Expressing Graph Algorithms Using Generalized Active Messages

Nicholas Edmonds
Indiana University
Bloomington, IN 47405
ngedmond@cs.indiana.edu

Jeremiah Willcock
Indiana University
Bloomington, IN 47405
jewillco@cs.indiana.edu

Andrew Lumsdaine
Indiana University
Bloomington, IN 47405
lums@cs.indiana.edu

ABSTRACT

Recently, graph computation has emerged as an important class of high-performance computing application whose characteristics differ markedly from those of traditional, compute-bound kernels. Libraries such as BLAS, LAPACK, and others have been successful in codifying best practices in numerical computing. The data-driven nature of graph applications necessitates a more complex application stack incorporating runtime optimization. In this paper, we present a method of phrasing graph algorithms as collections of asynchronous, concurrently executing, concise code fragments which may be invoked both locally and in remote address spaces. A runtime layer performs a number of dynamic optimizations, including message coalescing, message combining, and software routing. We identify a number of common patterns in these algorithms, and explore how this programming model can express those patterns. Algorithmic transformations are discussed which expose asynchrony that can be leveraged by the runtime to improve performance and reduce resource utilization. Practical implementations and performance results are provided for a number of representative algorithms.

Categories and Subject Descriptors

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

General Terms

Performance, Design

1 Introduction

A growing and increasingly diverse group of scientific disciplines including bioinformatics, social network analysis, and data mining are beginning to utilize graph analytics at HPC scales. However, graph problems have a number of inherent characteristics that distinguish them from traditional scientific applications [29]. Graph computations are often completely *data-driven*: they are dictated by the vertex and edge structure of the graph rather than being expressed directly in code. Execution paths and data locations are therefore highly unpredictable. Moreover, the connectivity of many graphs is not determined by 3D space (as is the case for discretized partial differential equations), resulting in data dependencies and computations with *poor locality*. Finding high quality partitions in such situations is computationally impractical since no good separators may exist [18,27]. The resulting graph distributions have a high ratio of surface area to volume which can significantly limit scalability. Finally, graph algorithms are often based on exploring the structure of a graph rather than performing large numbers of computations on the graph data, which results in *fine-grained data accesses* and a *high ratio of data accesses to computation*. Latency costs (memory accesses as well as communication) can dominate such computations.

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 communicating sequential processes or bulk-synchronous-parallel [45] computation and communication pattern (typically with two-sided message passing) to effect the overall computation. Although this approach has been tremendously successful for scientific applications based on discretized partial differential equations, it is not well-suited for graph-based data-intensive applications [29]. To address these issues, we have developed an approach for portably expressing high-performance graph algorithms based on fine-grained generalized active messages, as provided by the Active Pebbles programming model [49].

As traditionally defined, an active message [46] is a message that is sent explicitly but not received explicitly: a previously registered handler is called automatically when an active message arrives. The Active Pebbles model adds to this the ability to send messages at granularities as small as single words, with the runtime system combining messages as appropriate for particular platforms; message handlers are also able to send new messages in unlimited ways, with termination detected automatically. The ability to send and process messages asynchronously enables a high degree of latency hiding and a large number of messages concurrently in flight, giving several performance benefits on general-purpose, distributed-memory platforms.

Formulating graph algorithms using active messages has the dual advantages of being both conceptually simple while not over-constraining implementations. To convert a sequential algorithm expressed in an iterative form using loops, the loop bodies become message bodies. Rather than the coarse-grained, implicit dependencies expressed by the ordering of loop iterations, the active message form of the algorithm expresses the minimal set of dependencies necessary for correctness by allowing message handlers to send other messages directly. This method of expressing dependencies allows the critical path of the application to be efficiently captured and executed without artificially extending it through coarse-grained programming constructs. Active messages allow the user to directly modify algorithm code in an understandable fashion (individual vertex and edge operations) rather than requiring the use of complicated, vendor-tuned primitives which operate at a coarser level.

With this approach, we are able to capture the fine-grained dependency structure of graph computations at runtime, exposing maximal parallelism. While this fine-grained expression may not be directly suited to certain hardware (e.g., due to message injection rate limits for network adapters), the active message model allows the runtime system to adapt the algorithms to the hardware using coalescing and other transformations. Deferring these decisions until runtime is especially appropriate for graph algorithms as the structure of the graph determines the computational structure of the algorithm and thus both are discovered dynamically.

Finally, the active message phrasing permits both shared- and distributed-memory parallelism, and is amenable to acceleration as well. Separating the expression of an algorithm (the code in the active messages) from the implementation (the messages themselves) enables performance portability across a variety of platforms without having to change the algorithm specification. A single algorithm specification may be executed using an active message library [6, 26, 49] in a distributed-memory environment, a threading library [34,47] possibly in combination with work-stealing [5] in a shared memory environment, and hardware-specific programming environments [1, 41] targeting various kinds of accelerators. Most importantly, arbitrary combinations of these hardware environments are also supported.

2 Related Work

This paper builds on previous work which described a programming and execution model that allows specification and implementation of data-driven applications to be separated [48]. Here we make the case for fine-grained active messages as a suitable programming abstraction for graph applications and discuss the algorithmic refinements and runtime optimization techniques necessary to make this phrasing efficient. Before we attempt to justify the use of a new abstraction and programming model for graph applications, it is first appropriate to examine existing abstractions and models, as well as past attempts to use active messages to implement graph algorithms. We refer readers to [49] for a discussion of related work on active messages.

2.1 Parallel Graph Libraries

A number of libraries targeted at parallel graph computation exist which utilize a variety of abstractions. The original version of the Parallel Boost Graph Library (Parallel BGL) [22] utilized a coarse-grained Bulk Synchronous Parallel [45] programming model and targeted only distributed-memory parallelism. Later extensions added some ad hoc forms of active messaging, though the performance of these approaches is limited by the underlying communication layer.

The Graph Algorithm and Pattern Discovery Toolbox (GAPDT, later renamed KDT) [21] provides both combinatorial and numerical tools to manipulate large graphs interactively. KDT runs in parallel over Star-P [38] and targets distributed memory parallelism. KDT focuses on algorithms though the underlying sparse matrix kernels are also exposed. Later versions of KDT use the Combinatorial BLAS [8] as the computational back-end. The Combinatorial BLAS also uses linear algebraic primitives to perform graph computation and incorporates two-dimensional data decompositions. We view linear algebraic kernels as complementary abstractions to our visitor-based approach, which could be joined in the future to allow users the flexibility of using both.

The MultiThreaded Graph Library (MTGL) [4] and Small-world Network Analysis and Partitioning (SNAP) [3] are shared-memory libraries for manipulating graphs and are thus limited to the resources available on a single symmetric multiprocessor. The MTGL uses loop-level parallelism and complicated dependency analysis to generate efficient code for the Cray XMT and, via virtualization, commodity SMPs. Experimental results indicate that out-of-core approaches are incapable of matching the performance of distributed-memory implementations [2] meaning that implementations capable of leveraging both shared and distributed memory parallelism are essential for graph computation at large scale.

2.2 Active Messages for Graph Algorithms

The Active Pebbles programming and execution models are based on the identification of a common set of techniques for expressing and executing data-driven problems [49]; graph algorithms are one of the specific motivating examples for that work. The implementations of graph algorithms presented here utilize AM++ [48], a library implementing the Active Pebbles model. Further details on Active Pebbles are given in Section 3.1.

STAPL is a generic parallel library for graph and other data structures and algorithms, built upon the ARMI active-message runtime system [43]. ARMI supports automatic message coalescing to coarsen message granularity for performance, but does not provide routing or message reductions natively. STAPL's parallel graph class pGraph is based on the sequential Boost Graph Library, and some of its algorithms use nested active messages to create distributed-memory parallelism, but details are not provided [42].

PGAS frameworks have also identified and are addressing limitations in their handling of graph algorithms. For example, Jose et al. introduced UPC Queues, a mechanism for faster producer-consumer communication in Unified Parallel C [24]. The goal of this work is to improve UPC's performance on the Graph 500 benchmark (breadth-first search).

The Chapel [9] language provides direct support for asynchronous active messages using the ***on ... do begin*** idiom [14], as well as native atomic blocks. This model inspired one of the intermediate languages described in Section 5, and (if combined with an appropriate execution model) would be a suitable abstraction for expressing graph algorithms using the approach described in this paper. The X10 language [10] has similar constructs using ***async*** and ***at*** keywords for sending active messages. That language has been used to create the ScaleGraph [13] graph library; their implementation techniques are not described.

3 Programming Model

Programming models which are well suited to traditional HPC applications depend heavily on the locality inherent in these applications, in particular, each node communicating with only a few local peers. Coarse-grained approaches based on the BSP "compute-communicate" model thus tend to yield scalable solutions on current system scales.

In contrast, graph applications possess no underlying natural locality which can be determined analytically. The locality information is **irregular**, and embedded directly in the data itself. This locality information describes a dependency graph for computations which is **non-local**, i.e., it does not have good separators [37]. To complicate matters further, graph problems tend to be **fine-grained**, i.e., they possess a large number of small objects. Performing efficient static coalescing of these objects into larger, coarser-grained objects is hampered by the irregularity of the problems. Rather than coarse-grained static (or compile-time) approaches, we apply the Active Pebbles programming and execution model, specifically designed for such fine-grained applications. This execution model performs transformations at runtime to efficiently map applications expressed in the programming model to the underlying machine in an architecture-aware fashion.

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 [31]. However, graph applications need shared access to data structures which naturally cross address spaces. A number of choices exist for how to implement fine-grained, irregular remote memory access. The key requirement with regard to graph applications is that the remote memory updates performed by one process must be atomic with regard to those performed by other processes and must support "read-modify-write" operations (e.g., compare-and-swap, fetch-and-add, etc.). More importantly, only the process performing the updates has knowledge of which regions of memory are being updated and thus the process whose memory is the target of these updates cannot perform any sequencing or arbitration of the updates. Finally, some algorithms require dependent updates to multiple, non-contiguous locations in memory, which provides perhaps the greatest challenge to a programming model.

MPI-2 one-sided operations [31, ch. 11] and Partitioned Global Address Space (PGAS) models [33, 44] attempt to fill the gap left by two-sided MPI message passing 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 in current versions or do not perform well. Thus, we claim that these approaches do not provide the appropriate primitives for fine-grained graph applications. The designers of those approaches have also realized these limitations in the original models, leading to proposals such as UPC queues [24] and Global Futures [11] to add more sophisticated primitive operations (including active messages in some cases) to otherwise PGAS programming models. Additions to one-sided operations in MPI-3 [32, ch. 11] such as *MPI_Fetch_and_op()* and *MPI_Compare_and_swap()* may also rectify some of the shortcomings of the MPI one-sided communication model, though high performance implementations were unavailable when this work was conducted.

3.1 Background: Active Pebbles

Because of its specialization for the kinds of fine-grained applications that we consider, we use the Active Pebbles programming and execution model [49] as the underlying framework for our work. For completeness, we briefly summarize the model here. The Active Pebbles programming model takes traditional active messages [46] and generalizes them in a number of important ways, as well as providing an implementation designed to handle large numbers of tiny messages (both in data size and in computation) efficiently. The most important of the programming model generalizations is that Active Pebbles allows message handlers to directly send additional messages, to unbounded depth. In many cases this feature removes the need for application-level message buffering.

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 user-provided data distribution object. Pebbles are unordered (other than by termination detection), allowing flexibility in processing (such as threading or other forms of acceleration).

The termination detection features of Active Pebbles allow quiescence to be detected. In Active Pebbles, all messages must be sent within defined *epochs* (similar to the same concept in MPI-2 one-sided communication), and all messages globally are guaranteed to have been sent and handled—including those messages sent during handlers invoked by previous ones in the epoch—by the time the epoch completes. Active Pebbles provides runtime message coalescing to reduce network injection rate and increase bandwidth utilization, as well as software routing to reduce link and message buffer requirements in large networks. Finally, user-specified message reductions allow automatic message combining and duplicate message elimination, which is especially powerful in combination with routing and coalescing as will be seen in Section 5.

Active Pebbles allows graph applications to be expressed at their natural levels of granularity (e.g., individual vertex and edge operations) and then performs transformations at runtime to yield high performance. This programming model thus yields the expressiveness and programmability needed by a user-level library while providing the performance of hand-tuned code.

3.2 Message Configuration

Message information in the Active Pebbles model includes the data type to be sent, the C++ type of the message handler, and policies for coalescing, routing, and reductions. While the data type and message handler are algorithm-specific, other aspects are likely to be tied to hardware architecture and system configuration and possibly run-time information about the input data. Thus, we have extended Active Pebbles with a message configuration layer that separates the concerns of message semantics vs. configuration and run-time information.

The configuration layer allows algorithm users to create message configuration objects; each of these objects contains a desired coalescing approach, buffer size, and routing topology. The object also identifies whether the user would like message reductions to be used when applicable to a particular algorithm (and if so, which cache type and parameters to use). This configuration mechanism separates the specification of an algorithm's message semantics from the mapping of those semantics to a particular implementation. Thus optimizations can be applied to algorithm specifications by end users without modifying the algorithm specifications themselves. This mechanism allows algorithms to utilize optimization techniques that were unknown at the time the algorithm was developed, as long as the algorithm's implementation (and semantics) support that type of optimization. This retroactive optimization is key to supporting performance portability across current and future architectures without the need to reimplement algorithms.

4 Graph Algorithm Patterns

Identifying commonly used kernels or patterns has long been recognized as a useful method for enabling portability and high performance. A single common interface gives end users a target to program to, and system vendors a target to optimize for. Furthermore, a single widely-used implementation is more likely to be correct and efficient. The Basic Linear Algebra Subroutines (BLAS) [28] are perhaps the canonical example of such a library. Linear algebraic graph primitives have been proposed in [7]; we have observed higher-level patterns in visitor- or traversal-based algorithms as well. These patterns are representative of a significant fraction of existing graph algorithms (though by no means all of them). We examine how these patterns are implemented using Active Pebbles and possible implementations in other programming models.

4.1 Wavefront Expansion (Label-Setting)

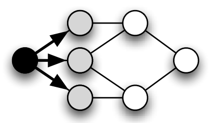
Perhaps the simplest traversal-based pattern is the exploration of a graph from a single source vertex. As depth-first searches are inherently sequential [36], the most basic *parallelizable* search is breadth-first (BFS). The wavefront in this case represents the active vertices at each level in the search. This traversal pattern is *label-setting* in the sense that once a vertex is visited and a label (distance, predecessor, etc.) assigned it will not be revisited or have this label changed.

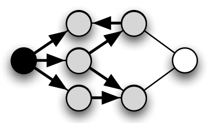
In order to achieve breadth-first exploration all vertices at distance i from the source vertex must be processed before any vertices at distance $i+1$. Messages in Active Pebbles are sent immediately with no ordering imposed between messages in the same epoch so this behavior requires one epoch per level, where level i consists of all vertices which have distance i from the source.

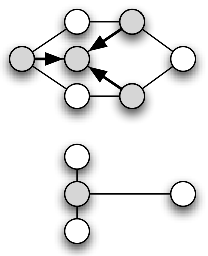
Alg. 1: Basic algorithm for breadth-first search.

```
     In  : Graph $\mathcal{G} = \langle V, E \rangle$, vertex $s$
     Out : $\forall v \in V$: $distance[v]$ = distance of $v$ from $s$
1    $Q \leftarrow nextQ \leftarrow$ empty queue;
2    $level \leftarrow 0$;
3    $distance[v] \leftarrow \infty, \forall v \in V$;
4    $distance[s] \leftarrow 0$;
5    enqueue $s \rightarrow Q$;
6    while $Q$ not empty do
7        $level \leftarrow level + 1$;
8        foreach $v \in Q$ do
9            foreach neighbor $w$ of $v$ do
10               if $distance[w] = \infty$ then
11                   $distance[w] \leftarrow level$;
12                   enqueue $w \rightarrow nextQ$;
13       $Q \leftarrow nextQ$;
14       $nextQ \leftarrow$ empty queue;
```

Algorithm 1 shows a breadth-first search implementation which uses two queues to force level-wise exploration. We have hidden the fact that in the distributed memory case each logical queue is actually composed of one local queue per process containing vertices from the set assigned to that process. Line 6 tests whether all local queues are non-empty, while line 8 iterates over the local queue on each process. To implement this algorithm using active messages (Algorithm 2), lines 10–12 become


(a) Wavefront


(b) Wavefront (wide)


(c) Coarsening and Refinement

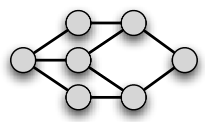

(d) Iterative

Figure 1: Graph algorithm patterns.

Alg. 2: Parallel active message algorithm for BFS.

```
In  : Graph $\mathcal{G} = \langle V, E \rangle$, vertex $s$,
      $\forall\, v \in V$: $owner[v]$ = rank that owns $v$
Out : $\forall\, v \in V$: $distance[v]$ = distance of $v$ from $s$
1  $Q \leftarrow nextQ \leftarrow$ empty queue;
2  $level \leftarrow 0$;
3  $distance[v] \leftarrow \infty, \forall v \in V$;
4  $distance[s] \leftarrow 0$;
5  enqueue $s \rightarrow Q$;
6  message handler explore(Vertex $v$)
7      if $distance[v] = \infty$ then
8          $distance[v] \leftarrow level$;
9          enqueue $v \rightarrow nextQ$;
10 while $Q$ not empty do
11     $level \leftarrow level + 1$;
12     active message epoch
13         parallel foreach $v \in Q$ do
14             parallel foreach neighbor $w$ of $v$ do
15                 send explore($w$) to $owner(w)$;
16     $Q \leftarrow nextQ$;
17     $nextQ \leftarrow$ empty queue;
```

an active message that inserts w into the local queue at its target, removing the need for communication inside the distributed queue data structure itself. The messaging layer ensures that each *enqueue()* operation executes on the owning process of the vertex being enqueued. If we make the *enqueue()* method of Q thread-safe and add a critical section around lines 10–12, this implementation is also thread-safe and the **foreach** statements on lines 8 and 9 can be executed in parallel.

An implementation using MPI collectives would also use one epoch per iteration of the loop on line 6, but rather than communicating immediately on lines 10–12 it would buffer all remote *enqueue()* operations locally and send them at the conclusion of the loop using an *MPI_Alltoall()* operation (to communicate the number of operations) followed by an *MPI_Alltoallv()* operation to send the data. This approach fails to effectively overlap communication and computation in addition to being more complex than the active message formulation. Non-blocking collectives [23] and asynchronous point-to-point methods fail to alleviate this problem since we cannot begin the next BFS level until the previous one is complete.

Implementing a distributed queue of the sort described above using MPI-2 one-sided communications or the RDMA semantics provided by a number of PGAS languages is complicated by the absence of high-performance atomic operations such as compare and swap or fetch and add. In the absence of these operations, each process must allocate separate input buffers for each other remote process to write into. These buffers must be large enough to contain all remote updates and, if this cannot be guaranteed, logic must be provided to ensure all remote updates are received, possibly by using multiple epochs per level (with its attendant extra synchronization). At the conclusion of the BFS level, each process must apply the updates from all other processes and clear the input buffers. This approach either requires large amounts of additional memory to be allocated—to ensure that updates fit in the input buffers—or requires additional synchronization to perform multiple rounds of remote update exchanges per level.

4.2 Wavefront Expansion (Label-Correcting)

While breadth-first search is a simple and concise graph algorithm to use as an example, the strengths of the active message abstraction are more apparent in algorithms with more complex dependency structures. The label-correcting wavefront expansion pattern differs from the label-setting one in one important way. Instead of exploring vertices and edges once and labeling them, label-correcting wavefront algorithms may recompute and correct vertex and edge labels multiple times. Although this approach may be less work-efficient than a label-setting algorithm, it often exposes much more parallelism by allowing updates that are "usually independent" to proceed in parallel with exceptions repaired later.

Some label-correcting algorithms also use "thick" wavefronts: instead of only a single layer (i.e., frontier) of vertices being active at a time, vertices within a given "distance" (for an algorithm-specific definition of distance) are active. The thickness of the wavefront is controlled in an algorithm-specific manner, such as by using a parameter δ to define the thickness of the wavefront. Thicker wavefronts potentially expose more work at a given time and are thus more parallelizable, while thinner wavefronts are likely to be more work efficient. Figure 1(b) illustrates how a single level of the wavefront computation may contain paths of differing lengths in the graph, and thus computational dependencies within the wavefront. The canonical examples of label-correcting wavefront expansion are parallel single-source shortest path (SSSP) algorithms [12,30].

Rather than each level of the wavefront containing a set of independent vertices with a common label as in the label-setting pattern, work is now grouped into sets of vertices with "tentative" labels in the range $[c\delta, (c+1)\delta)$ where c is the current level. Additionally, these vertices may now have dependencies between them (caused by "light" edges with weight less than δ, using the terminology from [30]). The fine-grained structure of these intra-level dependencies provides additional opportunities for parallelism, provided they can be captured efficiently. Using active messages, dependencies between vertices u, v, and w of the form $u \rightarrow v \rightarrow w$ can be handled directly by having vertex u send a message to vertex v which then performs the appropriate computation and sends a message to vertex w. Using MPI collectives, each stage in the dependency chain would require a separate *MPI_Alltoall()*/*MPI_Alltoallv()* pair leading to unnecessarily frequent synchronization and $\mathcal{O}(p \log p)$ overhead for each all-to-all communication amongst the p processors. As shown in the message diagrams in Figure 2, the BSP-style algorithm communicates in bursts at the end of a computation step (Figure 2(b)) while the active-message-based algorithm sends messages as coalescing buffers fill up, even if computation has not yet finished (Figure 2(b)). Modifying the BSP algorithm to communicate data early only partially solves this problem as messages must still be stored on the receiver until the conclusion of the epoch when they are processed. This approach more effectively hides the communication latency, but merely shifts the burden of additional memory utilization from the sender to the receiver.

4.3 Parallel Search

Both of the wavefront expansion patterns above use bounded descent (or lookahead) to maintain reasonable work efficiency. This limits the available parallelism by ensuring that only solutions reasonably close to the current optimal solution are explored. More work-efficient solutions synchronize more frequently, while infrequently-synchronizing approaches are less work-efficient but expose more parallelism. One pathological case of balancing work efficiency and parallelism is to discard the notion of work efficiency entirely. The parallel search pattern does this by phrasing the entire computation as a single epoch with unconstrained parallelism. This pattern is equivalent to the label-correcting wavefront pattern where $\delta = \infty$.

Reachability, computing the set of vertices reachable from one or more sources, is an example of a parallel search kernel that is straightforward to express using active messages. A traditional implementation using MPI collectives would most likely utilize the same structure as the breadth-first search example above. This would require a number of epochs equal to the distance of the farthest reachable vertex from the source. Using generalized active messages with message sends allowed inside message handlers, reachability can be performed in a single epoch, as shown in Algorithm 3.

Alg. 3: Reachability determination – Parallel Search pattern.

```
Input: Graph $\mathcal{G} = \langle V, E \rangle$, vertex $s$
Output: $\forall\, v \in V$: $visited[v] = 1$ iff $v$ reachable from $s$
message handler explore(Vertex $v$)
    if $visited[v] = 0$ then
        $visited[v] \leftarrow 1$;
        parallel foreach neighbor $w$ of $v$ do
            send explore($w$) to $owner(w)$;
procedure main()
    active message epoch
        if $owner[s]$ = this rank then
            run handler for explore($s$);
```

4.4 Coarsening and Refinement

The coarsening and refinement pattern differs from search or wavefront patterns because rather than starting with a single source vertex and discovering additional parallelism, algorithms based on this pattern start with a logical work list containing a set of vertices or edges which can be processed in parallel. Vertices and edges are contracted to representative "supervertices" shrinking the work list until it is empty and the algorithm is complete. As the working set of the algorithm becomes smaller, parallelism is reduced and contention increases as each active supervertex represents a larger set of vertices and edges. Figure 1(c) shows an example of this process. The connected components algorithm due to Shiloach and Vishkin is one classic example of this pattern [39].

The Shiloach-Vishkin connected components algorithm, shown in Algorithm 4, consists of two phases. The first phase, *hooking* (lines 20–24), combines trees if there is an edge between them. The second phase, *shortcutting* (lines 25–31), contracts or flattens trees to a *root* or representative vertex. Component membership is tracked using a *parent* array in which $parent[u] = parent[v]$ iff u and v are in the same component. Hooking must be performed carefully to avoid cycles and requires the ability to scan all the adjacent vertices of each

Alg. 4: Connected components algorithm using active messages.

```
   Input: Graph $\mathcal{G} = \langle V, E \rangle$, vertex $s$
   Output: $\forall\, v \in V$: $parent[v]$ = the component containing $v$
1  $hooked \leftarrow doubled \leftarrow false$;
2  message handler hook(Vertex $pv$, Vertex $u$)
3      $pu \leftarrow parent[u]$;
4      if $owner(pu)$ = this rank then
5          $ppu \leftarrow parent[pu]$;
6          if $pv < ppu$ then
7              $hooked \leftarrow true$;
8              $parent[pu] \leftarrow pv$;
9      else
10         send hook($pv$, $pu$) to $owner(pu)$;
11 message handler pointer-double(Vertex $pv$, Vertex $v$)
12     send pointer-double-reply($v$, $parent[v]$) to $owner(v)$;
13 message handler pointer-double-reply(Vertex $v$, Vertex $ppv$)
14     if $ppv < parent[v]$ then
15         $doubled \leftarrow true$;
16         $parent[v] \leftarrow ppv$;
17 procedure main()
18     $parent[v] \leftarrow v, \forall v \in V$;
19     do
20         $hooked \leftarrow false$;
21         active message epoch
22             foreach $v \in V$ do
23                 foreach neighbor $u$ of $v$ do
24                     send hook($parent[v]$, $u$) to $owner(u)$;
25         do
26             $doubled \leftarrow false$;
27             active message epoch
28                 foreach $v \in V$ do
29                     $pv \leftarrow parent[v]$;
30                     send pointer-double($pv$, $v$) to $owner(pv)$;
31         while $doubled$ on any node;
32     while $hooked$ on any node;
```

component and access those vertices' *parent* values. In a BSP model, *MPI_Alltoall()*/*MPI_Alltoallv()* can be used to exchange *parent* values, but two stages of communication (and thus synchronization) may be required when u, v, and $parent[v]$ are all on different nodes. Shortcutting also requires two *MPI_Alltoall()*/*MPI_Alltoallv()* pairs to perform the operation $\{\forall v \in V : parent[v] \leftarrow parent[parent[v]]\}$ (one to send a request for the parent of each $parent[v]$ value, then a reply to that request).

The active message version of this algorithm avoids copying values in the *parent* array between processors and thus uses significantly less storage, in addition to requiring less synchronization. The multiple levels of messages can share a single epoch in the active message version, without extra synchronization between the stages. The outer loops in the code use reductions (sums) to determine globally whether parents were changed and whether any hooks occurred; this step is done using the end-of-epoch sum capability in Active Pebbles.

When hooking two vertices v and u connected by an edge $v \leftrightarrow u$, up to three processes may be involved: the owners of v, u, and $parent[v]$, since hooking requires sending $parent[v]$ to $owner(parent[u])$. The owner of v fills in the value of $parent[v]$ (line 29) and sends a message to $owner(u)$ to fill in the value of $parent[u]$ (line 3). The values of $parent[u]$ and $parent[v]$ must then be sent to $owner(parent[v])$ (line 10). Shortcuts are applied for local operations. For hooking to terminate, a monotonic ordering must be defined on vertices to ensure that only

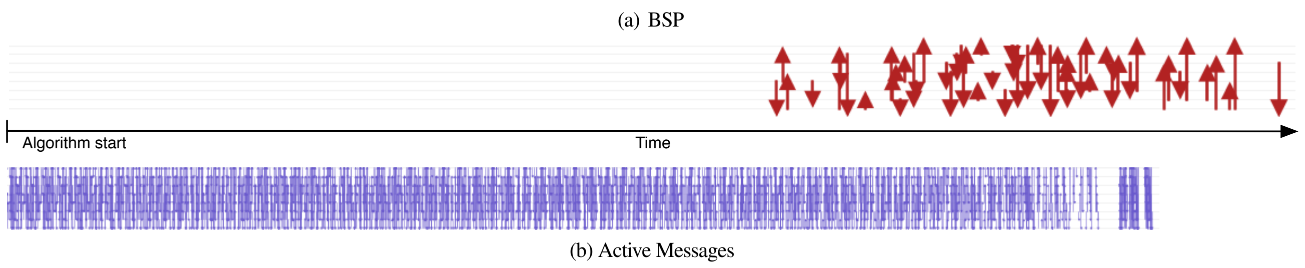


Figure 2: Message trace of (a portion of) a breadth-first search algorithm in BSP and Active Message styles. Process ranks are represented on the Y axis. Each vertical arrow represents a message; arrow size is proportional to message size.

one update will ultimately succeed for each target vertex; it can also be used to eliminate redundant updates to the same parent. Though hooking can generate message chains of length 2, the messages can be interleaved arbitrarily and only a single epoch is required.

The shortcutting stage involves each process performing pointer-doubling of the form $\{\forall v \in V : parent[v] \leftarrow parent[parent[v]]\}$ using a two-stage process, all executed in one epoch. The owner of v first sends a request to perform $parent[v] \leftarrow parent[parent[v]]$ to the owner of $parent[v]$, giving it the value of v for the reply (line 30). The owner of $parent[v]$ then fills in $parent[parent[v]]$ and sends a message to perform the update back to the owner of v (line 12), which then processes it (line 16).

4.5 Iterative Methods

Iterative methods such as PageRank [35] or graph layout algorithms [20, 25] often iterate computations with the same structure over all the vertices and edges in the graph until a fixpoint is reached (i.e., no changes occur for a round of updates). These algorithms yield consistent parallelism as the work available remains constant (Figure 1(d)). Parallelization is straightforward using halo-zone communication with MPI or techniques similar to those described for coarsening and refinement algorithms above using active messages.

4.6 Combinations of These Patterns

What makes these algorithm patterns useful is that they occur repeatedly in higher-order algorithms. For example, the strongly connected components of a directed graph can be found by finding the intersection of a pair of breadth-first search trees [19]. Betweenness centrality can be parallelized using a label-correcting SSSP algorithm combined with a pair of topological-order traversals [16]. Optimizing the execution of these patterns, which may be used as kernels of more complex algorithms, is one important part of achieving high performance. Another important element in this goal is scheduling the kernels themselves so as to avoid coarse-grained synchronization between them. Projects in numerical computing [40] have demonstrated the effectiveness of scheduling applications consisting of multiple, dependent algorithm kernels in a fine-grained fashion in order to balance computations and ensure less time is wasted waiting for global synchronization. "Fusing" graph kernels at a much finer granularity offers similar benefits with regard to computational balance while also offering temporal locality (frequently absent in graph computations). When active messages are used to express graph algorithms, kernel scheduling is handled automatically by the runtime layer. Kernel fusion can be accomplished by combining the message dependencies of the individual kernels. This makes the active message formulation extremely powerful from an optimization standpoint. Furthermore, these optimizations may be applied dynamically at runtime as the structure of the input data, and thus the computation, is discovered.

5 Converting Algorithms to Active Messages

Explicit use of active messages is not always a familiar form for expressing parallel graph algorithms. However, more standard representations (e.g., with PRAM-like semantics) can be converted to active message form with a straightforward translation process, which we describe below. Possible future work would be to automate some or all of this translation process, likely based on programmer annotations of what data structures are distributed and how, and what operations must be atomic, as well as to generate efficient thread-safe code for **atomic** blocks (even for larger ones that do not correspond to processor atomics).

The conversion process has three steps:

1. Identify distributed data structures and their data distributions.
2. Annotate where each program statement should execute, and when control flow and data could be moved between nodes.
3. Lift active message handlers out of the code, replacing control flow movements with sends of active messages.

Listing 1: Pseudo-PRAM version of breadth-first search.

```
push(Q, s);
while (!empty(Q)) {
  parfor (v ∈ Q, w ∈ neighbors(v))
    atomic
      if (test_and_set(color[w]) == 0)
        push(nextQ, w);
  swap(Q, nextQ);
  clear(nextQ);
}
```

Identifying Distributed Data Structures The conversion process starts with a program format we refer to as *pseudo-PRAM*. In this form, sequential vs. parallel loops are explicit, as well as atomic blocks, while the system is assumed to support shared memory. An example of breadth-first search in this form is given in Listing 1. The first step in converting it to run on a distributed-memory computer is to identify the data distributions of the shared variables; in the example, those are *Q*, *nextQ*, *neighbors*, and *color*. The other variables are scalar local variables and thus are not distributed. This formulation assumes a partitioning of vertices amongst processes with structural adjacencies (*neighbors*) and properties (*color*) for a vertex stored on the process owning that vertex. The method for mapping vertices to processes is the *owner* map. Given this information, we can then assign statements and expressions to owning processes.

Conversion to Remote Spawns Each statement must be executed on a process that has ownership of all expressions used in that statement; satisfying this requirement may require moving data around between subexpressions in a larger expression. The example is simpler: the primitive expressions that have data distributions are *neighbors(v)* (which must be run on *owner(v)*), *color[w]*, and *push(nextQ, w)* (the latter two of which must be run on *owner(w)*). Lifting these constraints out to larger blocks of code, the entire ***atomic*** block must be executed on *owner(w)*, while the second loop in the ***parfor*** statement must be executed on *owner(v)*—that is guaranteed by the distribution of *Q*. In a system without distributed transactions, ***atomic*** blocks are unable to span multiple nodes, and thus the block must execute in a single process. Given information on its execution location, the ***atomic*** block must be wrapped in an ***async_spawn*** statement to move its execution

Listing 2: Remote-spawn version of breadth-first search.

```
push(Q, s);
while (!empty(Q)) {
  epoch
    parfor (v ∈ Q, w ∈ neighbors(v))
      async_spawn(owner(w))
        if (test_and_set(color[w]) == 0)
          push(nextQ, w);
  swap(Q, nextQ);
  clear(nextQ);
}
```

Listing 3: Explicit active message version of breadth-first search.

```
class update_handler {
  queue& nextQ; color_map& color;
  void handle(Vertex w) {
    if (test_and_set(color[w]) == 0)
      push(nextQ, w);
  }
};
register_message_type update_msg using
  type Vertex,
  handler update_handler(nextQ, color),
  owner_map owner;
push(Q, s);
while (!empty(Q)) {
  epoch
    parfor (v ∈ Q, w ∈ neighbors(v))
      send update_msg(w);
  swap(Q, nextQ);
  clear(nextQ);
}
```

to *owner(w)*. Additionally, an ***epoch*** block needs to be wrapped around the synchronization region in the code; here, that is the entire ***parfor*** loop. Those changes produce Listing 2.

Conversion to Explicit Active Messages Given an algorithm expressed using explicit ***async_spawn*** commands, conversion to a set of active message types is similar to closure conversion used to compile functional languages. For every body of an ***async_spawn***, going from inside to outside, identify its set of free variables. For the algorithm in Listing 2, this set contains *w*, *color*, and *nextQ*. Of these, *color* and *nextQ* are distributed data structures, and so references to them should be stored in the message handler; *w* is local and should be part of the active message data. Thus, the handler will be an object that contains references to the *nextQ* and *color* variables, and will accept active messages of type *Vertex* (the type of *w*). The destination of each message can be computed using the owner map *owner*. Conversion of ***async_spawn*** calls to active message sends produces the explicit active message version of breadth-first search in Listing 3. Further optimization can be enabled by declaratively specifying the semantics for message reductions.

6 Experimental Evaluation

We have applied the active message abstraction to the design the Parallel BGL "2.0" as a message-driven library of graph algorithms utilizing the Active Pebbles programming and execution model. The underlying communication infrastructure is provided by AM++, an implementation of the Active Pebbles model, tested using MPI as its underlying data-movement layer. We used Challenger, a 13.6 TF/s, 1 rack Blue Gene/P with 1024 compute nodes. Each node has 4 PowerPC 450 CPUs and 2 GiB of RAM. Our experiments used Version 1.0 Release 4.2 of the Blue Gene/P driver, IBM MPI, and `g++` 4.3.2 as the compiler (including as the back-end compiler for MPI).

6.1 Implementation Details

For our evaluation, we choose algorithms from each of the classes discussed in Section 4. As previously discussed, breadth-first search is an example of a label-setting wavefront algorithm (§ 4.1). For a label-correcting wavefront (§ 4.2), we chose the Δ-stepping single-source shortest paths algorithm [30]. The Shiloach-Vishkin connected components algorithm is an excellent example of a coarsening and refinement algorithm (§ 4.4). PageRank is a well-known iterative algorithm (§ 4.5) for ranking vertices in a graph. Where suitable, we have used the Graph 500 generator to generate synthetic graphs of suitable sizes for our weak-scaling experiments. The exception to this is the connected components experiments, for which Graph 500 graphs of the degree used in the other experiments would be likely to have a single large component and few other components beyond isolated vertices. Erdős-Rényi graphs have a more interesting component structure, especially around average degree 2 which is the hitting time of the giant component (e.g., the point at which the second largest component contains fewer than $\log n$ vertices with high probability, where n is the number of vertices in the graph). For this reason we have used Erdős-Rényi graphs of average degree 2 for our connected components experiments. All experiments use routing with a rook graph topology (an approximately-balanced rectangle of nodes where nodes in the same row and/or column are connected) to reduce the number of message buffers required for P processes from $\mathcal{O}(P)$ to $\mathcal{O}(\sqrt{P})$.

6.2 Results

The following experiments compare algorithms written in an active message (AM) style to the same algorithms written in a BSP [45] style. While AM graph algorithms have a number of advantages with regard to leveraging on-node parallelism, we compare only the single-threaded case in this work. To provide a fair comparison between the programming models rather than the efficiency of the underlying communication libraries, both the AM and BSP implementations use AM++. Previous work has demonstrated that AM++ has minimal latency and bandwidth overheads relative to the data transport layer it is implemented over (MPI in this case) [48]. Both the BSP and AM algorithms use the routing features of AM++, as well as message reduction (caching) in some cases. Note that in a traditional BSP implementation using MPI directly, adding routing or caching capabilities would greatly complicate the code. In the AM implementations, routing and caching are abstracted and encapsulated in the runtime. The primary distinction in the implementations is that while the active message algorithms overlap communication and computation aggressively and send messages from message handlers, the BSP algorithms defer communication (or at least the handling of communicated data) until the end of the epoch and handle dependent operations using algorithm-level queues to restrict each epoch to a single round of communication. It is likely that the BSP algorithms presented here perform *better* than versions which do not leverage the routing and message reduction features of AM++; however, disabling these features would increase memory consumption and make only small problems solvable using BSP algorithms.

Formulating graph algorithms using active messages allows increased overlap of communication and computation *provided algorithms expose sufficient asynchrony*. Active message graph algorithms also utilize less memory because once messages are executed they can be retired. BSP implementations may communicate data before the end of an epoch, but do not process that data and free the associated memory until the conclusion of the epoch. Our results demonstrate that AM algorithms reduce memory utilization enabling them to scale to larger numbers of nodes than similar algorithms implemented in a BSP style, which fail due to memory exhaustion. In the regions where both styles of algorithms execute successfully, AM algorithms are often more efficient due to their ability to more accurately capture the critical path of the application using fine-grained asynchronous operations. However, some graph algorithms have a coarse-grained structure that is well-suited to the BSP programming model. In these cases AM implementations are unable to outperform BSP implementations because insufficient asynchrony is available in the algorithm. For these algorithms we demonstrate that the overheads imposed by the AM implementations are limited, and represent a reasonable trade off for the benefits this model provides—reduced memory utilization, performance portability, and retroactive tuning of algorithm implementations without changing their specifications. In some cases, we describe algorithmic modifications which expose additional asynchrony and allow AM implementations to outperform their BSP counterparts.

The algorithms presented include a mixture of highly-synchronous algorithms (BFS and PageRank) well suited to BSP, and more asynchronous algorithms well suited to active messages (Δ-Stepping and several connected components variants). The core algorithm code is the same for the BSP versions and their AM counterparts. The desired level of granularity is achieved by specifying appropriate runtime parameters, not by modifying the algorithms themselves. In this fashion, fine-grained implementations can be converted to coarse-grained ones by the runtime system; however, the opposite conversion is non-trivial.

All of the runs have much better performance on a single node because of the lack of any messaging overheads and explicit checks for sends-to-self in AM++. Missing data values in the charts indicate that the algorithm failed to complete due to memory exhaustion.

6.2.1 *Breadth-First Search*

Breadth-first search (BFS) is a coarse-grained, BSP algorithm. Every vertex at level i must be processed before any vertex at level $i+1$ can be explored. This provides limited asynchrony for the AM implementation to leverage as shown in Figure 3. The runs annotated with "1M cache" use duplicate message elimination, implemented in this case with a 2^{20}-message cache per neighbor in the routing topology. With fewer than 32 nodes and no caching we see somewhat worse performance of the AM implementation vs. BSP. Enabling caching for both algorithms eliminates the overhead of the AM version for fewer than 32 nodes. The BSP implementations begin to fail due to memory exhaustion at 256 nodes but scale relatively well to 128 nodes. It is likely that memory exhaustion would occur much sooner due to increased communication buffer utilization were AM++ routing not in use here. The poor scaling performance beyond 32 nodes for the AM implementations is attributable to a number of factors. First, the static choice of coalescing factor (2^{10} messages) is well suited to small-scale execution but provides unreasonable overhead at larger scale. This demonstrates the need to dynamically vary the coalescing factor depending on the problem size. Second, we see that the scaling is worse for the AM implementation using message reductions than for the same algorithm without reductions. As the size of the graph increases, the fraction of data which is non-local increases proportionally. With a statically-sized message cache this means that the hit rate of the cache decreases. Making cache sizes dynamic, or even dynamically enabling/disabling caching, would allow maximal benefit to be derived from message reductions without introducing undesirable overheads. A suitably intelligent runtime would be capable of monitoring the system state and making these decisions appropriately without risking memory exhaustion. This algorithm demonstrates two important points:

1. Some algorithms lack sufficient asynchrony to benefit from AM implementations. However, features of the Active Pebbles model (e.g., message reductions) may still improve performance.
2. Because algorithm execution is dependent on the structure of the input graph, static runtime optimization parameters are suboptimal vs. varying parameters dynamically in response to system state.

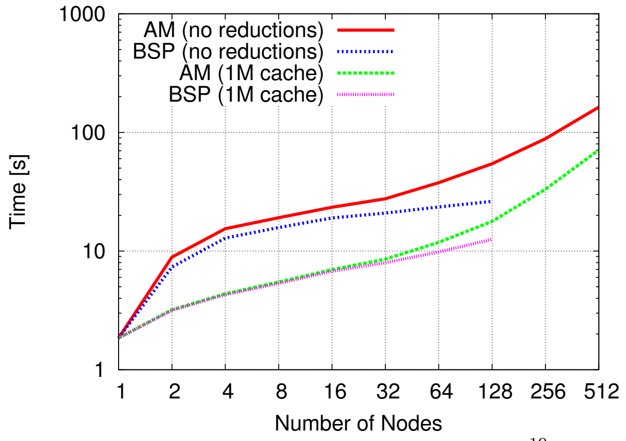


Figure 3: Breadth-first search weak scaling (Graph 500, 2^{19} vertices per node, average degree of 16, 2^{20}-message caches, average over 16 runs which are guaranteed to visit more than 100 vertices).

6.2.2 *Δ-Stepping Shortest Paths*

Figure 4 shows the performance of the active message and BSP implementations of Δ-stepping shortest paths. Δ-stepping allows all paths with weights in the range $(i \cdot \delta, (i+1) \cdot \delta]$ to be relaxed concurrently. This means that within a single step of the Δ-stepping algorithm there may be a path composed of multiple incident edges which need to be relaxed in sequence. The BSP implementation will require one epoch for each edge in the path to perform this relaxation due to the fact that the relaxation request for the first edge in the path is not processed until the conclusion of the epoch . This causes a second epoch to be initiated for the second edge in the path, and so on for each incident edge until the path distance is at least $(i+1) \cdot \delta$. The AM implementation requires only a single epoch to relax the entire path. The relaxation of the first edge sends a message to the target of that edge, which triggers relaxation of the second edge, and so on until the path length is at least $(i+1) \cdot \delta$. The termination detection feature of AM++ allows the runtime to wait until all dependent messages have been received before concluding the epoch.

The additional asynchrony in the Δ-stepping algorithm allows the AM implementations to outperform the BSP implementations in almost all cases. As with BFS, the BSP algorithms fail to complete due to memory exhaustion, in this case for more than 16 nodes. The fact that memory exhaustion occurs with fewer nodes than BFS is due to the fact that the Δ-stepping algorithm maintains an auxiliary data structure to order edge traversal which consumes significantly more memory than the BFS queue. In this implementation we do not remove a longer path to a vertex from this data structure when a shorter one is found because the AM algorithm supports multiple threads and this removal is difficult to perform in a manner that is both thread-safe and efficient. The increased memory requirements of the Δ-stepping algorithm require us to use a smaller graph and message reduction cache than in the BFS experiments.

As with BFS we see that message reductions improve performance at small scale but that the benefits decrease as the ratio of cache size to overall graph size shrinks. At larger node counts we see that the reduction caches are ineffectual but that because they are fast, direct-mapped caches their overhead does not adversely affect algorithm performance. The Δ-stepping algorithm also takes a δ parameter, the width of each bucket in vertex distances. The results shown use $\delta = 4$, which performed well across a range of input sizes. In addition to dynamically tuning the message coalescing factor and cache sizes to improve scalability, tuning δ would provide increased control over the amount of parallel work and asynchrony available vs. the likelihood that relaxing an edge contributed to the final solution of the algorithm (e.g., work efficiency). This trade off between work efficiency and parallelism is omnipresent in graph algorithms and careful control of it is the key to high performance.

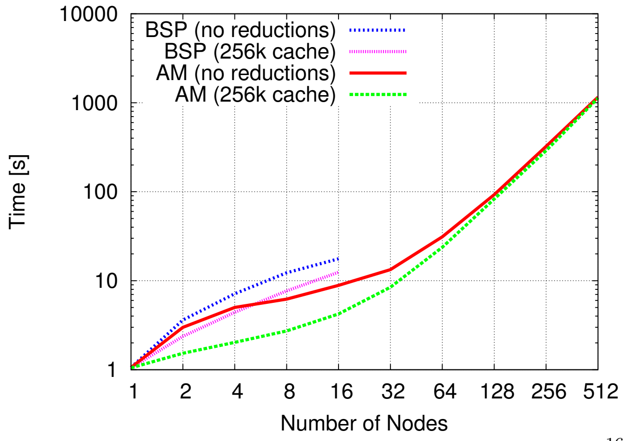


Figure 4: Δ-stepping shortest paths weak scaling (Graph 500, 2^{16} vertices per node, average degree 16, 2^{18}-element caches).

6.2.3 *Shiloach-Vishkin Connected Components*

The Shiloach-Vishkin connected components algorithm requires a number of hooking and contraction steps proportional to the diameter of the largest component to complete. Because we know that it is likely that the graphs used have a giant component, we also implement an optimized algorithm which does a parallel exploration from the highest degree vertex (which is likely to be in the giant component), and then applies the Shiloach-Vishkin connected components algorithm to the undiscovered portion of the graph to label the remaining components. We call this variant "Parallel Search + Shiloach-Vishkin" (PS+SV).

Figure 5 demonstrates that initializing the SV algorithm with a parallel search to discover the giant component ("PS + SV") provides significant performance improvement for both AM and BSP, and improves the scaling of the AM implementation. This transformation is an example of an algorithmic refinement that exposes asynchrony which can be effectively utilized by AM implementations. Message reductions again provide performance improvements in both cases, which diminish as the graph grows. It is likely that dynamically tuning the coalescing factor would mitigate the poor scaling of the the AM "PS + SV" algorithm between 128 and 512 nodes and result in a scalable algorithm capable of executing on larger numbers of nodes.

The "PS + SV" algorithm demonstrates the ability of the AM implementation to more efficiently leverage the available asynchrony in an algorithm, as well as the ability to perform minor modifications to an algorithm to make it more suitable for AM. Figure 5 demonstrates that the AM implementation of the classical Shiloach-Vishkin ("SV") algorithm outperforms the BSP version because hooking and pointer-doubling operations that involve multiple communication stages can be performed in a single AM epoch, while the BSP implementation requires multiple epochs. As with BFS and Δ-stepping, the BSP algorithms fail due to memory exhaustion, this time between 4 and 8 nodes.

6.2.4 *PageRank*

The PageRank algorithm is a highly synchronous graph algorithm well suited to BSP execution. This algorithm is implemented using a variant of the Parallel BGL's distributed property map abstraction [15], a PGAS-like data movement layer. Neither implementation uses reductions in this case because they have yet to be integrated into the distributed property map. Figure 6 shows that once again the BSP form of the algorithm fails to complete at larger node counts (at least 16 in this case), while the AM implementation runs successfully up to 512 nodes. In the range where both execute successfully, the AM implementation has comparable performance to the BSP version.

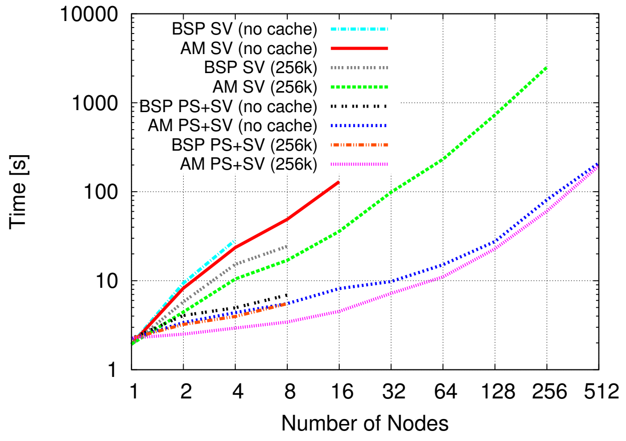


Figure 5: Shiloach-Vishkin connected components weak scaling (Erdős-Rényi, 2^{18} vertices/node, avg. degree 2, 2^{18}-element caches).[1]

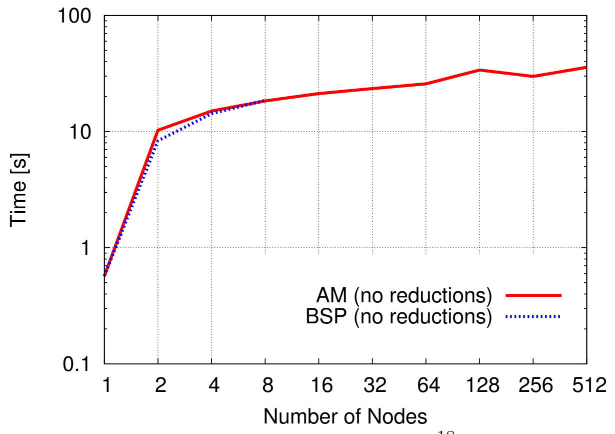


Figure 6: PageRank weak scaling (Graph 500, 2^{18} vertices per node, average degree 16, average of 20 iterations).

7 Conclusion

Phrasing graph algorithms as collections of asynchronous, concurrently executing, message-driven fragments of code allows for natural expression of algorithms, flexible implementations leveraging various forms of parallelism, and performance portability—all without modifying the algorithms themselves. Active messages are an effective abstraction for expressing graph applications because they allow the fine-grained dependency structure of the computations to be expressed directly in a form that can be observed dynamically at runtime as it is discovered. At this point a variety of optimizations are available (coalescing, reductions, etc.) that are difficult or impossible to apply effectively at compile time; if applied manually, they would greatly complicate the structure of the algorithm's implementation. This expression provides users a natural and flexible method to express applications in terms of individual operations on vertices and edges rather than in terms of artificially coarser operations. Furthermore, algorithms expressed in this form can be mapped to a many hardware platforms and parallelized using a variety of techniques (processes, threads, accelerators, and combinations thereof). By allowing algorithms to be expressed in a form that places minimal constraints on the implementation, the active message expression of algorithms can be implemented in the manner that is most efficient for a given execution context. Separating the specification of graph algorithms from the details of their execution using active messages yields flexible and expressive semantics and high performance.

[1]The missing data points for "AM SV" are due to the wall clock time limit expiring before the algorithm completed, not memory exhaustion. Challenger has a maximum time limit of 1 hour on all jobs.

One straightforward approach to improving performance would be to vary the parameters of the messaging framework dynamically at runtime in order to take advantage of the structure of the graph and algorithm workload as they are observed. The active message abstraction is extremely flexible with regard to its execution context which would make it straightforward to port to new architectures and accelerators including FPGAs and, given appropriate memory models, GPUs. The Active Pebbles model utilized here generalizes one-sided operations to user-defined handlers; applying the same technique to abstract transactions on the graph metadata to create a domain-specific language would enable similar optimization techniques to those utilized on the messaging portion of an application to be applied to the shared memory component.

8 Acknowledgements

This work was supported by NSF grant CCF-11118882. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. We would also like to thank Joseph Cottam for helpful visualizations, as well as the anonymous reviewers for their comments. A preliminary version of this work appeared as a poster [17].

9 REFERENCES

[1] *NVIDIA CUDA Programming Guide*, 2007.

[2] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementations. In *SIAM Meeting on Alg. Eng. and Exp.*, January 2007.

[3] D. A. Bader and K. Madduri. SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Intl. Par. and Dist. Proc. Symp.*, Apr. 2008.

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

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, et al. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

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

[7] A. Buluç. *Linear algebraic primitives for parallel computing on large graphs*. PhD thesis, University of California, Santa Barbara, 2010.

[8] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *J. HPC Applications*, 25(4), November 2011.

[9] 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.

[10] P. Charles, C. Grothoff, V. A. Saraswat, et al. X10: An object-oriented approach to non-uniform cluster computing. In *Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[11] D. Chavarria-Miranda, S. Krishnamoorthy, and A. Vishnu. Global Futures: A multithreaded execution model for Global Arrays-based applications. In *CCGRID*, pages 393–401, 2012.

[12] 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.

[13] M. Dayarathna, C. Houngkaew, and T. Suzumura. Introducing ScaleGraph: an X10 library for billion scale graph analytics. In *X10 Workshop*, pages 6:1–6:9, New York, NY, USA, 2012. ACM.

[14] S. J. Deitz, S.-E. Choi, and D. Iten. Five powerful Chapel idioms, May 2010.

[15] N. Edmonds, D. Gregor, and A. Lumsdaine. Extensible PGAS semantics for C++. In *Conference on PGAS Programming Models*, Oct 2010.

[16] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *Intl. Conference on High Performance Computing*, Goa, India, Dec. 2010.

[17] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. In *PPoPP*, Feb. 2013. Poster.

[18] P. Erdős, R. L. Graham, and E. Szemeredi. On sparse graphs with dense long paths. Technical Report CS-TR-75-504, Stanford University, 1975.

[19] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *Intl. Parallel and Distributed Processing Symposium*, volume 1800 of *LNCS*, pages 505–511. Springer, 2000.

[20] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software–Practice and Experience*, 21(11):1129–1164, 1991.

[21] J. R. Gilbert, V. B. Shah, and S. Reinhardt. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering*, 10(2):20–25, 2008.

[22] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Object-oriented programming, systems, languages, and applications*, pages 423–437, October 2005.

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

[24] J. Jose, S. Potluri, M. Luo, et al. UPC Queues for scalable graph traversals: Design and evaluation on InfiniBand clusters. In *Conference on PGAS Programming Models*, Oct. 2011.

[25] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

[26] S. Kumar, G. Dozsa, G. Almasi, et al. The Deep Computing Messaging Framework: Generalized scalable message passing on the Blue Gene/P supercomputer. In *Intl. Conference on Supercomputing*, pages 94–103, 2008.

[27] K. Lang. Fixing two weaknesses of the spectral method. In *Neural Information Processing Systems*, 2005.

[28] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

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

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

[31] MPI Forum. MPI: A Message-Passing Interface Standard. v2.2, Sept. 2009.

[32] MPI Forum. MPI: A Message-Passing Interface Standard. v3.0, Sept. 2012.

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

[34] OpenMP Architecture Review Board. OpenMP application program interface. Specification, 2011.

[35] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford, November 1998.

[36] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985.

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

[38] V. Shah and J. R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In *International Conference on High Performance Computing*, pages 144–155. Springer, 2004.

[39] Y. Shiloach and U. Vishkin. An $\mathcal{O}(log\ n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[40] F. Song, A. Yarkhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Supercomputing*, Portland, OR, November 2009.

[41] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010.

[42] G. Tanase, A. Buss, A. Fidel, et al. The STAPL parallel container framework. In *Principles and Practice of Parallel Programming*, pages 235–246, New York, NY, USA, 2011.

[43] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger. ARMI: A high level communication library for STAPL. *Parallel Processing Letters*, 16(02):261–280, 2006.

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

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

[46] 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.

[47] K. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Multithreaded Architectures and Applications*, 2008.

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

[49] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. Active Pebbles: Parallel programming for data-driven applications. In *International Conference on Supercomputing*, Tucson, Arizona, May 2011.