Active Pebbles: Parallel Programming for Data-Driven Applications

Jeremiah J. Willcock
Open Systems Lab
Indiana University
jewillco@cs.indiana.edu

Torsten Hoefler
University of Illinois at Urbana-Champaign
htor@illinois.edu

Nicholas Edmonds, Andrew Lumsdaine
Open Systems Lab
Indiana University
{ngedmond, lums}@cs.indiana.edu

ABSTRACT

The scope of scientific computing continues to grow and now includes diverse application areas such as network analysis, combinatorial computing, and knowledge discovery, to name just a few. Large problems in these application areas require HPC resources, but they exhibit computation and communication patterns that are irregular, fine-grained, and non-local, making it difficult to apply traditional HPC approaches to achieve scalable solutions. In this paper we present Active Pebbles, a programming and execution model developed explicitly to enable the development of scalable software for these emerging application areas. Our approach relies on five main techniques—scalable addressing, active routing, message coalescing, message reduction, and termination detection—to separate algorithm expression from communication optimization. Using this approach, algorithms can be expressed in their natural forms, with their natural levels of granularity, while optimizations necessary for scalability can be applied automatically to match the characteristics of particular machines. We implement several example kernels using both Active Pebbles and existing programming models, evaluating both programmability and performance. Our experimental results demonstrate that the Active Pebbles model can succinctly and directly express irregular application kernels, while still achieving performance comparable to MPI-based implementations that are significantly more complex.

Categories and Subject Descriptors

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

General Terms

Performance, Design

Keywords

Irregular applications, programming models, active messages

ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.

1. INTRODUCTION

Computation is now well-accepted as a "third pillar" of science (complementary to theory and experimentation). High-performance computing (HPC) in particular has enabled computational scientists to expand the frontiers of their disciplines. More recently, a "fourth pillar" of science has been proposed, namely, data-intensive science [14]. The computational resource requirements for data-intensive science are just as vast as for traditional compute-intensive science (consider Google or the Human Genome Project as exemplars). Thus, there is a pressing need to expand the scope of HPC to include data-intensive applications. An important sub-class of data-intensive applications is data-driven applications. In this class of problems, the computational dependencies are embedded in the data and discovered dynamically at run-time.

Traditional compute-intensive applications, such as those based on discretized systems of PDEs, have natural locality (from the local nature of the underlying operators). Scalable applications can therefore be written to solve such problems using a relatively coarse-grained approach, such as the BSP "compute-communicate" model [28]; the communication operations themselves can also be coarse, involving only a few (local) peers per process.

In contrast, data-driven problems tend to be **fine-grained**, i.e., a large number of small objects; **irregular**, i.e., connections between objects cannot be expressed analytically; and **non-local**, i.e., the dependency graph does not have good separators [24]. As a result, these problems are not well suited to the parallel programming approaches that have proven to be so effective for compute-intensive applications. Additionally, most HPC hardware has been developed for problems with compute-intensive characteristics. This hardware has also become increasingly complex, including clusters of multi-core systems that have widely varying communication characteristics within the same machine. These developments further hinder programming with standard models.

Many data-driven problems can scale on traditional HPC hardware. The difficulty is in expressing fine-grained, irregular, non-local computations in such a way as to be able to fully exploit hardware that was designed for coarse-grained, regular, local computations. This can be done (and has been done), with great difficulty, by hand. However, with home-grown solutions like this, the application developer is responsible for developing all layers of the solution stack, not just the application. Furthermore, the application developer is responsible for re-implementing this entire stack when target platforms change, or when new applications must be developed. In many ways, this is similar to the state of affairs that faced the compute-intensive community prior to the standardization of MPI.

Accordingly, an approach is needed that separates data-driven applications from the underlying hardware so that they can be

expressed at their natural levels of granularity, while still being able to (portably) achieve high performance.

To address this need, we have developed Active Pebbles[1] (AP), a new programming model accompanied by an execution model specialized for data-driven computations. AP defines control and data flow constructs for fine-grained data-driven computations that enable low implementation complexity and high execution performance. That is, with the Active Pebbles programming model, applications can be expressed at their "natural" granularities and with their natural structures. The Active Pebbles execution environment in turn coalesces fine-grained data accesses and maps the resulting collective operations to optimized communication patterns in order to achieve performance and scalability.

At the core of the Active Pebbles model are *pebbles*, light-weight active messages that are managed and scheduled in a scalable way, and which generally have no order enforced between them. In addition to pebbles, the AP model includes *handlers* and *distribution objects*. Handlers are functions that are executed in response to pebbles (or ensembles of pebbles) and are bound to data objects with distribution objects to create *targets*. To provide the simultaneous benefits of fine-grained programmability with scalable performance, our model relies on the following five integrated techniques:

1. **Fine-grained Pebble Addressing** — light-weight global addressing to route pebbles to targets.
2. **Message Coalescing** — combining messages to trade message rate for latency and bandwidth.
3. **Active Routing** — restricting the network topology to trade message throughput for latency for large numbers of processes.
4. **Message Reductions** — pebble processing at sources and intermediate routing hops (where possible).
5. **Termination Detection** — customizable detection of system quiescence.

The Active Pebbles model has two distinct aspects: a programming model plus an execution model. Pebbles and targets, in combination with Pebble Addressing (1), define an abstract programming model. The techniques in 2–5 describe an execution model which translates programs expressed using the programming model into high-performance implementations. Accordingly, techniques used in the execution model are not simply implementation details: e.g., message reductions in combination with routing cause a decrease in the asymptotic message complexity of some algorithms and are thus essential to our model.

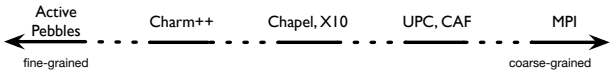
2. RELATED WORK

Solutions to data-driven problems on shared memory machines has been studied by several groups [2, 4, 18]. A fundamental result from these efforts is that locking can limit performance due to lock contention and additional memory traffic, decrease programmer productivity, and stop progress in faulty environments [12]. Predefined atomic memory operations such as *compare and swap* or *fetch and add* allow the design of non-blocking and wait-free algorithms but their expressiveness is limited [13] (e.g., to integer addition). Transactional Memory provides an extension to the shared memory programming model by allowing user-defined operations (*transactions*) which succeed or fail atomically, thus reducing programming complexity while still enabling non-blocking and wait-free algorithms [12, 25].

[1]The term Active Pebbles expresses the idea that messages are active and independent, but without individual identity (transported and processed in bulk).

Scalable computing systems are by necessity distributed memory machines with multiple coherence domains, and are thus more complex to program. Message passing, an effective programming model for regular HPC applications, provides a clear separation of address spaces and makes all communication explicit. The Message Passing Interface (MPI) is the de facto standard for programming such systems [21]. However, irregular and dynamic applications often need shared access to data structures which naturally cross address spaces. MPI-2 One Sided [21, §11] and Partitioned Global Address Space (PGAS) [22, 27] models strive to fill this gap by allowing transparent access to remote memory in an emulated global address space. However, mechanisms for concurrency control are limited to locks and critical sections; some models support weak predefined atomic operations (e.g., *MPI_Accumulate()*). Stronger atomic operations (e.g., compare and swap, fetch and add) and user-defined atomic operations are either not supported or do not perform well. Thus, we claim that these approaches do not provide the appropriate primitives for fine-grained data-driven applications. Just as Transactional Memory generalizes processor atomic operations to arbitrary transactions, Active Pebbles generalizes one-sided operations to user-defined pebble handlers.

Active Pebbles are similar to active messages [29]. However, active messages are mostly used for low-level communication layers that are not exposed to the users. Other advanced object communication models like Charm++ [17], X10 [7], and ParalleX [10] are also based on active messages behind the scenes. Our approach differs from these approaches in that AP has a much finer natural granularity (i.e., AP naturally expresses fine-grained problems while also obtaining high performance). Moreover, our approach allows direct expression of operations on fine-grained distributed items. The figure below compares the natural granularities of AP with other programming models.



Our execution model could be a compilation target for an active PGAS language such as X10 or Chapel [6]; however, we claim that the techniques described in our execution model are required to allow those languages to efficiently target fine-grained applications—and are thus a substantial part of our contribution.

2.1 Comparative Example

We compare and contrast related approaches with a simple example problem. Assume that each of P processes wants to insert n items into a hash table which is statically distributed across the P processes and uses chaining to resolve collisions. The keys for the hash table are uniformly distributed in $[0,N)$, $N \gg P$. We now compare possible implementations in different programming models.

MPI. In one possible MPI implementation, each process would collect distinct sets of items, each destined for one remote process. After each process inserted all n requests, all processes would participate in a complete exchange. This communication can either be done with direct sends or with a single *MPI_Alltoallv()* operation (plus an *MPI_Alltoall()* stage to determine the pairwise message sizes). Each process would receive and add items to its local portion of the hash table. On average, each would receive and process $\Omega(n)$ items from $\Omega(P)$ peers, incurring a cost of $\Omega(n+P)$.

PGAS. A possible PGAS implementation would create the hash table in the global address space and each process would add

items directly ensuring mutual exclusion by locking. This would need $\Omega(n)$ lock/unlock messages in addition to the $\Omega(n)$ data transfers per process. Resolving collisions is likely to require further messages and locks (e.g., to allocate additional space).

Object-Oriented. In object-oriented parallel languages, such as Charm++ [17] or X10 [7], the hash table would be a global object (e.g., a *Chare*). Each item would trigger a member function (e.g., insert) of the hash table object. For large n and P the vast number of remote invocations and their associated management overhead, as well as the small amount of computation per object, would impact performance significantly.

Active Pebbles. Figure 1 shows a schematic view of the Active Pebbles execution model. In an Active Pebbles implementation the user would specify a handler function which adds data items to the local hash table. The user would then send all data elements successively to the handler for each individual key (which is globally addressable). The Active Pebbles framework takes these pebbles and coalesces them into groups bound for the same remote process (two items with keys 6 and 7 sent from process P2 to P3 in Figure 1). It can also perform reductions on these coalesced groups of messages to eliminate duplicates and combine messages to the same target key (shown at P3 "single-source reduction" in Figure 1). Active routing sends all (coalesced) messages along a virtual topology and applies additional coalescing and reductions at intermediate hops (Figure 1 shows routing along a hypercube, i.e., P0 sends messages to P3 through P1 where they are coalesced and reduced, "multi-source reduction," with other messages).

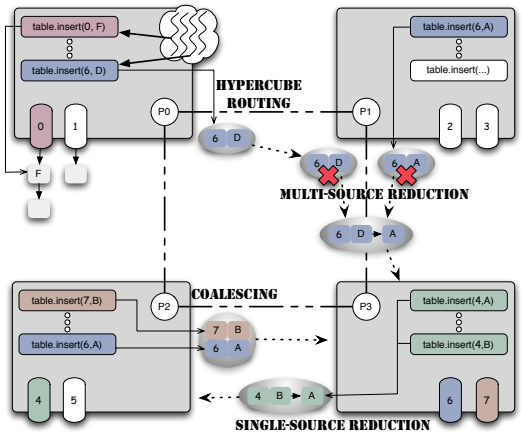


Figure 1: Overview of the Active Pebbles model.

This example shows the advantages of the Active Pebbles model, and how PGAS and object oriented models are inherently limited by the per-process message rate. Although MPI's flexibility allows the use of collective communication, high programming effort would be required to implement coalescing and multi-stage communication manually. The Active Pebbles model performs such optimizations transparently and eliminates redundant communication.

3. THE ACTIVE PEBBLES MODEL

In this section we analyze each of the Active Pebbles mechanisms in detail. Sections 3.1 and 3.2 describe the programming model while Sections 3.3–3.6 describe the execution model. We utilize the well-known LogGP model [1] as a framework for formal analysis. The LogGP model incorporates four network parameters: L, the maximum latency between any two processes; o, the CPU injection overhead of a single message; g, the "gap" between two messages, i.e., the inverse of the injection rate; G, the "gap" per byte, i.e., the inverse bandwidth; and P, the number of processes. We make two modifications to the original model: small, individual messages (pebbles) are considered to be of one-byte size, and we assume that the coefficient of G for an n-byte message is n rather than $n-1$ as in the original LogGP model. LogGP parameters written with a subscript p refer to pebble-specific parameters imposed by our *synthetic network*. For example, L_p is the per-pebble latency, which might be higher than the network latency L due to active routing or coalescing; analogously, o_p, g_p, and G_p are the overhead, gap, and gap per byte for a pebble.

3.1 Active Pebbles Abstractions

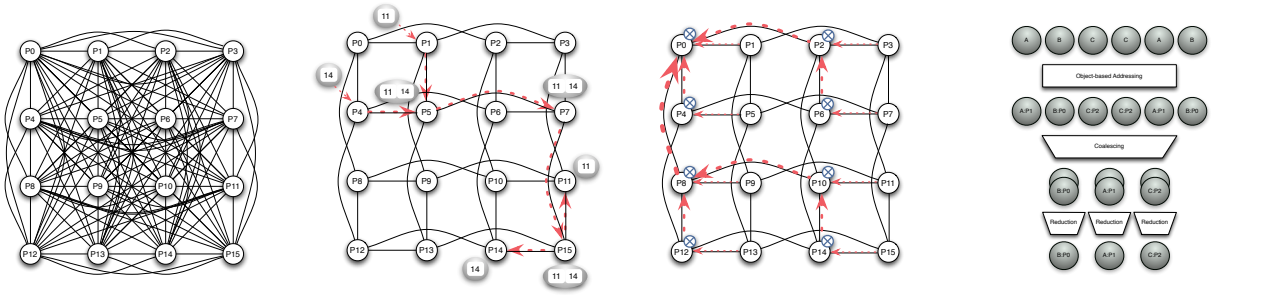
The primary abstractions in the Active Pebbles model are pebbles and targets. Pebbles are light-weight active messages that operate on targets (which can, transparently, be local or remote). Targets are created by binding together a data object with a message handler, through the use of a distribution object. In Figure 1, targets are the destination buckets and their keys are used directly as destination addresses. The distribution object in this example maps each key i to process $\lfloor i/2 \rfloor$. Pebbles flow through the network from their source ranks to their destination ranks (determined by the distribution object). Pebbles are unordered (other than by termination detection), allowing flexibility in processing (such as threading, when the underlying handler function is thread-safe, or other forms of acceleration).

3.2 Fine-Grained Pebble Addressing

In the Active Pebbles model, messages are sent to individual targets, not process ranks. Target identifiers (which are typically domain-specific) are converted to ranks using a user-defined distribution object. Target identifiers form a global address space, as in other GAS models. Active Pebbles supports both static and dynamic distributions. When a static distribution is used, the distribution will often require only constant space and constant time for location computations. A dynamic distribution is likely to require larger amounts of storage; however, applications need not use dynamic distributions if static ones will suffice. Note that, unlike some other object-based messaging systems, Active Pebbles does not require any particular information to be kept to communicate with a target; i.e., there is no setup required to communicate, and a sending thread does not require any local information about the destination target (other than the distribution, which can be shared by many targets). A target identifier can be as simple as a global index into a distributed array; such identifiers can be created and destroyed at will, and are thus very lightweight. This mechanism is similar to *places* in X10 or *Chare arrays* in Charm++ but those mechanisms enable migration and other other advanced management and thus require $\mathcal{O}(n)$ time and space overhead to manage n elements; a statically-distributed array in Active Pebbles would only require $\mathcal{O}(1)$ space overhead to manage n elements, on the other hand. To simplify applications, the fine-grained addressing layer traps pebbles destined for the sending node and calls the corresponding handler directly, avoiding overheads from message coalescing, serialization, etc. In LogGP terms, addressing moves some time from the application to the model's o_p term without an overall change to performance.

3.3 Message Coalescing

A standard technique for increasing bandwidth utilization is message coalescing. Message coalescing combines multiple pebbles to the same destination into a single, larger message. This



(a) Full system (as exposed by MPI). (b) Pebble routing in a synthetic hypercube network. (c) Pebble reduction in a synthetic hypercube network. (d) Coalescing and reductions on idempotent objects bound for processes $P0$, $P1$, and $P2$.

Figure 2: The Different Active Pebbles Features.

technique is well-known in the MPI arena and often performed manually. In LogGP terms, the packing of n 1-byte messages into one, n-byte message changes the overall message latency from $(n+1)o+L+G+(n-1)g$ to $2o+L+nG$. Thus, coalescing improves performance if $G < o+g$, which is true for all practical networks. However, coalescing works against pipelining. While without coalescing, the first pebble can be processed at the sender after $2o+L+G$, coalescing delays this to $2o+L+nG$. Thus, coalescing improves bandwidth utilization and reduces the use of g and o at cost of per-pebble latency L_p. Finding the optimal coalescing factor is system-dependent and subject to active research. Our LogGP discussion can be used as a good starting point and, in Active Pebbles, a user can specify the coalescing factor at run-time. Also, it may not be known exactly when the stream of pebbles will end, and so a timeout or similar scheme can be used to know when to stop accumulating pebbles. Active Pebbles includes a *flush()* call to send pebbles immediately.

One disadvantage of message coalescing is its use of memory. For fast access to buffers, one buffer must typically be kept for each possible message destination. Some implementations also use an individual, preallocated message buffer to receive messages from each possible sender. Those buffers can be established lazily, i.e., allocated at first use, however, reserving space for P buffers per process is impractical for large P if the communication is dense. The next feature, active routing, is an efficient technique to reduce the number of such buffers and increase performance.

3.4 Active Routing

Highly irregular applications often send messages from each process to almost every other process. Thus, the packet injection rate on each process's outgoing network links and the number of coalescing buffers become performance bottlenecks. To work around these limitations, our implementation of the Active Pebbles model can route messages through one or more intermediate hops on their ways to their destinations. For example, a hypercube topology can be embedded, with messages routed in such a way that they are only sent along the edges of the hypercube. Thus, each node only sends directly to a small number of other nodes, allowing fewer, but larger, coalesced messages to be sent. Users can also define their own routing schemes, including dynamic routing for fault tolerance.

Several researchers have found (e.g., Bruck et al [5]) that software-based routing improves performance on personalized (all-to-all) communications. For example, Bader, Helman, and JáJá show that adding one intermediate node into each message's path leads to a substantial performance improvement [3]; Plimpton et al show a performance benefit, both in theory and in practice, of simulating a hypercube topology for the HPCC RandomAccess benchmark [23]. Garg and Sabharwal discuss the benefits of manual software routing along a virtual torus on Blue Gene/L [11] and Yoo et al. [32] use manual routing and message reductions to optimize breadth-first search at large-scale. The original paper that introduced the LogGP model shows a personalized broadcast (*MPI_Scatter()*) operation that uses tree-based routing [1], reducing the number of messages sent from $\mathcal{O}(P^2)$ to $\mathcal{O}(P\log P)$, although the total number of message bytes is increased by a factor of $\log P$. In LogGP terms, the assumption made is that $o+g$ is large enough compared to G that it is acceptable to use extra bandwidth to reduce the number of messages sent.

In many networks, routing increases the number of links used by any given pebble, leading to greater bandwidth utilization. However, routing allows pebbles from multiple sources and/or to multiple destinations to be packed together into a single, larger message, increasing the effectiveness of coalescing compared to a non-routed network. A similar argument applies to reductions across pebbles from multiple source nodes to the same destination target; see Section 3.5 for an analysis. Thus, active routing can reduce the number of packets sent across the network, potentially leading to less congestion and a performance improvement.

Active routing also reduces the amount of memory needed for message coalescing because it limits k, the number of other processes that each process communicates with. Without routing, $k = P-1$, and thus each node requires a large number of communication buffers. Hypercube routing, with $k = \log_2 P$, reduces the number of buffers needed from $\Theta(P)$ to $\Theta(\log P)$.

Active routing, in combination with other AP features, effectively converts fine-grained point-to-point messaging operations into coarse-grained optimized "collective" operations. For example, one source sending separate pebbles to different destinations will, by combining routing and message coalescing, actually use a tree-based scatter operation that takes advantage of large message support in the network (as in [1]). Similarly, a number of sources sending pebbles to the same target will combine them into a tree-based, optimized gather operation (similar to [5]).

Figure 2(b) shows routing along a hypercube (two messages are coalesced at $P5$ and split up at $P15$; routing reduces the message volume significantly). Messages flow only along the edges of the hypercube compared with (logically) direct all-to-all routing in Figure 2(a). Active Pebbles' synthetic topologies can be optimized for the topology of the physical communication network, as with MPI collective operations (e.g., [5]).

3.5 Message Reductions

In many applications, multiple pebbles of the same type to the same object are redundant: duplicate messages can be removed or combined in some way. We call this optimization message

reduction; it can occur either at a message's sender or—with active routing—at an intermediate node. Reduction is implemented using a cache. For duplicate removal, previous messages are stored in a cache at each process; messages found in the cache are ignored. As analyzed below, lookup cost is important for the benefit of message reductions, requiring a fast, constant-time cache lookup at the expense of hit rate. In our experiments, we use a direct-mapped cache with runtime-configurable size; a miss replaces the contents of a cache slot with the new pebble.

For messages with data payloads, two cases are possible: the reduction operation is max (in some ordering; min is dual), in which case the cache simply removes messages with suboptimal values; or the reduction operation is something else that requires messages to be combined. In the latter case, message data payloads can be concatenated or combined (e.g., additively) as shown in Figure 1. In LogGP terms, reductions replace n messages by $n(1-h)$ messages (where h is the cache hit rate), but they also increase the value of o_p for each pebble to $o_p + c$, where c is the average cost of searching and maintaining the cache. The decrease in messages from n to $n(1-h)$ is equivalent to sending n messages but replacing G by $c+(1-h)G$. With message coalescing, G, c, and the message count are the only important factors in messaging performance; other factors are constant overheads. Without reductions, the time to send one pebble is G; reductions reduce the (expected) time to $c+(1-h)G$, leading to a benefit when $hG > c$.

This analysis only considers the effect of reductions on message latencies and bandwidth consumption; however, the reduction in computation at the target is likely to be even more important. Reductions reduce the expected processing cost p for each pebble to $p(1-h)$. Messages requiring expensive computation thus benefit from reductions by reducing the number of those computations that occur. In particular, if a message can trigger a tree of other messages, one reduction at the source of that message can prevent many others from being sent at all.

Active routing can increase the benefit of reductions by allowing reductions across messages from multiple sources to the same target. With routing, a message is tested against the cache of every node along its path, increasing the chance of a match being found. For example, in a hypercube, each message is tested against up to $\log_2 P$ caches. Assuming the probabilities of hitting in each cache are independent, the overall hit rate becomes $1-(1-h)^{\log_2 P}$ (approximately $h \log_2 P$) rather than the rate h for a single cache. These multiple checks thus may lead to a greater reduction in message volume than would occur without routing.

When messages are sent from different sources to one target, routing and local message reductions at intermediate nodes combine to synthesize a reduction tree as would be used by an optimized implementation of *MPI_Reduce()*. This emergent property creates an efficient collective operation from point-to-point messages, with the routing algorithm defining the structure of the generated tree. Hypercube routing, for example, would generate binary reduction trees (Figure 2(c) shows an all-to-*P*0 reduction) with a logarithmic number of stages.

3.6 Termination Detection

The Active Pebbles model allows the handler for each pebble to trigger new communications. Thus, in a message-driven computation, it is non-trivial to detect the global termination (quiescence) of the algorithm. In the standard model for termination detection [8], each process can either be active or passive. Active processes perform computation and can send messages, as well as become passive. Passive processes can only be activated by incoming messages. A computation starts with one or more active processes and is terminated when all processes are passive and no messages are in flight. Many algorithms are available to detect termination [20], both for specialized networks and general, asynchronous message passing environments.

The optimal termination detection algorithm for an algorithm can depend on the features of the communication subsystem and on the structure of the communication (dense or sparse). Our framework enables easy implementation of different algorithms by providing several hooks into the messaging layer. We implement one fully general termination detection scheme (SKR [26]), using a nonblocking *allreduce()* operation [15], similar to the four-counter algorithm in [20].

3.6.1 Depth-Limited Termination Detection

Some applications can provide an upper bound for the longest chain of messages that is triggered from handlers. For example, a simple application where each message only accumulates or deposits data into its target's memory does not require a generic termination detection scheme. Another example would be an application that performs atomic updates on target locations that return results (e.g., read-modify-write). These examples would require termination detection of depths one and two, respectively. Graph traversals can generate chains of handler-triggered messages of unbounded depth (up to the diameter of the graph), however. Several message-counting algorithms meet the lower bound discussed in [16] and detect termination in $\log P$ steps for depth one, while unlimited-depth termination detection algorithms usually need multiple iterations to converge. In the SKR algorithm, termination detection takes at least $2 \log P$ steps (two *allreduce()* operations). Our framework offers hooks to specify the desired termination detection depth to exploit this application-specific knowledge. We implement two depth-one termination detection schemes: a message-counting algorithm based on nonblocking *reduce_scatter()* [15], and an algorithm that uses nonblocking barrier semantics and is able to leverage high-speed synchronization primitives [16]. Both algorithms are invoked n times to handle depth-n termination detection.

3.6.2 Termination Detection and Active Routing

In active routing, each message travels over multiple hops which increases the depth of termination detection. For example in hypercube routing, an additional $\log_2 P$ hops are added to the termination detection. This is not an issue for detectors that can handle unlimited depths, but it affects limited-depth detection. With s-stage active routing and a depth-n termination requested by the application, limited-depth termination detection would take $n \cdot s$ steps. However, termination detection could take advantage of the smaller set of possible neighbors from active routing, such as the $\log_2 P$ neighbors of each node in a hypercube. This would reduce the per-round time to $\mathcal{O}(\log \log P)$, for a total time of $\mathcal{O}(\log P \log \log P)$ as compared to $\mathcal{O}(\log P)$ without routing. Routing may benefit the rest of the application enough to justify that increase in termination detection time, however.

3.7 Synthetic Network Tradeoffs

The Active Pebbles model uses coalescing, reductions, active routing, and termination detection to present an easy-to-use programming model to the user while still being able to exploit the capabilities of large-scale computing systems. Active Pebbles transforms fine-grained object access and the resulting all-to-all messaging into coarse-grained messages in a synthetic, or overlay, network. The synthetic network transparently transforms message streams into optimized communication schedules similar to those used by MPI collective operations. Various aspects of the Active

Pebbles execution model can be adjusted to match the synthetic network to a particular application, such as the coalescing factor, the synthetic topology, and termination detection. The programming interface remains identical for all of those options. Active Pebbles can thus be optimized for specific machines without changes to application source code, allowing performance-portability.

4. APPLICATION EXAMPLES

We examine four example applications using the Active Pebbles model in order to explore the expressiveness and performance of applications written using it. Implementations of these applications in three models are evaluated: Active Pebbles; MPI; and Unified Parallel C (UPC), which we use to illustrate the programming techniques used in PGAS languages.[2] We first present a simple summary of these applications with more detailed explanations to follow:

RandomAccess randomly updates a global table in the style of the HPCC RandomAccess benchmark [19]. We use optimized reference implementations where appropriate, as well as simplified implementations.

PointerChase creates a random ring of processes and sends messages around the ring.

Permutation permutes a data array distributed across P processes according to another distributed array, as might be used to redistribute unordered data after loading it.

Breadth First Search (BFS) is a graph kernel that explores a random Erdős-Rényi [9] graph breadth-first.

4.1 RandomAccess

The parallel RandomAccess benchmark measures the performance of updating random locations in a globally distributed table [19]. The benchmark resembles access patterns from distributed databases and distributed hash tables. It uses a global *table* of N elements distributed across P processes. The timed kernel consists of $4N$ updates to the table of the form *table[ran % N] ^= ran* where *ran* is the output of a random number generator. Processes may not buffer more than 1024 updates locally.

PGAS. In UPC the *table* can be allocated in the shared space and accessed just as in the sequential version of the algorithm:

UPC
```
uint64_t ran;
shared uint64_t* table = upc_all_alloc(N ,sizeof(uint64_t));
for (int i = 0 ; i < 1024 ; ++i) {
  ran = (ran << 1) ^ (((int64_t)ran < 0) ? 7 : 0); // compute index
  table[ran % N] ^= ran; // perform update
}
```

The UPC compiler/runtime then performs the necessary communication to perform the update to *table*.

MPI. MPI has no notion of shared data structures, so the updates to non-local portions of the table must be explicitly communicated to the remote process which then applies them. Rather than sending individual updates we buffer 1024 updates sorted by destination then communicate them collectively. The MPI implementation of the RandomAccess application first buffers local updates, then communicates the number of updates followed by the updates themselves:

[2]"PGAS" here refers to fine-grained remote memory accesses (the basic PGAS model), without active message extensions.

MPI
```
for (int i = 0 ; i < 1024 ; ++i) {
  ran = (ran << 1) ^ (((int64_t)ran < 0) ? 7 : 0); // compute index
  long index = ran % N;
  int owner = index / (N/P);

  // perform local update
  if (rank == owner)
    table[index % (N/P)] ^= ran;
  else // remote
    out_bufs[owner].buf[out_bufs[owner].count++] = ran;
}
// ... allocate and prepare all-to-all communication buffers
MPI_Alltoall(out_bufs.count,. . . ,in_bufs.count,. . . );
// ... allocate and prepare all-to-allv communication buffers
MPI_Alltoallv(out_bufs.buf,out_bufs.count,. . . ,
              in_bufs.buf,in_bufs.count,. . . );
```

AP. In Active Pebbles we invoke remote handlers using pebbles. To implement RandomAccess we first encapsulate the update operation inside a handler:

AP
```
struct update_handler {
  bool operator()(uint64_t ran) const
  { table[ran % (N/P)] ^= ran; } // update to table
};
```

This handler is then invoked from a remote process by creating a pebble type and assigning the handler to it. The pebble type encapsulates pebble addressing (through the *block_owner_map* type) and routing (through the *hypercube_routing* object passed to the type's constructor). A separate operation attaches a particular handler object to the pebble type:

AP
```
pebble_addressing_dest_hbr<. . . >
update_msg(transport, . . . , block_owner_map(N/P),
           hypercube_routing(rank, size));
update_msg.set_handler(update_handler(table));
```

Active Pebbles detects messages which would be sent to the current rank using pebble addressing and simply calls the appropriate handler directly. This eliminates the need for applications to treat local and remote data differently:

AP
```
for (int i = 0 ; i < 1024 ; ++i) {
  ran = (ran << 1) ^ (((int64_t)ran < 0) ? 7 : 0);
  update_msg.send(ran);
}
```

4.2 PointerChase

The PointerChase application creates a random permutation of $[0,P)$; each processor i then relays a single, small message to element $(i+1) \mod P$ of the permutation. This benchmark is intended to model the performance of chains of dependent operations in an irregular application. It primarily tests message latency, and thus is expected to favor PGAS models.

PGAS. In UPC, notifying the next process to relay the message can be implemented by polling on a counter allocated in the shared space and waiting for its value to be updated:

```
shared int* flags = upc_all_alloc(THREADS, sizeof(int));                    UPC
for (int i = 0; i < rounds; ++i) {
  while (flags[MYTHREAD] != i) {}
  flags[next_rank] = i;
}
```

MPI. The implementation of the PointerChase application is similar in MPI, except that messages replace the memory operations and *MPI_Wait()* is used instead of polling (example simplified):

```
for (int i = 0; i < rounds; ++i) {                                          MPI
  MPI_Recv(&data, 1, MPI_ANY_SOURCE, . . . );
  MPI_Send(&next_rank, 1, next_rank, . . . );
}
```

AP. In Active Pebbles there is no main loop at all; the control flow is entirely represented in the message handler:

```
struct msg_handler {                                                        AP
  bool operator()(int source, const int* data, int count) {
    if (rank != start) msg->send(round, next);
    else if (--round > 0) msg->send(round, next);
} };
```

The message handler is initialized with the rank which sends the first message (*start*), the number of loops around the ring to perform (*round*), and the next rank in the ring (*next*). After the handlers are initialized all that is required to start the application is for the *start* rank to send the first message:

```
typed_message<. . . >::type                                                 AP
  msg(typed_message<. . . >
        ::make(transport, 1, msg_handler(0, rounds, next, msg)));
if (rank == start) msg.send(0, next);
```

4.3 Permutation

The Permutation application is designed to be representative of a common task in scientific computing: distributing and permuting unsorted data (e.g., after it has been read from a file). The data distribution may exist to optimize locality, provide load-balancing, or for domain-specific reasons. Permutation uses three arrays representing input data (*data*), a permutation (*perm*), and the reordered data ($data_{perm}$). These arrays each contain N elements, and each is distributed across P processors. The *data* array contains an index into the *perm* array which functions as a unique identifier for the data element, as well as some associated data. The *perm* array contains the destination for each input element; the inverse of the permutation *perm* is applied. The result satisfies $\{\forall i \in [0,N) : data_{perm}[perm[data[i]]] = i\}$.

PGAS. In UPC we allocate the arrays in the shared space and use *upc_forall()* to distribute the work:

```
upc_forall (uint64_t i = 0; i < N; ++i; &data[i])                           UPC
  data_perm[perm[data[i]]] = i;
```

In the preceding example we simply store the indices i into $data_{perm}$; a real application using this technique would assign the application data associated with index i.

MPI. In the MPI implementation of Permutation a similar communication pattern to that described in Section 4.1 is used. Rather than a single *Alltoall()*/*Alltoallv()* round, two rounds are required for Permutation. In the first round, elements of *data* are sent to the process that stores $perm[data[i]]$. In the second round *perm* is applied and the data sent to the process which owns $data_{perm}[perm[data[i]]]$. We have omitted the lengthy code for the MPI implementation of Permutation in the interest of brevity.

AP. The Active Pebbles implementation combines the movement of *data* and the application of *perm* into a single phase using dependent messages and depth-two termination detection to detect completion. The MPI implementation must address the situations where some dependent elements of *data*, *perm*, and $data_{perm}$ may be local and others remote. Active Pebbles handles these locality concerns automatically because pebbles sent to local targets will simply call the correct underlying handler with no performance penalty. The Active Pebbles implementation uses two handlers, the first receives elements of *data*, applies *perm*, and sends a pebble to the owner of the appropriate target in $data_{perm}$:

```
struct permute_handler {                                                    AP
  bool operator()(const pair<uint64_t, uint64_t>& x) const {
    uint64_t target_idx = perm[get(local_map, x.first)];
    put->send(make_pair(target_idx, x.second));
} };
```

The second handler is the *put* handler used by the *permute_handler*. This handler writes values to the specified target in $data_{perm}$:

```
struct put_handler {                                                        AP
  bool operator()(const put_data& x) const
  { data_perm[get(local_map, x.first)] = x.second; }
};
```

After the message handlers are initialized, each process simply sends pebbles for all of its local data:

```
make_coalesced_mt<. . . >::type data_permute(transport, . . . );            AP
for (int i = 0; i < N/P; ++i)
  data_permute.send(make_pair(data[i], N/P * rank + i));
```

Once termination detection completes, all pebbles sent (both originally and from handlers) will have been processed, and so all elements of $data_{perm}$ will have been updated.

4.4 BFS

The final application we consider is breadth-first search on a directed graph. Graph algorithms are an excellent application of the Active Pebbles model because they often create many fine-grained asynchronous tasks. All of the implementations use an Erdős-Rényi random graph with a one-dimensional vertex distribution across the P processors.

In a bulk-synchronous implementation BFS is implemented with a single logical distributed queue, composed of logical queues on each process. A *push()* operation on any process results in a vertex being placed on the local queue of the vertex's owning process. Neither MPI nor UPC support dynamic shared data structures such as queues directly. Buffering queue operations at the source and applying them collectively addresses this limitation in both cases. In the case of MPI, this is necessary because *MPI_Accumulate()* is not expressive enough to implement queue update operations; e.g., it cannot atomically fetch and increment a counter. In UPC, vertices could be pushed directly onto the targets' local queues,

but the queues would then need to be locked remotely, limiting concurrency.

PGAS + MPI. The MPI and UPC implementations use similar algorithms:

MPI/UPC Pseudocode

```
if (source is local) Q.push(source);
while (!Q.empty()) {
  for (v : Q)
    if (visited[v] == 0) {
      visited[v] = 1;
      for (w : neighbors[v]) {Q2.push(w);}
    }
  Q.clear(); swap(Q, Q2);
}
```

AP. In Active Pebbles we can choose a formulation of the BFS algorithm that expresses a better mapping to the programming model. Level-wise traversals of the BFS tree are possible using a queue to buffer pebbles, as are versions which compute a BFS numbering using a single-source shortest path algorithm. The latter approach is asymptotically more expensive, but significantly reduces the synchronization required and thus may be desirable in practice. An implementation in Active Pebbles would define a handler which utilized a *distance* property for each vertex:

AP

```
struct bfs_handler {
  // x is a ⟨vertex, distance⟩ pair
  bool operator()(const pair<Vertex, int>& x) const {
    if (x.second < distance[x.first]) {
      distance[x.first] = x.second;
      explore->send(x.first, x.second + 1);
} } }; // explore is an instance of a message type
```

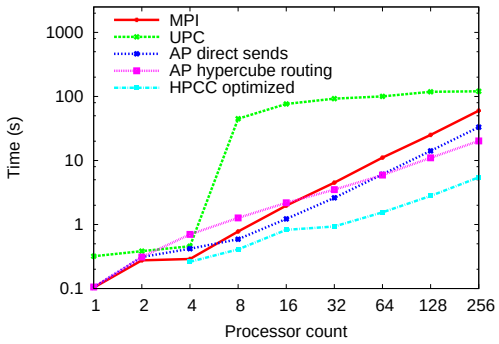
In this formulation there is no need for any queues: all the message buffering and work coalescing performed by the queue in the other implementations is performed by Active Pebbles. Running the algorithm simply requires exploring the source vertex by sending a message to a *bfs_handler()*.

5. EXPERIMENTAL EVALUATION

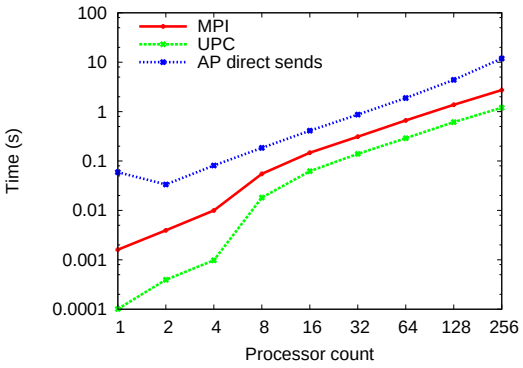
We used Odin, a 128-node InfiniBand cluster (Single Data Rate), for our performance experiments. Each node has two 2 GHz dual-core Opteron 270 CPUs and 4 GiB of RAM. Our experiments used Open MPI 1.4.1, OFED 1.3.1, and Berkeley UPC 2.10.2 (compiled with `--disable-multirail --enable-pshm`). We used `g++` 4.4.0 as the compiler in our experiments (including as the back-end compiler for MPI and UPC). Except for single-node runs, all tests used four MPI processes per node. Our Active Pebbles implementation, written in standard C++ using the AM++ active message library [31], uses MPI as its underlying communication mechanism. All scaling experiments test weak scaling, so data sizes are reported per processor rather than globally.

5.1 Implementation Details

The coalescing buffer size (Section 3.3) was 4096 elements for BFS and Permutation, 1024 for RandomAccess (due to the lookahead limit in that benchmark), and no coalescing was used for PointerChase. Our implementation uses advanced C++ features to allow all pebble handlers for a coalesced message buffer to run in a single, statically analyzable loop, avoiding dynamic dispatch at the level of a single pebble. Routing experiments use a hypercube



(a) RandomAccess (2^{19} elements per processor).



(b) PointerChase

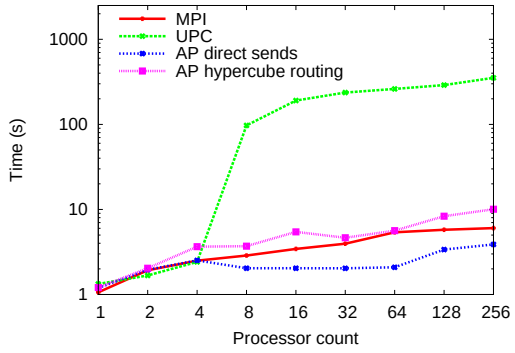
Figure 3: Benchmark Results (part 1).

topology (as in [23]), the best-performing of the four topologies implemented; termination detection used the PCX (reduce-scatter) algorithm from [16], generalized to support multiple termination detection levels but unaware of message routing.

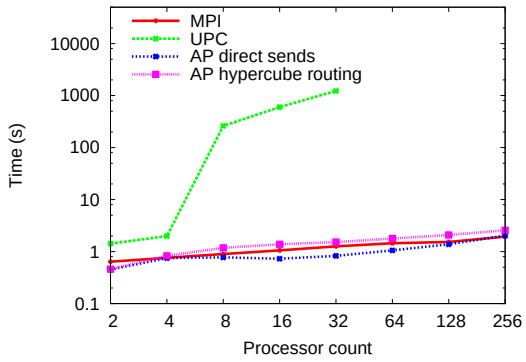
5.2 Results

RandomAccess. In Figure 3(a), we see the performance of the RandomAccess benchmark implementations. Because of the availability of official, highly optimized, MPI-based implementations of the benchmark, we also compared to those (HPCC version 1.4.1 using Sandia Opt 2, the fastest version on Odin). The graph shows that our current Active Pebbles implementation performs worse than the optimized, specialized, and more complex HPCC implementation [23]. However, Active Pebbles performs better than the reference UPC implementation and our MPI implementation of similar implementation complexity (see Section 4). Active routing is slower than a non-routed implementation on small node counts; routing has a cost in latency and its advantages (such as reduced buffer memory usage) do not appear at small scales. Once beyond 128 processors (32 nodes), however, hypercube routing provides a performance benefit. Similarly, our MPI implementation is fast on small process counts, but it too suffers from the use of many sends (through the *MPI_Alltoallv()* collective) at large scales.

PointerChase. Figure 3(b) shows Odin's performance on the PointerChase latency benchmark. This benchmark shows a clear performance benefit for UPC over both MPI and Active Pebbles, which is expected because PGAS models are designed for sending fine-grained, asynchronous messages with very low latencies and the network supports remote direct memory access (RDMA). Active Pebbles, unlike UPC, is designed to support a large *volume*



(a) Permutation (2^{22} elements per processor).



(b) BFS (2^{19} vertices per processor).

Figure 4: Benchmark Results (part 2).

of small messages, emphasizing throughput for millions or billions of messages over individual message latency. Even with message coalescing disabled, overheads from dynamic memory allocation and other features to support asynchronous messages still hindered Active Pebbles's performance.

Permutation. The results for the Permutation benchmark are shown in Figure 4(a). For this experiment, 2^{22} eight-byte elements were permuted on each processor. As can be seen from the graph, once multiple nodes (rather than processors on the same node) are in use, UPC's performance degrades substantially. On the other hand, the Active Pebbles and MPI versions exhibit almost linear weak scaling. In this benchmark, active routing was not useful—sending messages directly between processes performed better. Unlike RandomAccess's 1024-element lookahead limit, elements in Permutation can be streamed at any rate and elements never need to wait for previous elements to complete. Thus, termination detection is done only at the end of the overall benchmark, rather than periodically within it. Additionally, without routing, message handlers only send messages to a limited, fixed depth, enabling use of a specialized termination detection algorithm. When routing is used, on the other hand, messages can be nested to depth $2\log_2 P$, and so the generalized PCX termination detection algorithm must perform that many global communications. An optimized implementation could replace those by localized operations with each node's neighbors in the hypercube.

BFS. Figure 4(b) shows the performance of the BFS benchmark on Odin. Note that the BFS implementation for AP tested here is level-synchronized; i.e., the message handler inserts each incoming vertex into a queue to be processed in the next level, but does not itself directly trigger the exploration of other vertices. As in the other benchmarks (except PointerChase), the MPI and AP implementations show minimal increase in runtime as problem size is increased (because the experiment uses weak scaling), while the UPC version's time grows quickly as more nodes are added. UPC results beyond 32 processors were not shown because the version failed to finish in an acceptable time. The BFS benchmark is level-synchronized [32], so termination detection is performed for every vertex level.

6. CONCLUSION

We have presented a programming model, Active Pebbles, designed for the direct expression of highly irregular applications at their "natural" granularities. The key elements of the model are pebbles sent between very fine-grained objects: targets. These pebbles trigger actions on the receiving targets, as in a model such as Charm++, but allow for finer object granularities. For example, sending messages to a target does not require any explicit bookkeeping. Our model allows high-performance, performance-portable, and intuitive high-level expression of fine-grained algorithms such as graph traversals. We implement a corresponding execution model that effectively converts fine-grained, point-to-point communications into optimized collective operations using five main techniques: fine-grained target addressing, message coalescing, active routing, message reductions, and configurable termination detection. These techniques combine to allow fine-grained algorithms, expressed at their natural granularities, to perform as well as more complicated, MPI-based implementations.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported by a grants from the Lilly Foundation and Intel Corporation, NSF grant CNS-0834722, and DOE FASTOS II (LAB 07-23). The Odin system was funded by NSF grant EIA-0202048. Portions of this work were presented as a research poster in [30].

8. REFERENCES

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. *J. of Par. and Dist. Comp.*, 44(1):71–79, 1995.

[2] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Intl. Conference on Parallel Processing*, pages 523–530, August 2006.

[3] D. A. Bader, D. R. Helman, and J. JáJá. Practical parallel algorithms for personalized communication and integer sorting. *Journal of Experimental Algorithmics*, 1, 1996.

[4] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Intl. Parallel and Distributed Processing Symposium*, Mar. 2007.

[5] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1994.

[6] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, April 2004.

[7] P. Charles, C. Grothoff, V. A. Saraswat, et al. X10: An object-oriented approach to non-uniform cluster computing. In *Obj. Oriented Prog., Sys., Lang., and Apps.*, 2005.

[8] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[9] P. Erdős and A. Rényi. On random graphs. *Publ. Math. Debrecen*, 6:290–297, 1959.

[10] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A study of a new parallel computation model. In *IPDPS*, pages 1–6, 2007.

[11] R. Garg and Y. Sabharwal. Software routing and aggregation of messages to optimize the performance of HPCC RandomAccess benchmark. In *Supercomputing*, pages 109–, 2006.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[13] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Principles of Distributed Computing*, pages 276–290, 1988.

[14] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[15] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Supercomputing*, Nov. 2007.

[16] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Principles and Practice of Parallel Programming*, pages 159–168, Jan. 2010.

[17] L. V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Notices*, 28(10):91–108, 1993.

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

[19] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, et al. The HPC Challenge (HPCC) benchmark suite. In *Supercomputing*, pages 213–, 2006.

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

[21] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, Sept. 2009.

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

[23] S. Plimpton, R. Brightwell, C. Vaughan, and K. Underwood. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *Intl. Conf. on Cluster Comp.*, pages 1–7, 2006.

[24] A. L. Rosenberg and L. S. Heath. *Graph Separators, with Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[25] N. Shavit and D. Touitou. Software transactional memory. In *Principles of Dist. Computing*, pages 204–213, 1995.

[26] 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.

[27] UPC Consortium. UPC Language Specifications, v1.2. Technical report, Lawrence Berkeley National Laboratory, 2005. LBNL-59208.

[28] L. G. Valiant. A bridging model for parallel computation. *Communications of the 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 *Intl. Symposium on Computer Architecture*, pages 256–266, 1992.

[30] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. Active Pebbles: A programming model for highly parallel fine-grained data-driven computations. In *Principles and Practice of Parallel Programming*, Feb. 2011. Poster.

[31] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. AM++: A generalized active message framework. In *Par. Arch. and Comp. Tech.*, 2010.

[32] A. Yoo, E. Chow, K. Henderson, et al. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Supercomputing*, pages 25–. IEEE, 2005.