. Each message coalescing layer encodes the static types of the messages it sends and of the handler that it calls, leading to the optimizations described in Section 3.2.2.

In our implementation, a buffer of messages is kept for each message type that uses coalescing and each possible destination. Messages are appended to this buffer, and the entire buffer is sent (using the underlying message type) when the buffer becomes full; explicit flush operations are also supported. A timeout is not currently provided, but could be added in the future; a user could also create his/her own coalescing layer supporting a timeout. When a buffer of messages arrives at a particular node, the lower-level handler (provided by the coalescing layer) runs the user's handler on each message in the buffer. The one complication in coalescing is its interaction with threads. We have two implementations for thread-safety of message sends: one uses an explicit lock, while another uses the CPU's atomic operations to avoid explicit locking in the common case (the buffer has been allocated and is not full) and spins in other cases. Message receiving is thread-safe without any difficulty: a single thread is used to call handlers for all of the messages in one buffer, while other threads can simultaneously call handlers for messages in other buffers.

3.2.2 Message Handler Optimizations

The static knowledge of message types and handlers available to coalescing implementations enables several potential optimizations that are not available to other, more dynamic, AM frameworks such as GASNet. The simplest is that the user handler is called for each message in a buffer; this loop's body is simply a call to a known function that can be inlined. After inlining, other loop optimizations can be applied, such as loop unrolling, vectorization, and software pipelining. Hardware acceleration, such as in a GPU, could potentially be used for especially simple handlers, as could execution in the network interface hardware (NIC).

One particular way in which the handlers for messages in a single buffer can be optimized is through fine-grained parallelization. For example, an OpenMP directive could be applied to mark the handler loop as parallel, or a more explicit form of parallelism could be used. One benefit of AM++'s modularity is that parallelism granularity is message-type-specific and can be varied with minimal application modifications. The fine-grained model is different from handling different buffers in different threads, and its finer granularity of parallelism requires low-overhead thread activation and deactivation. This granularity is applied within the AM system, without requiring modifications to application code (other than thread-safety of handlers).

3.2.3 Redundant Message Combining/Elimination

One convenient feature that AM++ provides in library form is the removal or combining of redundant messages on the source node. For example, a breadth-first search only processes each vertex once, and so later attempts to process it will be ignored (i.e., messages are idempotent). If bandwidth was a performance bottleneck, removing these messages would benefit performance. Similarly, a transposed sparse matrix-vector multiplication accumulates values into an output array; updates to the same array element can be combined on the source node rather than the destination. These capabilities would normally need to be provided directly by the application, and thus entangled with the rest of its code.

In AM++, redundant message combining and elimination can be provided as a wrapper atop a coalescing layer: a new layer filters the messages by keeping a cache, and passes new messages to the next layer down. For duplicate message removal, a cache can be used with a very simple filtering scheme. We have implemented a direct-mapped cache that allows some duplication but with very fast queries and updates, even in a multi-threaded environment; and a cache that uses a hash table to be more accurate but also slower. For messages that are ⟨key, value⟩ pairs, we also provide a reduction layer that combines messages with the same key using a binary operator; this layer is equivalent to local combiners in the MapReduce system [5]. Again, a simple cache is used for performance.

3.3 Epoch Model and Termination Detection

In AM++, periods in which messages can be sent and received are referred to as *epochs*. Our notion of an epoch is similar to an active target access epoch for MPI 2.2's one-sided operations [18, §11.4] in that all AM operations must occur during the epoch, while administrative operations such as modifying handlers and message types must occur outside the epoch. In particular, all nodes must enter and exit the epoch collectively, making our model similar to MPI's active target mode. Epochs naturally structure applications in a manner similar to the Bulk Synchronous Parallel (BSP) model [28], except that AM-based applications are likely to do much or all of their computation within the communication regions. AM++ does not provide a guarantee that active messages will be received in the middle of an epoch, but does guarantee that the handlers for all messages sent within a given epoch will have completed by the end of that epoch. This relaxed consistency model allows system-specific optimizations.

One feature that distinguishes AM++ from other AM libraries such as GASNet is that it gives much more flexibility to message handlers. In particular, handlers can themselves send active messages to arbitrary destinations, and are not limited to only sending replies. A major benefit of this capability is that it greatly simplifies some uses of AM; for example, a graph exploration (such as that shown in Section 4.2) can be directly implemented using chained message handlers in AM++, while a separate queue, including appropriate locking, is required when direct sending is forbidden.

More sophisticated message handlers, on the other hand, cannot be implemented using system interrupt handlers; see Section 3.4 for more information on this tradeoff. Traditional AM systems such as GASNet require each handler to send at most one reply; this restriction can be used to avoid deadlocks [29]. Also, end-of-epoch synchronization becomes more difficult with unrestricted handlers: in the event that a message handler itself sends messages, all of these *nested messages* will also need to handled by the end of the epoch. Distributed termination detection algorithms [7, 16] are required to determine this property reliably in a distributed system.

The literature contains several algorithms for termination detection with arbitrary chains of nested messages; our current implementation uses the four-counter algorithm described in [16, §4] with a non-blocking global reduction (all-reduce) operation from libNBC [10] to accumulate the counts; this approach is similar to the tree-based algorithm by Sinha, Kalé, and Ramkumar [23]. General termination detection algorithms allowing arbitrary nested messages can be expensive, however; in the worst case, the number of messages sent by the program must be doubled [3, §5.4]. AM++ allows the user to specify the depth of nested messages that will be used; finite depths allow simpler algorithms with lower message complexity to be used, such as generalizations of the algorithms by Hoefler et al. [11]. Users can add and remove requests for particular depths, with the largest requested depth used; these requests can be scoped to a particular region of code (see Section 3.5).

Because many termination detection algorithms are based on message or channel counting, a user-defined integer value can be summed across all nodes and then broadcast globally without extra messages in many cases. AM++ supports this operation as an optional part of ending an epoch; termination detection algorithms that do not include that feature automatically would need to do an extra collective operation if a summation is requested in a particular epoch. This feature is useful for graph algorithms; many algorithms consist of a phase of active messages followed by a reduction operation (for example, to determine if a distributed queue is empty), and thus benefit from this capability in the AM library.

3.4 Progress and Message Management

Active messages can be received and processed in several places in the application. For example, GASNet can run message handlers inside a signal handler, leading to restrictions on what operations can be done in a handler (except when a special region is entered) [2]. For flexibility, and to avoid the use of operating-system-specific features, we run all handlers in normal user mode (i.e., not in a signal context). As we currently build on top of MPI, our own code also runs as normal user code. Another option would be to use a background thread to process messages, as GASNet allows. We do not mandate the use of threads—and thus thread safety—in user applications, and so we do not create a background thread automatically. The user could spawn a background thread that simply polls the AM engine in order to handle message progress. If there is no progress thread, the user must periodically call into AM++ to ensure that messages are sent and received. In the worst case, ending an epoch will provide that assurance. MPI and GASNet also use this model of progress, and so is likely to be familiar to users.

Our approach to threads is to allow but not mandate them. Actions such as registering and unregistering message types are not thread-safe; it is assumed that the user will perform them from only a single thread. A single epoch can be begun by several threads on the same node; the epoch must then be ended by the same number of threads. Actions such as message sends and handler calls are thread-safe for all of AM++'s standard coalescing layers and duplicate message removers. A compile-time flag can be used to disable locking when only one thread is in use. Our model is thus similar to MPI's ***MPI_THREAD_MULTIPLE*** or GASNet's ***PAR*** mode. For our MPI-based transport, we normally assume ***MPI_THREAD_MULTIPLE*** in the underlying MPI implementation, with a compile-time flag to switch to ***MPI_THREAD_SERIALIZED*** for MPI implementations that do not support multiple threads in the library simultaneously.

3.5 Administrative Objects

"Resource Acquisition is Initialization" (RAII) [24] techniques are used throughout AM++ to simplify applications. For example, handlers are registered by message type objects, and automatically

unregistered when the message type is deleted. Requests for particular levels of nested messages (termination detection depth) are also managed using this technique. Epochs are scoped in a similar manner, except that a user-defined summation at the end of an epoch requires it to be managed manually. RAII is used internally for the management of several other types of registration and request objects. RAII prevents many types of resource leaks by giving the compiler responsibility for ensuring exception-safe deallocation of objects.

4. EVALUATION

We present performance results on Odin, a 128-node InfiniBand cluster (Single Data Rate). Each node is equipped with two 2 GHz Dual Core Opteron 270 CPUs and 4 GiB RAM. We ran our experiments with Open MPI 1.4.1, OFED 1.3.1, and GASNet 1.14.0. We used the latency/bandwidth benchmark `testam` included in GASNet (recording section L, as it is closest to the messaging model AM++ uses) and a simple ping-pong scheme for AM++. The compiler used was `g++` 4.4.0. We have observed that multiple MPI processes per node, each with a smaller portion of the graph data, is less efficient than a single MPI process with more graph data. This effect is likely due to the increased communication caused by partitioning the graph into more pieces and the overhead of communicating through MPI to perform work on graph data that is present in a node's local memory. Thus, all of our tests used a single MPI process per node.

4.1 Microbenchmark: Latency and Bandwidth

Figure 4 shows a comparison of AM++ and GASNet with respect to latency and bandwidth. We remark that the AM++ implementation runs on top of MPI while the best GASNet implementation uses InfiniBand (OFED) directly. We see a minimal difference in latency between GASNet over MPI and AM++ ($< 0.6\mu s$) and a slightly larger difference ($< 3.1\mu s$) between the optimized GASNet over InfiniBand and AM++ versions. Our design also allows for an implementation on top of OFED, however, we have limited our focus to MPI for increased portability. As with latency, GASNet over InfiniBand performs slightly better than AM++ with respect to bandwidth. We note that GASNet's MPI conduit does not support messages larger than 65 kB.

Our modular design allows for runtime-configurable termination detection and uses dynamic memory management (but with heavy use of memory pools) for send and receive buffers. These capabilities impose a small overhead on all messages, primarily due to virtual function calls.

4.2 Kernel Benchmark: Graph Exploration

In order to demonstrate the benefits of sending arbitrary active messages from within a handler context, we now discuss a simple graph exploration kernel. The graph exploration algorithm is similar to breadth-first search in that each vertex of a graph reachable from a given source is explored exactly once, with a color map used to ensure this property. Unlike BFS, however, graph exploration does not place any constraints on the vertex order, and thus avoids internal synchronization.

We implemented two versions of the kernel. In the first, *queue-based* implementation, the active message handler adds remote vertices that have been discovered to the local queue for processing. The second, *nested-message* implementation's active message handler performs the exploration step and sends all remote messages immediately. The nested-message implementation thus has significantly lower overhead (fewer push/pop operations) and is much more agile because it generates new messages without the delay of a queue. However, it requires that handlers be able to send messages to arbitrary processes. While we needed to use the queue-based approach for GASNet, we implemented both versions for AM++.

Figure 5 shows the results of the benchmark. For each run, a random graph was generated using the Erdős-Renyi model, and then a Hamiltonian cycle was inserted to ensure that the graph was strongly connected. Erdős-Renyi graphs are characterized by a high surface-to-volume ratio and normally-distributed vertex degrees. We seeded the random number generator statically to ensure reproducible results.

One cause for the large performance difference between AM++ and GASNet queue-based implementations is that AM++ includes message coalescing, while GASNet does not. AM++ is designed for user applications that benefit from coalescing, even when it adds latency, while GASNet is designed for optimal latency and assumes features such as coalescing will be built atop it. Although we could have implemented coalescing on top of GASNet, it would have complicated the application; the AM++ version sends its messages through a layer that applies coalescing automatically.

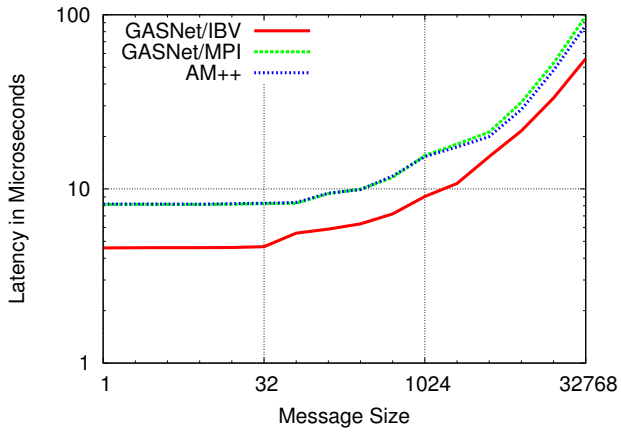
Another difference, as explained above, is that in the nested-message implementation, the handler explores all local vertices reachable from the received vertex and sends messages for any remote neighbors of those vertices. A local stack is used to avoid overflowing the system stack, with a separate stack for each handler call. The queue-based implementations, on the other hand, use a global stack to communicate between the message handler and the code's main loop. The handler pushes vertices onto the queue because it cannot send messages directly. The main loop then processes the elements on the queue, sending messages for those vertices' remote neighbors.

The third difference is termination detection. In AM++, termination detection is included in the library, while the user must implement it on top of GASNet. For the queue-based implementations, we chose a simple scheme based on message counting: each sent message increases a local counter and the handler generates a reply message that decrements the counter; termination is detected if all counters reach zero. This scheme adds additional overhead in comparison to the optimized termination detection in AM++, used in the nested-message implementation.

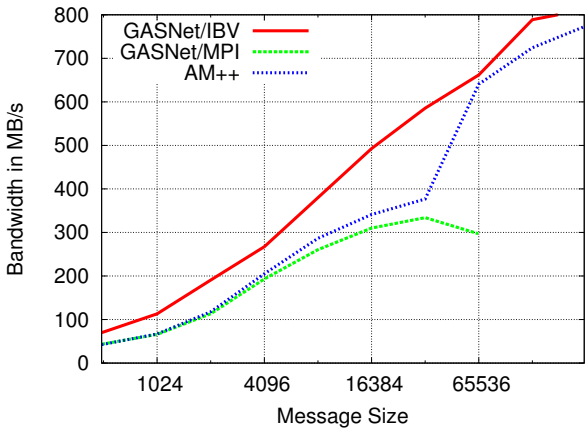
Thus, we note that the huge performance differences between AM++ and GASNet on this benchmark stem from the different goals of the two libraries. While GASNet is intended as a low-level interface for parallel runtimes and thus tuned for the highest messaging performance, AM++ is more user-friendly and supports direct implementation of user algorithms and thus enables higher performance with less implementation effort (the nested-messaging AM++ implementation has 40% fewer lines of code than the queue-based GASNet version). In this sense, our results in this section emphasize the different purposes of the two libraries rather than fundamental performance differences (as one could, with significant implementation effort, reproduce most of the features of AM++ on top of GASNet).

4.3 Application Benchmarks

Distributed-memory graph algorithms are an excellent application use case for active messages as they can be highly asynchronous and extremely latency-sensitive. The Parallel Boost Graph Library (Parallel BGL) [9] is one of the most successful publicly available distributed-memory graph libraries. The Parallel

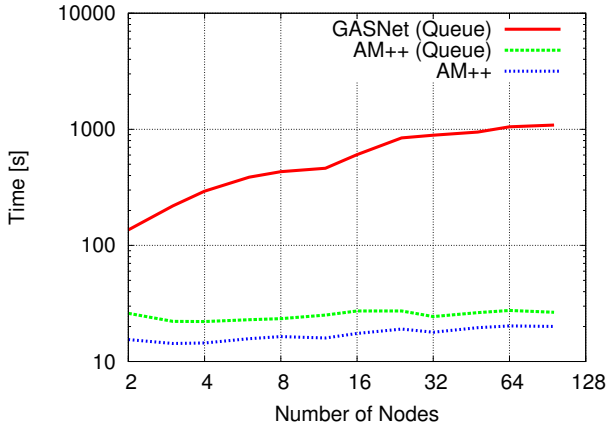


(a) Latency.

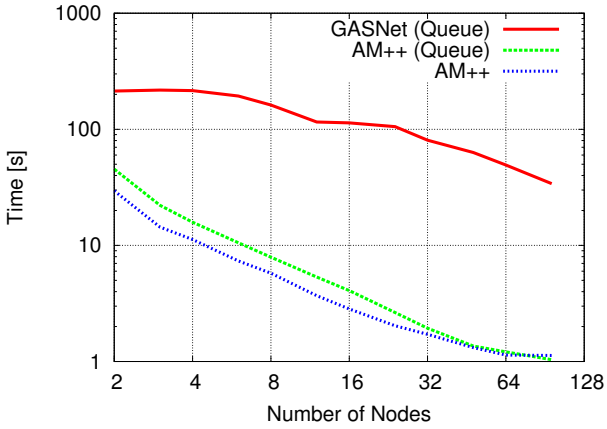


(b) Bandwidth.

Figure 4: GASNet over MPI or InfiniBand vs. AM++ over MPI with a ping-pong benchmark.



(a) Weak Scaling (5M vertices per node).



(b) Strong Scaling (15M vertices total).

Figure 5: Comparison of GASNet and AM++ with a simple graph exploration benchmark.

BGL includes the concept of a Process Group which abstracts the notion of a set of communicating process. The *MPI Process Group* is one implementation of the Process Group concept; it performs message coalescing, early send/receive, and utilizes asynchronous MPI point-to-point operations and traditional collectives for communication.

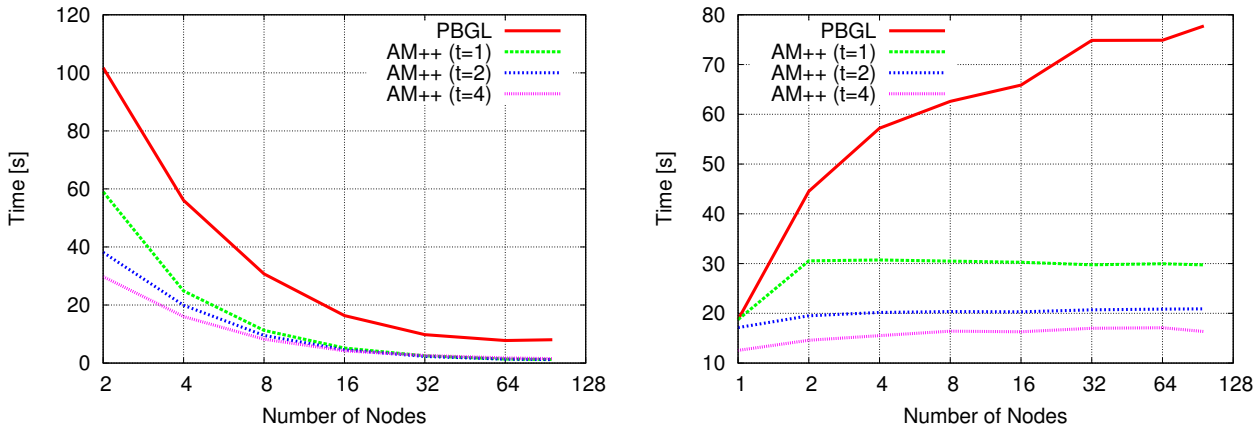
One key benefit of phrasing graph algorithms as message-driven computations is that the work in the algorithm is broken into independent quanta, making fine-grained parallelism straightforward to leverage. Because AM++ is both thread-safe and—more importantly—efficient in the presence of threads, implementing distributed-memory algorithms that utilize fine-grained parallelism on-node was straightforward. This is not the case for the Parallel BGL: the *MPI Process Group* is not thread-safe and would likely require significant effort to be made so. AM++ operations are performed directly by all threads involved in the computation, rather than being funneled to a single communication thread or serialized. While it would be possible to simply run additional processes to use multiple cores on each node, communicating with other processes on the same node using MPI is less efficient than communicating with other threads in the same address space. Furthermore, additional processes would require further partitioning the graph and associated data structures, leading to poorer load balancing.

We benchmark two graph algorithms from the Parallel BGL implemented using the *MPI Process Group* against those same algorithms implemented using AM++ and reusing code from the Parallel BGL extensively. The AM++ implementations are benchmarked utilizing various numbers of threads to demonstrate the effectiveness of combining fine-grained parallelism with AM++. The latest development version of the Parallel BGL and a pre-release version of AM++ were utilized in these tests. We present results on Erdős-Renyi graphs as in the graph exploration kernel.

4.3.1 Breadth-First Search

Breadth-First Search (BFS) is a simple, parallelizable graph kernel and performs a level-wise exploration of all vertices reachable from a single source vertex.

Figure 6 shows the performance of the Parallel BGL's BFS implementation, as well as a BFS implemented using similar but thread-safe data structures and AM++. In the case of the AM++ implementation, all data structures that support concurrent access are lock-free. Figure 6(a) shows the performance of both implementations on a constant-size problem as the number of cluster



(a) Strong scaling (2^{27} vertices and 2^{29} edges).

(b) Weak Scaling (2^{25} vertices and 2^{27} edges per node).

Figure 6: Performance of the Parallel BGL (using the *MPI Process Group*) and AM++ with various numbers of threads performing a parallel breadth-first search.

nodes is increased. The Parallel BGL implementation stops scaling at 64 nodes while the AM++ implementation's runtime continues to decrease in all cases as nodes are added. The AM++ implementation also benefits from additional threads in all cases except with 4 threads on 32–96 processors, likely due to contention as the amount of work available on each node decreases.

Figure 6(b) shows the performance of both implementations on a problem where the work per node remains constant. The Parallel BGL calls the sequential BGL implementation of BFS in the single-processor case, as does AM++ when only one thread is present. Not only does the AM++ implementation perform better and benefit from fine-grained parallelism as threads are added, it also exhibits no increase in runtime as the problem size is increased. This is expected as BFS performs $\mathcal{O}(|V|)$ work, and so increases in the problem size (and work) are balanced by increases in the number of processors available.

We applied duplicate message removal to our BFS implementation but saw no resulting performance benefit. Our tests used a high-speed interconnect, and the BFS handler is fast for redundant messages (because of the early color map check), so the cost of the cache is likely too large compared to the work and bandwidth it saves. Redundant message combining techniques are likely to show a benefit for other applications and systems, however; more expensive handlers, slower networks, and larger messages obtain greater benefit from removing excess messages. One advantage of AM++ is that various caching schemes can be plugged in without extensive modifications to user code; the user's message type is wrapped in the appropriate caching layer and then sends and handler calls occur as usual.

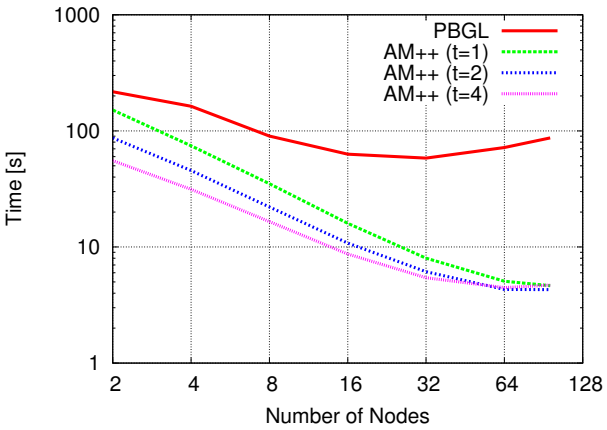
4.3.2 *Parallel Single-Source Shortest Paths*

Single-source shortest paths (SSSP) finds the shortest distance from a single source vertex to all other vertices. A variety of SSSP algorithms exist. The classic algorithm by Dijkstra is *label-setting* in that distance labels are only written once, leading to an inherently serial algorithm. *Label-correcting* algorithms such as [4, 17] are common in parallel contexts, as label-setting algorithms do not parallelize well. Edmonds et al. have found Δ-Stepping [17] to be the best-performing parallel algorithm on distributed-memory systems [6].

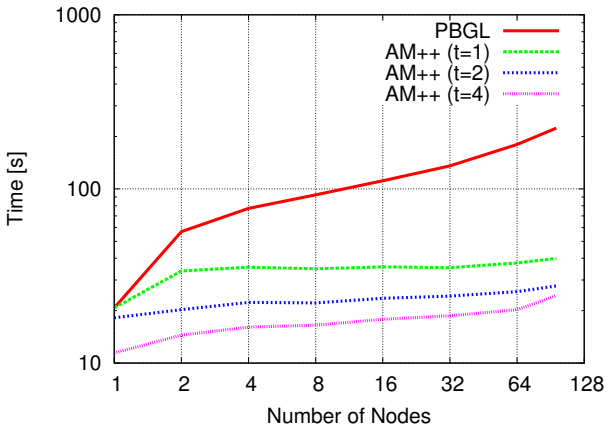
Figure 7 shows the performance of the Parallel BGL's Δ-Stepping implementation vs. the same algorithm implemented with AM++. Slight modifications were performed to the Δ-Stepping algorithm which reduce work efficiency and increase redundant communication but which show dramatic performance improvement in practice. In the strong scaling chart in Figure 7(a) the Parallel BGL implementation scales inversely between 32 and 96 nodes while the AM++ implementation displays an increase in performance as both additional nodes and threads are added in almost all cases. In Figure 7(b), the AM++-based algorithm once again displays almost flat scaling (the runtime is expected to increase proportionally to the problem size because SSSP performs $\mathcal{O}(|V|\ log\ |V|)$ work while the number of nodes increases linearly). The Parallel BGL implementation also exhibits an increase in runtime as the problem size grows, albeit with a significantly steeper slope. AM++ is again able to benefit from additional threads in this case.

Much of the difference in absolute performance is accounted for by the difference in serialization and memory management between the two implementations. AM++'s MPI transport uses MPI datatypes to describe the formats of messages, and the buffer used to accumulate coalesced messages is simply an array of the appropriate type; serialization is done by the MPI implementation if required. On the other hand, Parallel BGL uses explicit serialization using the Boost.Serialization library. In particular, Parallel BGL's message coalescing uses ***MPI_Pack*** to append to a data buffer, leading to extra function calls and data structure traversals. A 64-node profile of Δ-Stepping on a 2^{27}-vertex, 2^{29}-edge graph (the size used in the strong scaling test) shows that 29% of the runtime is spent in these functions. The Parallel BGL approach can send more general objects, including those that cannot be directly described by MPI datatypes; however, AM++ could use a more flexible serialization library while keeping arrays as coalescing buffers.

Another performance difference between the two implementations is in memory management. AM++ uses various memory pools to reduce the use of MPI's memory management functions (***MPI_Alloc_mem*** and ***MPI_Free_mem***); these functions are slow for some interconnects that require data to be pinned in memory. Parallel BGL, on the other hand, uses the MPI memory functions more directly and more often, leading to reduced performance; 31% of the profile described above is spent in these calls.



(a) Strong Scaling (2^{27} vertices and 2^{29} edges).



(b) Weak Scaling (2^{24} vertices and 2^{26} edges per node).

Figure 7: Performance of the Parallel BGL (using the *MPI Process Group*) and AM++ with various numbers of threads computing single-source shortest paths in parallel using Δ-Stepping.

5. CONCLUSION

AM++ is a new library for active messaging, intended for use by irregular applications. It attempts to fit into a "middle ground" between low-level AM libraries such as GASNet and higher-level object-based libraries such as Charm++. AM++ supports message handlers that can send arbitrary messages, simplifying the expression of some applications. A modular design enables configurability by the user without substantial application modification. A mix of object-oriented and generic programming is used to balance runtime flexibility with high performance; type safety is also provided for messages. AM++'s design enables several optimizations, including inlining and vectorization of message handler loops, multithreaded message handlers, and redundant message combining and elimination. AM++ provides these features while having competitive performance to existing AM systems. Graph algorithms written using AM++ have higher performance than those using the previous, ad hoc AM framework in the Parallel Boost Graph Library.

There are several avenues for extensions to AM++. Extra transport layers can be added; we are in the process of implementing one atop GASNet to be able to take advantage of its more direct hardware support. The implementation is not a trivial wrapper: arbitrary actions in handlers, termination detection, message coalescing, and one-sided sending of large messages (without an explicit handshake with the receiver) must be built in terms of GASNet's primitives. Our current GASNet transport always runs AM++ handlers outside GASNet handlers. Optimistic Active Messages [31] could be employed to run suitable portions of AM++ handlers in GASNet handlers. AM++ could also be extended with other approaches to message coalescing (such as the use of multiple threads to run handlers for the same buffer) and other caching strategies for detecting redundant messages. Additional termination detection algorithms could also be added, as could special types of handlers that represent one-sided (remote memory access) operations that might be accelerated in a system's network hardware.

Acknowledgments

This work was supported by a grant from the Lilly Endowment, as well as NSF grant CNS-0834722 and DOE FASTOS II (LAB 07-23). The Odin system was funded by NSF grant EIA-0202048. We also thank Prabhanjan Kambadur and Laura Hopkins for helpful discussions.

6. REFERENCES

[1] Berkeley UPC system internals documentation, version 2.10.0, Nov. 2009. http://upc.lbl.gov/docs/system/index.html.

[2] D. Bonachea. GASNet specification, v1.1. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002. http://gasnet.cs.berkeley.edu/CSD-02-1207.pdf.

[3] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[4] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 722–731. Springer, 1998.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design & Implementation*, pages 137–157, Berkeley, CA, USA, 2004.

[6] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Piscataway, NJ, November 2006.

[7] N. Francez. Distributed termination. *ACM Trans. Program. Lang. Syst.*, 2(1):42–55, 1980.

[8] P. Geoffray. Myrinet eXpress (MX): Is your interconnect smart? In *High Performance Computing and Grid in Asia Pacific Region*, pages 452–452, Washington, DC, USA, 2004. IEEE Computer Society.

[9] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing*, July 2005.

[10] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *IEEE/ACM Supercomputing 2007 (SC'07)*, Nov. 2007.

[11] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable Communication Protocols for Dynamic Sparse Data Exchange. In *Principles and Practice of Parallel Programming*, Jan. 2010.

[12] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX. *International Conference on Parallel Processing Workshops*, pages 394–401, 2009.

[13] L. V. Kalé and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[14] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer. The Deep Computing Messaging Framework: Generalized scalable message passing on the Blue Gene/P supercomputer. In *International Conference on Supercomputing*, pages 94–103, New York, NY, USA, 2008. ACM.

[15] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[16] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, Sept. 1987.

[17] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[18] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009.

[19] R. W. Numrich and J. Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[20] Object Management Group. *CORBA 3.1*, Jan. 2008. http://www.omg.org/spec/CORBA/3.1/.

[21] G. Shah and C. Bender. Performance and experience with LAPI—a new high-performance communication library for the IBM RS/6000 SP. In *International Parallel Processing Symposium*, page 260, Washington, DC, USA, 1998. IEEE Computer Society.

[22] W. Shu and L. V. Kalé. Chare kernel—a runtime support system for parallel computations. *J. Parallel Distrib. Comput.*, 11(3):198–211, 1991.

[23] A. B. Sinha, L. V. Kalé, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, UIUC, 1993.

[24] B. Stroustrup. *Design and Evolution of C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[25] N. Thomas, S. Saunders, T. G. Smith, G. Tanase, and L. Rauchwerger. ARMI: a high level communication library for STAPL. *Parallel Processing Letters*, 16(2):261–280, 2006.

[26] R. Thurlow. *RPC: Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems, May 2009. http://tools.ietf.org/html/rfc5531.

[27] UPC Consortium. *UPC Language Specification, v1.2*, May 2005. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[28] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[29] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pages 256–266, New York, NY, USA, 1992. ACM.

[30] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6:5–7, 1998.

[31] D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, and W. E. Weihl. Optimistic Active Messages: a mechanism for scheduling communication with computation. In *Principles and Practice of Parallel Programming*, pages 217–226, New York, NY, USA, 1995. ACM.