Avalanche: A Fine-Grained Flow Graph Model for Irregular Applications on Distributed-Memory Systems

Jeremiah J. Willcock Ryan R. Newton Andrew Lumsdaine
Indiana University
{jewillco,rrnewton,lums}@cs.indiana.edu

Abstract

Flow graph models have recently become increasingly popular as a way to express parallel computations. However, most of these models either require specialized languages and compilers or are library-based solutions requiring coarse-grained applications to achieve acceptable performance. Yet, graph algorithms and other irregular applications are increasingly important to modern high-performance computing, and these applications are not amenable to coarsening without complicating algorithm structure. One effective existing approach for these applications relies on *active messages*. However, the separation of control flow between the main program and active message handlers introduces programming difficulties. To ameliorate this problem, we present Avalanche, a flow graph model for fine-grained applications that automatically generates active-message handlers. Avalanche is built as a C++ library on top of our previously-developed Active Pebbles model; a set of combinators builds graphs at compile-time, allowing several optimizations to be applied by the library and a standard C++ compiler. In particular, consecutive flow graph nodes can be fused; experimental results show that flow graphs built from small components can still efficiently operate on fine-grained data.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.2 [*Programming Languages*]: Language Classifications—Data-Flow Languages

General Terms Design, Performance

Keywords Data flow programming, irregular applications, distributed-memory parallelism, active messages

1. Introduction

Recently, there has been a resurgence of *flow-graph-based* parallel programming models, based on earlier data-flow work [15, 16, 26, 31], wherein a program takes the form of a graph of kernel functions, connected by edges through which streams or other data collections flow [3, 13, 17, 35, 47]. This class of models is differentiated from more general purpose programs by having very structured, predictable control-flow and data-flow. As a result, they offer some of the best examples of implicit parallelism and portability across parallel architectures [17], and have been successful both for fine-grained parallelism [17, 35], as well as distributed processing of big data [3, 13]. Flow-based techniques, however, have not been applied as heavily to graph algorithms and other irregular applications, although other domain specific languages have been proposed [22], as well as several libraries [18, 30]. To address this need we propose a new C++ library, *Avalanche*, for expressing fine-grained applications as flow graphs and executing them efficiently on distributed-memory systems using the Active Pebbles execution model [50].

When programs must ultimately take the form of flow graphs, there are a few major questions that must be answered:

1. How is the graph constructed? (e.g., literally/programmatically, explicitly/implicitly, statically/dynamically)
2. Do data items communicated through edges have a one-to-one correspondence to messages in the implementation? (This often depends on how much is known statically about communication in the flow graph.)
3. Likewise, does a flow graph node (a computation) have a one-to-one correspondence to a parallel task in the implementation?

There are many answers to the first question. Some systems include syntax to encode flow graphs literally and statically [10]; others include functions for building flow graphs edge-by-edge at program runtime [5], and some even allow the graph to be extended while it is executing [5, 12]. Still other models make the flow graph itself implicit; for example, in FRP (Functional Reactive Programming) [47] and related models [36, 37], programs manipulate stream values, or functions over streams, as first class objects, thereby implicitly creating graphs of computations.

While there have been both imperative and functional formulations of flow-graph-based programming models, the history of data-flow models is intertwined with the history of functional programming [15, 16], especially as regarding the formation of data-flow graphs as recurrence equations. Also, from the functional tradition comes a technique of employing *combinator libraries* for composing flow graphs [1]. Avalanche adopts this approach, as well as making use of other hallmarks of functional programming (e.g., λ-expressions).

1.1 Kernel Fusion in Libraries

For questions two and three above, it is often important that the answer be an emphatic *no*—kernel fusion, deforestation, and granularity adjustment[1] are important in many flow-based systems in

[1] In other words: combining graph nodes, eliminating intermediate communication structures during combining, and adjusting the size of data messages and therefore the amount of computation involved in processing a single message.

order to achieve both composability and performance. Fusion and deforestation in particular have a long history in functional programming [46], based on a recognition that the ability to fuse recursive functions over lists, eliminating (deforesting) intermediate lists, enables the composition of applications from modular pieces without performance cost.

This principle has now become quite relevant to the computing world at large. For example, these optimizations are critical to FlumeJava [13], a high-level data-processing library, built on MapReduce, developed and heavily used by Google. FlumeJava optimizes pipelines of data-parallel kernels, scheduling them onto MapReduce jobs, which are by convention constrained to only a single map phase and single reduce phase. By *fusing* kernels, FlumeJava requires many fewer MapReduce jobs than a naïve implementation.

Normally, fusion and deforestation require a compiler. Traditional libraries such as MPI typically provide a collection of functions but cannot optimize *combinations* of those functions, with the exception of compilers that have extensible optimizers, such as the Glasgow Haskell Compiler's system of rewrite rules. And yet, with the right kind of support from its host language, libraries can include compiler-like features [23, 32, 44], with the extreme case being embedded languages packaged as libraries. For example, in FlumeJava, *parallelDo* operations do not immediately execute. Rather, they employ a form of delayed evaluation, building up a representation of the pipeline, which is compiled and executed only when results are requested. This technique achieves deforestation—intermediate results are not stored between fused *parallelDo* operations—but it does not enable inlining and cross optimization of the user's code between the fused operations, making it not as suitable for fine-grained use as other flow-based models (e.g., the StreamIt [17] compiler).

Avalanche employs a similar technique, but is able to take it further through use of C++ template metaprogramming. As we will see in Section 4, flow graphs composed using Avalanche's combinators are able to make a node's downstream consumers known at compile time. Therefore calls from user code to *emit* messages can become direct, inlinable function calls to downstream nodes. This approach to composition is used everywhere that adjacent nodes execute on the same processor, that is, are not separated by a communication operator (e.g., *redistribute*, Section 4).

Avalanche's implementation makes heavy use of functional programming within C++, with many higher-order functions built using C++11's lambda expressions and the *std::bind* function (a more flexible, but more cumbersome, way to create function objects). The implementation uses rank-2 polymorphism in many places, with interfaces that require nodes to be able to be composed with arbitrary continuation types.

The contributions of this paper are:

- A novel library combining a flow graph abstraction with asynchronous fine-grained distributed messaging.
- A flow graph library in a standard language that enables full kernel fusion and deforestation.
- A flow graph model that enables a suite of advanced communication optimizations, including granularity adjustment, by leveraging the Active Pebbles framework (Section 2).

In the remainder of this paper, we will first review the underlying communication framework, Active Pebbles, in Section 2. Then Section 3 describes the design of Avalanche, including the behavior of flow graphs and the major combinators used in programming. Section 4 then addresses the implementation, focusing on the mechanics of composing nodes with all the necessary compile-time information. Section 5 introduces the example programs which will also serve as our benchmarks, and Section 6 reports the results of our evaluation.

2. Background: Active Pebbles

Avalanche is built on top of the Active Pebbles programming and execution model [50], and uses the AM++ implementation of that model [49] as its underlying infrastructure. The version of AM++ used in this work uses the Message Passing Interface (MPI) standard [34] as its underlying communication mechanism, although it could be re-targeted to a lower-level message passing mechanism such as InfiniBand Verbs.

The overall Active Pebbles model is distributed-memory and single-program, multiple-data (SPMD); a number of *processes* run the same program on separate data, and the only sharing of data between the processes is by sending messages and other explicit communication operations.

The communication model consists of fine-grained active messages [45] sent to arbitrary user-defined targets. A target can be an arbitrary entity (including an element in an array, or something completely implicit), as long as a user-defined data distribution object can map from a target identifier to a destination rank number (i.e., machine or processor identifier).

Message handlers in Active Pebbles can themselves send messages to arbitrary destinations, increasing flexibility. The goal of Active Pebbles is to allow irregular applications with fine-grained messages and control flow to be expressed at their natural levels of granularity; e.g., a graph algorithm can be implemented by sending messages to individual graph vertices. The Active Pebbles execution model then performs any necessary coarsening (such as message coalescing) to enable efficient message passing on a given architecture.

The execution model also includes software routing of messages to limit the number of neighbors that a particular node must communicate with directly, reducing the number of coalescing buffers required at large scales. It also includes message reductions: using a cache to remove duplicate messages or combine related messages to save communication. The synchronization model in Active Pebbles is based on epochs; at the end of an epoch, all messages must have been received and their handlers executed, including messages recursively sent from handlers. Standard termination detection algorithms are used for this purpose. An integer value (of type ***uintmax_t***) can be summed across all processes at the end of an epoch; this feature is used to globally determine when all processes' queues are empty in a breadth-first search, for example.

Transports (similar to communicators in MPI) are used to separate independent communication domains. Different transports have completely separate sets of message types and epochs. This feature is used to allow subgraphs of a larger flow graph to have their own synchronization behavior (i.e., their own epochs and summed values).

3. The Avalanche System

The Avalanche programming model is built on the same basic infrastructure as Active Pebbles: a SPMD, distributed-memory model with explicit communication between processes running in separate memory spaces. Fine-grained messages are sent between flow graph nodes, either locally within a process or using Active Pebbles communication operations between processes. The full flexibility and performance optimizations of the Active Pebbles execution model are exposed in Avalanche, including configuration of data distributions, message coalescing, routing, and message reductions. Flow graph nodes can also write to the integer value accumulated during termination detection, although limitations in AM++ restrict the timing of these writes.

Some aspects of Avalanche are modeled after tbb::flow, a component of Intel's Threading Building Blocks [4], including the names of some operators and types (*function_node*, *continue_msg*, etc.). Although the dynamic graph construction is similar as well, the model for building graphs at compile-time is very different from tbb::flow's approach of adding nodes and edges at run-time.

Flow graphs are built using *combinators*, which in this case are functions that produce flow graph nodes (objects with a particular interface). These combinators can also accept partial flow graphs as input, allowing composition of graphs. The structure of most flow graphs is set at compile-time by the pattern of combinator calls used to define them, enabling the C++ compiler to optimize patterns of nodes. However, flow graphs can also be defined at run-time, with arbitrary wiring patterns between the nodes; a performance penalty is present when these are used.

Most flow graph nodes are single-input, single-output, with sequential composition as the main coordination operation between them; however, combinators can accept pipelines as arguments, allowing for other forms of composition. For example, combinators can form loops or branching structures. The compile-time combinators currently implemented produce structured flow graphs, but new combinators could be defined to create unstructured flow graphs at compile-time. Arbitrary series-parallel graphs can be created directly from these combinators. Arbitrary graphs can be implemented by creating a superset of the desired graph (possibly using discriminated unions to multiplex several message types in the same pipeline), with filters to restrict message flow. The run-time graph combinators and wiring functions do not impose any structure on the flow graphs that they create. The set of primitive combinators is described in Table 1; others can be built out of the initial, continuation-passing style function, and final function node types.

The current implementation runs all pipeline nodes on all processes, interleaving the nodes' sequential executions. Thus, for a 10-node flow graph running on 20 processes, 20 threads will be running in parallel on different processes, each interleaving execution of the 10 flow graph nodes within its process. This model is very efficient because of the optimizations that can be performed between nodes, but does not exploit shared-memory parallelism within a process. A flow graph node type could be defined, however, that creates a task for each of its input values using a work-stealing system such as Cilk [9].

3.1 Combinators

3.1.1 Pipeline

The pipeline combinator takes two flow graph nodes and composes them by feeding the output of one to the input of the other, producing a new flow graph node as output. Given two nodes a and b, the composition $a \mid b$, when given input type t_1 first defines t_2 as a's output type on input t_1 (t_2 can be an arbitrary function of t_1); it then gives its output type as b's output type when run with input type t_2. This operation composes the two type functions. After that has been computed, continuations for each node must flow in the other direction: given an output continuation f, the implementation passes that to b, receiving a new continuation g. The function g is then passed to a, giving the continuation type for the pipeline. Thus, two passes are made over each pipeline: one from left to right to compute the types of values flowing between nodes, and a second one from right to left to compute the continuation for each node.

3.1.2 Initial Node

The base case at the beginning of a pipeline is an initial node. The initial node constructor accepts an object with two methods, *setup* and *run*. The *setup* method is called at the beginning of pipeline execution; *run* is called repeatedly while an outer flow graph execution loop waits for the rest of the pipeline and its communications to terminate. The *setup* method can also update the accumulated value in the AM++ transport; the sum of this value on all processes will be computed implicitly as one result of finishing the pipeline. An initial node also accepts a successor and can send messages to it, either in *setup* or *run*.

3.1.3 CPS Function Node

The most common intermediate node in a pipeline is a single-input, single-output continuation-passing style (CPS) function node. The basic CPS function node uses a function that is called with a continuation, along with a type function to compute the node's output stream element type. Because the type of the continuation is not known until the rest of the flow graph is assembled at compile-time, functions passed to the CPS function node must be polymorphic in their continuation type. Because of limitations in the C++11 language, these functions cannot be built using lambda expressions or local classes, and so must be defined in separate classes with captured local variables explicitly stored and accessed.

The continuations passed to a CPS function node represent the computation (portion of the flow graph) downstream from that node; they can be invoked multiple times by the user,[2] allowing the node to produce zero or more values for a single input. However, a wrapper (*function node*) is provided for the common case of functions that take a single input and process it into a single output. The non-CPS function wrapper converts an ordinary function to a CPS function, deriving the output type automatically from the function.

Many other types of nodes can be implemented as special cases of CPS function nodes, sometimes with "functions" that have internal state that they modify when called. For example, one special case provided by the library is *filter*, which takes a user-defined function object that returns a Boolean value and uses it to filter a stream of messages: only those for which the function returns ***true*** are passed on, and others are removed. Special kinds of function nodes are also used for specific applications, such as the node used in breadth-first search that outputs the neighbors of each vertex passed to it as input.

One particular kind of function node useful for synchronization is the *subgraph* node. This combinator accepts a full pipeline as argument, and produces a node encapsulating that pipeline. When a message is sent to the resulting node, the entire subgraph is run in a separate AM++ transport, and the subgraph node does not produce any output until the subgraph has finished completely on all processes.

In the current prototype, the results of subgraphs are often communicated back to the main program through side effects; the type of the subgraph's output is fixed to be ***uintmax_t***, and the subgraph node implicitly does a sum-reduction over all outputs from the subgraph. After the subgraph's execution is complete, it produces a single output—the global, summed value—which is output on the subgraph node's output port on all processes. The subgraph node is thus important for providing a barrier that ensures message receipt and processing globally. The return value, while limited, is sufficient for a number of purposes. More general reductions would be a topic for future work.

3.1.4 Final Function Node

A special kind of function node is used to end pipelines. It is similar to a normal function node, except that it may not have a successor

[2] This makes them similar to *delimited* continuations; they do not destroy the current stack when called. The continuation of an Avalanche CPS function node is used identically to the *emit* function which is passed into the body of FlumeJava's *parallelDo*.

Name	Description	Input(s)
Pipeline	Sequentially composes two pipeline fragments	Two pipeline fragments
Initial	Starts a pipeline	Object with *setup* and *run* methods
CPS function	Converts input into zero or more outputs based on function	Function object
Final function	Like CPS function, but ends a pipeline	Function object
Loop	Cyclic flow of messages	Pipeline fragment
Broadcast/merge	Copies message to several subgraphs, merging the results	Two pipeline fragments
Redistribute	Moves data between processes based on data distribution	Data distribution and message configuration
Parallel	Combines independent, complete pipelines	Two pipelines
Dynamic sender	Allows run-time connection to dynamic receiver	Nothing
Dynamic receiver	Receives messages from dynamic sender	Nothing

Table 1. Primitive Combinators in Avalanche.

and accordingly does not pass an output continuation to its user-provided function. Thus, it creates a single-input, no-output node.

3.1.5 Loop

The *loop* combinator takes a single-input, single-output node as argument. It returns a new node which, upon receiving input, repeatedly executes the child node, feeding the child's subsequent output back to its input, until no further outputs are produced. There is an implicit merge of two streams at the top (both external inputs and self-feedback are fed into the child), while outputs from the child are sent back to the top of the loop. This node type never emits any values on its output.

The loop node works by employing the type-erased *std::function* class to store the continuation corresponding to its body. Thus, the continuation's type does not need to be known statically, avoiding a circular dependency. A higher-performance approach would be to store the continuation in a type-erased way, but to downcast it to the correct type (the continuation type for the body when given the loop node as successor) to allow the function call to be resolved at compile-time.

3.1.6 Broadcast/Merge

The broadcast/merge combinator accepts two pipelines with the same input and output types as parameters, returning a new pipeline with the same input and output types. When a message is sent to this pipeline, it is broadcast to both constituent pipelines; any results from those pipelines are merged in an arbitrary order to form the output of the combined pipeline,[3] that is, they are merged rather than *zipped*.

To make this combinator more useful, a fork/join wrapper is implemented on top of it. This wrapper takes two sub-pipelines, which must have the same input type but may have different output types, and coordinates their behavior. When an input message arrives, it is assigned a unique identifier, then passed down both branches to the sub-pipelines. The result of each pipeline is placed into a discriminated union to mark which pipeline it came from, and then the identifiers of the outputs are matched to form a combined result. Each branch of the pipeline is expected to produce one output for each input, allowing a one-to-one matching of the outputs. Other than the broadcast/merge node, all of these operations are performed by CPS function nodes, some of which use stateful function objects. The fork/join functionality provides the ability to coordinate processing of a single message by multiple processes simultaneously, giving directed-acyclic-graph (DAG) coordination structures similar to those provided by Structured Dagger [28].

[3] This *nondeterministic merge* makes Avalanche potentially non-deterministic, even if the code within function nodes were effect-free. This along with the output data-rates of CPS function nodes being unknown, makes Avalanche more similar to asynchronous data-flow models like WaveScript [37] rather than synchronous ones like StreamIt [17].

3.1.7 Redistribute

Data parallelism across a distributed-memory computer is obtained via two features: the SPMD nature of the programming model, and data redistribution nodes. A redistribute node takes a user-defined, arbitrary data distribution (represented as a function from messages to node numbers) and some configuration settings and moves messages from its input channel to the nodes named by the data distribution, then outputs them on its output channel on those nodes. The data distribution can be completely user-defined, and can use compile-time and/or run-time information; the data distributions implemented so far are static distributions such as a cyclic distribution. As the Active Pebbles programming and execution model is used underneath the flow graph system, messages can be sent using techniques such as coalescing, software-based routing, and duplicate message elimination to improve performance. The configuration information passed to the redistribute node contains the relevant settings for these capabilities. The continuation-based nature of the flow graph is very helpful for the redistribute node: the continuation for the output of the redistribute node can be almost directly be used as the active message handler for the communication library.

An alternative model for handling data distributions would be to have each node associated with a data distribution, performing communication implicitly between nodes to match the distributions. This model would be cleaner in some ways, but any node could perform remote communication without showing that explicitly. In the current model, only explicitly designated nodes can cause communication, and only those nodes need to be associated with data distributions. Since most nodes do not need data redistribution, needing to specify the same data distribution for each node would be verbose. Also, our framework benefits greatly from type-based code specialization at compile-time, and whether two data distributions are equal (removing the need for communication) may not be testable until run-time. Code could be generated for both cases at compile-time, with the decision between them made at run-time, but pipelines could then lead to exponential growth in code size.

The semantics of the redistribute node allow messages passing through it to be reordered arbitrarily (subject to the rule that they must all be handled when the Active Pebbles epoch, i.e., overall pipeline execution or a *subgraph* node, finishes). Messages may take different amounts of time to be sent, for example, and write-back caches for message combining in AM++ may lead to cache entries being sent in a different order than they were put into the cache. Most applications targeted by Avalanche do not have ordering constraints, and any desired ordering can be done by assigning tags to messages then creating a node that reorders incoming messages by tag (as is done in tbb::flow [4]).

3.2 Dynamic Graphs

In addition to the compile-time combinators for building flow graphs, customization points can be added to graphs to allow rewiring at run-time. Because the graph's structure can no longer be fully exposed to the compiler, there is a small performance penalty for each message sent through a link established at run-time. Run-time wiring is achieved using two node types: *dynamic_sender<T>* and *dynamic_receiver<T>*. In these two types, *T* is the type of data being communicated; that is still checked at compile-time. A *dynamic_sender* node contains a place for a type-erased function object representing the node to send to. (We use *std::function* from the C++11 standard library, which erases the original nominal type for the object, leaving something similar to a basic function type in Haskell or ML, e.g., $\alpha \to \beta$). When an edge is added from that sender to a *dynamic_receiver*, the receiver's continuation is stored into the sender. Indirect calls through the *std::function* object are the source of the overhead in this graph structure. In addition to the dynamic sender and receiver nodes, a parallel pipeline combinator must be used to merge two independent-looking pipelines into a single graph at compile-time to be run together, even though their exact linkage will not be known until run-time. Run-time-sized parallel combinators and broadcast nodes could be added later.

3.3 Hidden Data

One operator, for adding "hidden data," creates a modified copy of an existing graph structure rather than wrapping the structure. The goal of the hidden data combinator is to pass a given piece of data around with all messages in a portion of the graph, automatically tracking message dependencies. This combinator is meant to apply to a single-input, single-output portion of a graph, and so operates on pipeline, function, and redistribution nodes; broadcast and dynamic nodes could be added later. This combinator can be viewed as the Avalanche analog of Haskell's *Control.Arrow.second* function. Given a flow graph with input type *I* and output type *O*, this combinator produces a new flow graph with input type *(H, I)* and output type *(H, O)* for an arbitrary type *H*. For each input to a flow graph, this combinator tracks all outputs derived from that input, using the semantics of the function, pipeline, and redistribute operators to determine which outputs are derived from which inputs.

The hidden data operator copies the structure of the graph portion given as its input, modifying function and redistribute nodes while preserving pipeline structure. On each function node, it creates a new function node that accepts pairs of values, calls the user function on the second element of each pair, then attaches the first element of each pair to each of the function's outputs. On redistribute nodes, the hidden data operator modifies the node to communicate the extra data along with the data the node was originally communicating, modifying the data distribution to ignore the extra data when computing the destination of each message.

The main use case for this operator is to refactor a flow graph to add extra information that was not accounted for in its original design. For example, a graph can be built that accepts vertices from a graph and outputs their outgoing edges. However, if a user wants to instead give $\langle vertex, distance\rangle$ pairs and get $\langle out\text{-}edge, distance\rangle$ pairs, the hidden data operator would be used to pass the distance value through the existing graph and attaches it to each outgoing edge corresponding to a given input vertex.

4. Implementation

Avalanche is written in standard C++11 [25], taking advantage of its new features such as lambda functions, standardized function composition operations, move semantics (a form of uniqueness type system), and local type inference. It also incorporates older C++ template metaprogramming techniques such as the ability to achieve higher-rank polymorphism by passing objects containing polymorphic members as arguments to other templates and heavy use of continuation-passing style. The goal of using these sophisticated techniques rather than a more direct implementation is to enable the underlying C++ compiler to optimize the flow graphs more effectively.

At an abstract level, a flow graph pipeline can be defined in Haskell (used as a pseudocode for explanation) as:

```
-- i is the input type, o is the output type
data Pipeline i o = Pipeline ((o → IO ()) → IO (i → IO ()))
```

This formulation allows side effects both in creating the node and in its (and its continuation's) action on each input element. The first generalization we make for programming simplicity is to allow a pipeline to work on a set of input types, with potentially a separate output type for each one. In C++, this allows intermediate types in the pipeline to be inferred; it also requires that pipeline nodes may have different types, necessitating a type class in Haskell. In the Haskell pseudocode, functional dependencies are used for brevity; the actual implementation uses member types, a C++ equivalent of associated type families.

```
class Pipeline p i o | p i → o where
  composePipeline :: p → (o → IO ()) → IO (i → IO ())
```

Note that the type p may have non-trivial values: it is not just a phantom type, although it could be used that way if there is only one pipeline of a given type. In C++, using the equivalent to a type of the form $t \to u$ (*std::function*) causes a run-time overhead for each call to the function, and usually disables compiler optimizations such as inlining. The standard technique to avoid this is for each function definition to have a separate type: the type determines which code will be run, while the function object's value containing values for the function's free variables (if any). Thus, each continuation has a separate type, and a new type class is used to represent function-like types.

```
-- This class introduces a new form of functions and function
-- applications, allowing dispatch to code based on type alone:
class Function f a | f → a where
  call :: f → a → IO ()

-- Node composition, based on the new notion of a Function.
-- 'nx' is the type of the node's continuation
class (forall nx. Function nx o ⇒ Function (Composed p nx) i)
      ⇒ Pipeline p i o | p i → o where
  type Composed p :: * → *
  composePipeline :: p → nx → IO (Composed p nx)
```

In this example, the *Pipeline* class requires a polymorphic function; thus, any function that accepts a pipeline as argument is rank-2 polymorphic. In the C++ implementation, the equivalent to *composePipeline* is written in continuation-passing style, and so its continuation needs to be able to accept arbitrary composed continuations (shown as *Composed* in this pseudocode), leading to more rank-2 polymorphism.

In the actual C++ implementation, a pipeline node is required to support one function, *set_input_type*. The signature for a pipeline node type *node* and input value type *in* is:

```
template <typename K>
void set_input_type(
  K&& k, node& n, boost::mpl::identity<in>);
```

Here, *boost::mpl::identity* is a class from the Boost Metaprogramming Library [19] that represents an arbitrary type without storing a value of that type. The *set_input_type* function is required to call its continuation *k* on some new object of unspecified type; call that type *node2*. The type *node2* is required to have a member type named *output_type* giving the type of values that it outputs to its successor. Additionally, a second function must be defined:

template <***typename*** *K*, ***typename*** *Next*>
void *bind_next_message_type(*
  K&& k, node2& me, Next&& nx,
  amplusplus::transport& trans, ***uintmax_t****& end_value);*

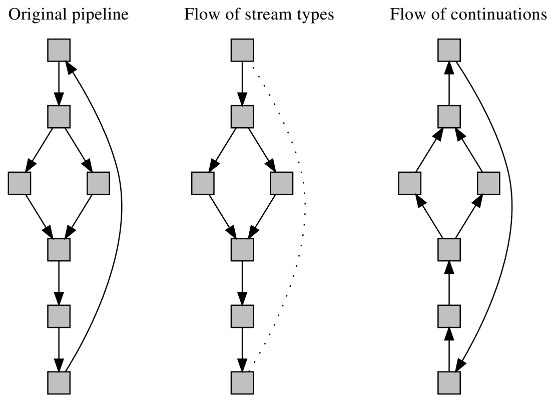

Figure 1. Flow of element types and continuations during processing of a pipeline.

This function is required to call its continuation *k* on the composition of the node *me* and the flow graph continuation *nx*; this composition can have arbitrary type. The type produced is required to be accessible via a member *bind_next_message_type_result*. Finally, the result of *bind_next_message_type* is an object having a *get_send_func()* member function, which then becomes the flow graph continuation for this node's predecessor. The overall flow of data at compile-time is shown in Figure 1. In effect, two passes over the flow graph are done: one in the forward direction to set input and output types, and a second one in the reverse direction to link each flow graph node to its successor's continuation.

The goal of all of this complexity is performance: C++ compilers specialize polymorphic classes and functions for each particular set of input types,[4] allowing the particular continuation types in use to be computed statically. Thus, exactly which code will be called as the continuation of each flow graph node is known, allowing that code to be inlined into the node's body. Unlike some functional language compilers [48], C++ compilers typically have difficulty optimizing complex patterns of indirect function calls. This type specialization only applies to parts of the flow graph built at compile-time; the dynamic graph combinators do not allow this type information to be passed around as it is not known at compile-time, and so optimization opportunities are lost in exchange for flexibility. These type-system techniques, a form of expression templates [43], end up acting as a synthetic form of deforestation [46].

Because the functions for processing and rebuilding the flow graph at each step are written in continuation-passing style, local variables in each function will stay allocated until the flow graph has finished executing. Thus, this control flow structure can be viewed as a form of Cheney queue [8] in that it uses the stack as a source for dynamically-allocated memory. The main benefit is that functions can create data structures locally, then return references to those data structures by calling the functions' continuations. The disadvantage of this approach is that the compiler appears to be unable to statically determine where those references point, treating them as pointers to unknown places in memory. Thus, some data needed to be copied into value members of continuation objects and flow graph nodes. C++11 move semantics, a form of uniqueness type, were used to optimize copying into moving when it is known that the object being copied is no longer alive after the copy. A move operation can perform ownership transfer of large objects rather than copying them, increasing performance. In AM++, message types contain members such as coalescing buffers that should not be copied, and so these types are declared as *move-only*; the type system ensures that only one copy of each move-only object exists at a time.

[4] Although the standard does not mandate this behavior, all mainstream compilers work this way.

The approach used in Avalanche leads to two main optimizations: inlining of non-redistribution nodes into their predecessors, and message type optimizations from Active Pebbles/AM++. Inlining comes from the representation of node inputs as continuations, and the composition of those continuations onto predecessor nodes. In combination with the tracking of flow graph structure through C++'s type system, the compiler is able to determine the function call graph statically and is thus able to optimize it. Most flow graph functions are small, and so compositions of them are likely to be optimized. The second source of optimization is through the use of AM++ as underlying communication layer. AM++ does many optimizations on communication operations, including calling handlers immediately if a process is sending a message to itself; the need to do this check in the application or Avalanche is thus avoided. AM++ also coalesces small messages into larger ones, calling the handlers for the resulting small messages in a tight loop in the destination process; that loop's body is also statically known, allowing compiler optimizations even for remote active message operations. Avalanche also exposes the performance capabilities of AM++ directly to users, including the ability to add removal of duplicate and other combinable messages using caches directly in AM++; in combination with routing, duplicates can be removed even on intermediate nodes between the initial sender and final receiver of a message.

Because of Avalanche's heavy use of higher-order functions, the ability to create function objects with non-trivial closures is important. C++11 provides lambda expressions, which automatically create closures containing all free local variables used in the body of the lambda function; however, lambda functions cannot be polymorphic and must either copy or refer to free variables (the closure cannot move values of its free variables in). The first of these limitations is especially problematic to this work: many types of nodes accept functions as input that must be polymorphic to operate on unknown continuation types. Thus, either manually written classes or a more cumbersome Standard Library mechanism, *std::bind*, must be used instead. The bind operators are a general syntax for function composition, in which placeholders represent function parameters. Bind expressions are difficult to write for complicated functions, but they do handle building closures almost automatically and allow polymorphic functions to be generated. Manually-written function objects are often used instead in Avalanche and its examples for readability. Thus, this work exposes the limitations of functional programming in C++, even with the improvements in C++11.

5. Examples

To demonstrate and evaluate Avalanche, we converted two example programs from the paper that introduced the Active Pebbles programming and execution model [50]. These examples are Permutation (a simple benchmark that uses two steps of data redistribution) and Breadth-First Search (a graph algorithm). These programs had already been written in manual active message style as part of that previous work, and thus can easily be compared against flow graph translations of them (see Section 6).

5.1 Permutation

The Permutation benchmark models the reorganization of data that may follow a data input operation. The benchmark starts with a set of $\langle address, data\rangle$ pairs distributed randomly across the computer's processes. The first step of the benchmark is to apply a lookup table (itself distributed) to the address fields of the data elements, producing new addresses. The benchmark then requires that the elements be placed into a distributed array based on these new addresses. Thus, two communication steps are required: one to move the data to access the correct parts of the lookup table, and a second one to move the data elements to their final locations. The pipeline for this operation is straightforward and linear, and is shown in Figure 4; a simplified version of the code is in Figure 2. The overall computation performed for each value of *i* is *data_p[perm[i]] = data[i]*, with conflicts between those writes resolved arbitrarily. Compared to the pseudocode for the Active Pebbles version, shown in Figure 3, the flow graph representation shows the operations in the program in the order in which they are run, while the handler-based approach tangles the program's control flow. Note also that the Active Pebbles version is heavily abbreviated: the actual code is much longer than the Avalanche version shown.

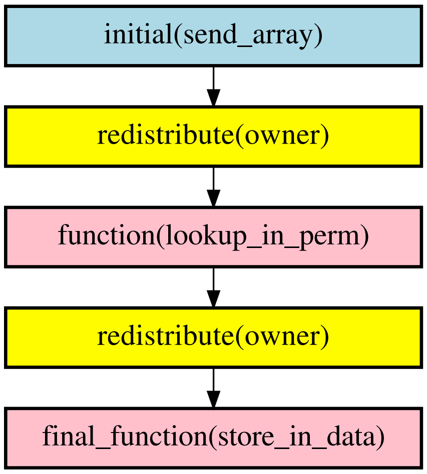

Figure 4. Flow diagram for Permutation.

5.2 Breadth-First Search

The second benchmark shown is breadth-first search (BFS), a simple graph algorithm also used as a benchmark in [50]. Breadth-first search explores all the neighbors of a vertex before proceeding onward to neighbors-of-neighbors. To do a strictly breadth-first search in a parallel implementation it is typically necessary to synchronize with a global barrier before proceeding deeper (otherwise a data-race may occur in which a longer path reaches a node before a shorter one and its depth is misclassified). Such an approach is called *level-synchronized*.

The code for level-synchronized BFS is shown in Figure 5 and the pipeline structure is in Figure 7. In this description, "vertices" and "edges" refer to the input graph being searched by the algorithm, while "node" refers to computation steps in the flow graph. The *loop* node in Figure 5 iterates once for each level by connecting the level's output (the ***return*** *continue_msg*{}; node) back to its input (the subgraph node). The *subgraph* node processes a completely separate graph that represents an individual level, imposing a global synchronization at the end. The output of the subgraph node is the system-wide number of local queues that were nonempty at the start of the subgraph; when that value reaches zero, the *filter* node deletes the message and thus causes the loop to terminate.[5] If the loop does not terminate, the output queue from the loop body becomes the input queue for the next BFS level, and the output queue is cleared. The loop is initiated by the *send_constant* object (a type of *initial* node); *continue_msg* is the name used in tbb::flow [4] for an empty class used to send as a message when no data is required, and we use the same name in Avalanche. The body of the loop looks like a standard breadth-first search algorithm: the input queue is streamed to a function node that gets the targets of each vertex's outgoing edges, redistributes those targets to match the data distribution used for the color map and output queue, and adds any un-visited target vertices into the queue and marks them as visited. In the manually-written Active Pebbles pseudocode shown in Figure 6, the message handler is relatively simple, but the main loop includes several explicit control flow and synchronization operations.

```
// q1 and q2 are of type std::vector<vertex_descriptor>
send_constant(continue_msg{}) | // Trigger loop
loop(
  // Single level of BFS
  subgraph(
   (// Iterate through vertices in input queue
    // This sets the output from the subgraph
    // if q1 is not empty
    iterate_container(q1) |
    // Get their neighbors
    cps_function_node(
      bfs_get_outgoing_edges_t<...>{g, local}) |
    // Move neighbors to their owners
    redistribute(owner_gen, msg_gen) |
    filter([&](vertex_descriptor v) {
              auto key = get(local, v);
              // Test color map
              if (get(color, key) == false) {
                // Not yet visited
                put(color, key, true);
                return true;
              } else {
                // Already visited, so skip
                return false;
              }
            }) |
    // Push into output queue
    final_function_node(
      [&](vertex_descriptor v) {q2.push_back(v);})),
   trans) |
   // Check for all local queues being empty
   filter([](uintmax_t val) {return val != 0;}) |
   // Swap queues for next level
   function_node([&](uintmax_t) {
                 q1.swap(q2); q2.clear();
                 // Go to top of loop
                 return continue_msg{};})) |
eat()
```

Figure 5. Avalanche Code for Breadth-First Search (simplified).

[5] A successor node is required due to an implementation limitation, although no messages will be sent to it.

```
// Send input data
initial(send_data_pairs{data.get(), elements_per_proc, (uint64_t(elements_per_proc) * rank)}) |
// Distribute it by address to match lookup table
redistribute([=](...) {return pair_owner_map;}, msg_gen) |
// Apply lookup table
function_node([=](pair<uint64_t, uint64_t> p) {return make_pair(perm_ptr[get(local_map, p.first)], p.second);}) |
// Distribute by new address for storage in output array
redistribute([=](...) {return pair_owner_map;}, msg_gen) |
// Store data in output array
final_function_node([=](pair<uint64_t, uint64_t> p) {data_p_ptr[get(local_map, p.first)] = p.second;})
```

Figure 2. Avalanche Code for Permutation (simplified).

```
// Write into output array
void final_store_handler(pair<uint64_t, uint64_t> p) {
  data_p[get(local_map, p.first)] = p.second;
}
auto final_store_message = register_message_handler(final_store_handler);

// Look up element in table
void lookup_table_handler(pair<uint64_t, uint64_t> p) {
  final_store_message ->send(make_pair(perm[get(local_map, p.first)], p.second));
}
auto lookup_table_message = register_message_handler(lookup_table_handler);

// Loop over all data elements
for (uint64_t i : my_range) {
  lookup_table_message ->send(make_pair(data[i], i));
}
```

Figure 3. Active Pebbles Pseudocode for Permutation.

6. Evaluation

To show the performance effects of the higher-level flow graph model compared to manually written active message handlers, we compared performance on the two examples shown in Section 5 against the raw AM++ and MPI implementations shown in [50]. The test was performed on Challenger, an IBM Blue Gene/P system at Argonne National Laboratory. The system consists of 1024 nodes, each containing a quad-core, 850 MHz PowerPC 450 processor and 2 GiB of RAM. GCC 4.7.0 in C++11 mode was used as the compiler, with `-Ofast -DNDEBUG` as optimization flags. Both AM++ and the example programs were compiled with thread support disabled (removing locks and atomic operations). Only one core from each node was used for the experiments.

Both tests showed some overhead from the use of Avalanche, with smaller effects on BFS than on Permutation, but the overheads were similar across different scales. The results from Permutation are shown in Figure 8. The overhead from Avalanche varies between 4.5% on one node to 9.7% on 32 nodes. On BFS, shown in Figure 9, the overhead is 12.9% on one node but is between 2.7% and 5.9% on larger node counts. These overheads appear to be caused by extra memory references from the compiler failing to resolve that references between parts of the pipeline actually point to fixed offsets relative to each other. Relative to manually-written MPI implementations, AM++ and Avalanche perform better for Permutation, and worse for BFS; the collective-based MPI implementation ran out of memory running on 512 nodes.

7. Related Work

Other work has explored non-flow-graph approaches for simplifying the programming of active-message-based applications. For example, the X10 [14] and Chapel [11] languages provide asynchronous mechanisms for running a block of code on a designated remote process. This syntax avoids the need to write a separate handler function, but still requires explicit movement of individual pieces of the code as invoked on individual pieces of data. Another approach is the domain-specific language Structured Dagger [28], built on top of the Charm++ active message framework [29]. Structured Dagger provides imperative-like syntax, but also has constructs for sending out multiple messages, waiting for results to arrive from all of them, and combining those results. This model allows for more sophisticated coordination, rather than targeting bulk handling of many messages through pipelines of simple operators like Avalanche does.

Some libraries in C++ use simpler forms of pipelines in the sequential context. For example, Boost.Range 2.0 [38] uses pipeline syntax to compose adaptors on sequences (such as map and filter), with compiler fusion of the sequence adaptors into traversals of the output sequence. However, only linear pipelines are supported, and Boost.Range is purely sequential. The Oven library [41] extends Boost.Range with many more adaptors, including some (such as set union) that merge two sequences, as well as some simple parallel algorithms. However, these libraries do not handle distributed-memory parallelism, and are limited in the types of flow graphs they can express.

The Standard Template Adaptive Parallel Library (STAPL) [7], a generic, parallel C++ library similar to the C++ Standard Library but also containing graph and matrix algorithms, uses task graphs as an internal mechanism for implementing algorithms on distributed-memory systems. STAPL also has parallel views, built by wrapper interfaces on top of physical containers. Algorithms in STAPL are adaptive, and STAPL's scheduler executes nodes of task DAGs as data arrives. It is also based on its own active message framework, ARMI. However, task DAGs represent concrete

```
// Vertex update handler
void update_handler(vertex_descriptor v) {
  if (get(color, v) == false) {
    put(color, v, true);
    q2.push(v);
  }
}
auto update_message = register_message_handler(update_handler);

// Main body of code
while (true) {
  // Local and global tests for all queues being empty
  const unsigned long my_queue_is_empty = q1.empty();
  unsigned long all_queues_are_empty;
  {
    // Create a synchronization epoch
    amplusplus::scoped_epoch_value
      epoch(transport, my_queue_is_empty, all_queues_are_empty);
    // For each vertex in the input queue
    for (vertex_descriptor v : q1) {
      // For each of its neighbors
      for (vertex_descriptor w : neighbors(v)) {
        // Send update
        update_message->send(w);
      }
    }
  }
  // If all ranks had empty queues, stop
  if (all_queues_are_empty == size) break;
  // Swap queues for next level
  q1.swap(q2);
  q2.clear();
}
```

Figure 6. Active Pebbles Pseudocode for Breadth-First Search.

operations on known sets of data, not streams of fine-grained data flowing in a demand-driven manner.

FlumeJava [13] is a Java library for writing MapReduce jobs (described in Section 1). FlumeJava's fundamental abstraction is the parallel collection; operations on parallel collections are not executed, but are built into flow graphs to be optimized and executed later. This model is in a sense dual to the type of model used in Avalanche: rather than working with the flow graph nodes, a system such as FlumeJava works with the edges as the main abstraction. FlumeJava also differs greatly from Avalanche in its target domain. MapReduce workers cannot communicate directly with their peers during a MapReduce job. Avalanche, on the other hand, targets graph applications where high-frequency, asynchronous messaging between workers is the norm. Further, FlumeJava's fusion is limited. FlumeJava essentially interprets a representation of an optimized pipeline at runtime, precluding inlining and cross-optimization of adjacent nodes even when they are fused into the same MapReduce job.

Pregel [33] is a C++ library for large-scale, distributed-memory graph algorithms. A bulk-synchronous parallel model with separate computation and communication steps is assumed; communication is fine-grained, however, with graph vertices able to send messages to other individual vertices. User-defined reduction operations can be applied during communication to all values being sent to a particular vertex, as in Active Pebbles. Pregel's interfaces are primarily object-oriented, likely limiting possible compiler optimizations. It is also not based on flow graphs, but instead on supersteps of imperative computation and communication. Avalanche does not have a superstep structure: multiple nodes in the flow graph can execute in

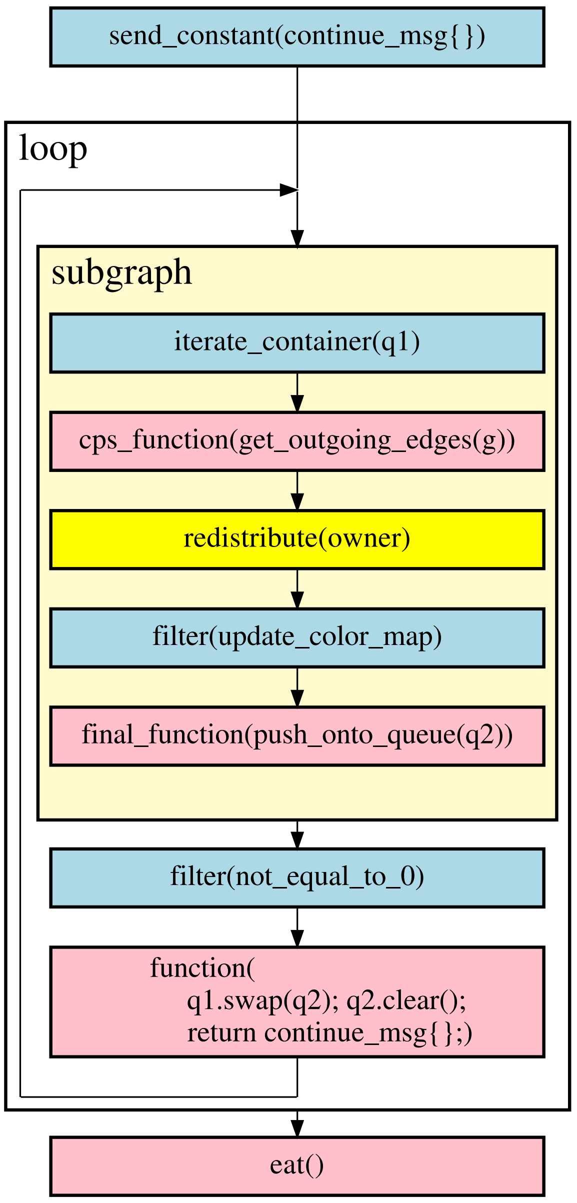

Figure 7. Flow diagram for Breadth-First Search.

parallel, without any global barriers between stages of the pipeline (except for subgraph nodes and other places where synchronization is specified by a particular algorithm).

Green-Marl [22] is a recent domain specific language specifically for writing analyses of immutable graphs. In its current form, Green-Marl primarily targets SMP execution, but a back-end targeting a Pregel-like framework (GPS [40]) is under development. By having a tight focus on a specific class of algorithms, Green-Marl programs can be significantly more concise than Avalanche ones (for example, BFS is a built-in language construct). However, Green-Marl both requires using new syntax (rather than a library) and cannot express as broad a class of algorithms as Avalanche.

Flow Graph Systems

Some flow-graph systems (like FlumeJava) have edges corresponding to unordered collections. Intel's Concurrent Collections (CnC) [10] is another example, wherein graph nodes communi-

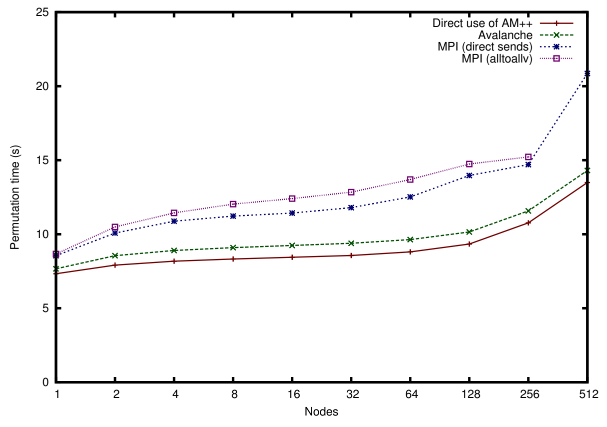

Figure 8. Permutation weak scaling performance (10M elements/node).

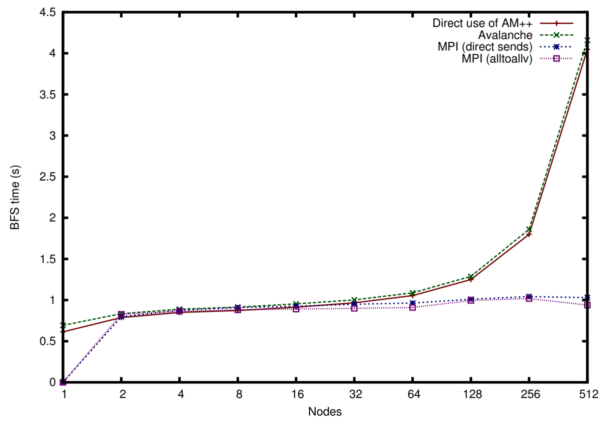

Figure 9. Breadth-first search weak scaling performance (400,000 vertices/node).

cate through key-value stores. However, the majority of previous systems employing flow-graphs communicate through ordered streams. These *stream processing* systems are often based on the synchronous data-flow model. StreamIt [17] is an example in this class; it targets signal processing algorithms and achieves portability across a number of parallel multicore architectures. WaveScript [37] is another example which is more similar to Avalanche in that it allows an arbitrary and data-dependent number of outputs for each input processed by a node. Finally, tbb::flow, part of Intel's Threading Building Blocks [4], is an example of a library approach, rather than a language approach, to parallel stream processing. It is very flexible, allowing both "push" and demand-driven scheduling, and some of its abstractions were inspiration for Avalanche. However, it is limited to shared-memory systems, and is intended for coarser-grained computations without any of the node merging techniques used in Avalanche.

The FastFlow framework [6] also provides high-performance streaming networks for multi-core systems, with distributed-memory support (based on ZeroMQ) recently released. FastFlow provides highly efficient, lock-free message-passing communication operations for shared-memory systems, as well as patterns (combinators) built on top of those. The patterns supported are similar to those in Avalanche, but configuration is done at run-time, requiring coarser-grained computations.

S-Net [2] is a coordination language (as is CnC) for describing flow graphs (of ordered streams) via composition combinators such as serial or parallel node composition and node replication. Unlike Avalanche, which is used entirely within C++, the S-Net coordination language is separate from the implementation language and includes a sophisticated type system for records. These record types map onto the host language, and a notion of *flow inheritance* solves the problem solved by our hidden-data operator.

Avalanche is a hybrid of the above options in that its edges fundamentally represent ordered streams of values; however, *redistribute* nodes are necessary for distributed programming and they can introduce reordering. The sensor network programming language, Regiment [36], has some similarities to Avalanche in its use of distributed (region) streams, which have local instantiations on each of many nodes.

CPS-Based Concurrency

The relationship between continuations and coroutines has long been recognized [21]. Recently, CPS transformations are increasingly used inside compilers to deal with asynchronous programming (i.e., to handle the generation of callbacks/event handlers). For example, both F# asynchronous workflows [42] and Scala use this technique; it has also been proposed as an extension to C++ [20]. This can be used to enable a natural, "straight-line" style of programming even in the presence of blocking, high-latency operations like fetching a Web site. In the case of the Akka actor library for Scala, messages to actors are essentially active messages, which, together with CPS support, is an attractive option. These actors are migratable and relatively expensive, however; they are targeted towards Web services rather than high-performance computing.

In contrast with these systems, Avalanche cannot provide CPS support in the compiler while using standard C++ compilers (without the adoption of [20]). Although Avalanche does require writing separate pipeline nodes, it enables rich forms of composition and messaging.

8. Conclusions and Future Work

We have presented Avalanche, a C++ library implementing a flow-graph-based interface on top of the Active Pebbles programming model. This library uses template metaprogramming techniques to optimize flow graphs while using an existing programming language and standard, un-modified compiler. In particular, it uses functional programming techniques such as continuation-passing style to allow fusion of successor graph nodes into their predecessors, as well as to provide stack allocation of complex structures of nodes. Uniqueness types are used to efficiently allocate objects in their parents when using a separate reference is too inefficient. Thus, functional programming is applied in a multi-paradigm language to achieve efficiency and expressiveness.

Several avenues remain for future work. The most direct is to add more pipeline combinators into Avalanche. One opportunity would be to define a combinator that takes a description of an arbitrary graph at compile-time and uses metaprogramming to link the flow graph nodes accordingly. Another would be to create nodes for distributing data evenly among a set of output nodes, rather than broadcasting it to all outputs. Threading support could also be added: one possibility would be a node that sends messages to its successor as tasks in a work-stealing system such as Cilk [9] or Threading Building Blocks [4], while another would be to keep the distributed-memory structure but add ownership-transfer intra-process messaging to AM++. Also, the current prototype requires side effects to communicate information out of subgraphs and loops; adding operations that allow the number of side effects to be reduced would be beneficial future work.

Although Avalanche is currently written in C++, a version in Haskell or another more traditional functional programming language could be implemented as well. A functional-language compiler might also be able to optimize the flow graph data structure and algorithms on it more automatically, without requiring information about the graph's structure and particular functions used in it to be encoded into its type. A pipeline fragment's input and output streams are appropriate to be part of its type since they control how a fragment can be used; a better compiler might be able to avoid the need to put other information into types. The arrow construct [24], and associated ***proc*** notation [39], is likely to be a elegant approach for describing pipelines in Haskell.

There is nothing limiting a system such as Avalanche to being built on top of AM++. Other models based on active messages, such as Charm++ [29] and ParalleX [27], could also be used underneath a flow graph system. Charm++ already has a domain-specific language, Structured Dagger [28], for specifying active message flows but a system such as Avalanche could be an alternative in pure C++. A flow graph model could be especially beneficial for ParalleX because it specializes in complex active message flows in which messages trigger other messages to separate systems; Avalanche would make specifying such flows much easier. Even non-active-message frameworks can be used underneath Avalanche; however, Avalanche's semantic model still would be based on streams of objects that have functions called on them when they arrive at flow graph nodes.

Acknowledgments

The authors wish to thank the anonymous reviewers for their helpful suggestions. This work was supported by NSF CCF-1111888 and CNS-0834722, as well as by the Lilly Foundation and Intel Corporation. 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.

References

[1] pipes Haskell package. URL http://hackage.haskell.org/package/pipes-1.0.

[2] S-Net project homepage. URL http://www.snet-home.org/.

[3] Introducing Crunch: Easy MapReduce Pipelines for Hadoop, 2012. URL http://www.cloudera.com/blog/2011/10/introducing-crunch/.

[4] Threading Building Blocks Reference Manual, 2012. URL http://threadingbuildingblocks.org/documentation.php.

[5] K. Agrawal, C. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IPDPS*, pages 1–12, Apr. 2010. doi: 10.1109/IPDPS.2010.5470403.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, 2012. ISBN 0470936908. URL http://www.di.unipi.it/~aldinuc/paper_files/2011_FF_tutorial-draft.pdf.

[7] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Workshop on Languages and Compilers for Parallel Computing*, pages 193–208, August 2001.

[8] H. G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *SIGPLAN Not.*, 30(9):17–20, Sept. 1995. ISSN 0362-1340. doi: 10.1145/214448.214454. URL http://doi.acm.org/10.1145/214448.214454.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/209937.209958.

[10] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar. Concurrent collections. *Scientific Programming*, 18(3–4):203–217, Aug. 2010. ISSN 1058-9244. URL http://dl.acm.org/citation.cfm?id=1938482.1938486.

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

[12] D. Carney, U. Centiemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *VLDB*, 2002.

[13] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Programming Language Design and Implementation*, pages 363–375, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806638. URL http://doi.acm.org/10.1145/1806596.1806638.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1094811.1094852.

[15] A. Culler, D. E. Culler, and K. Ekanadham. The price of asynchronous parallelism: an analysis of dataflow architectures. In *CONPAR*, pages 541–555, New York, NY, USA, 1989. Cambridge University Press. ISBN 0-521-37177-5. URL http://dl.acm.org/citation.cfm?id=90523.90636.

[16] J. Dennis. First version of a data flow procedure language. volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin / Heidelberg, 1974. ISBN 978-3-540-06859-4.

[17] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems*, pages 151–162, New York, NY, USA, 2006. ACM.

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

[19] A. Gurtovoy and D. Abrahams. The Boost C++ metaprogramming library. www.boost.org/libs/mpl, 2012.

[20] N. Gustafsson. Resumable functions. Technical Report N3328=12-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Jan. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3328.pdf.

[21] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3.4):143–153, 1986. ISSN 0096-0551. doi: 10.1016/0096-0551(86)90007-X. URL http://www.sciencedirect.com/science/article/pii/009605518690007X.

[22] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151013. URL http://doi.acm.org/10.1145/2150976.2151013.

[23] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), Dec. 1996. ISSN 0360-0300. doi: 10.1145/242224.242477. URL http://doi.acm.org/10.1145/242224.242477.

[24] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1–3):67–111, May 2000. ISSN 0167-6423. doi: 10.1016/S0167-6423(99)00023-4. URL http://dx.doi.org/10.1016/S0167-6423(99)00023-4.

[25] International Organization for Standardization. *ISO/IEC 14882:2011: Programming languages — C++*. Geneva, Switzerland, Sept. 2011. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[26] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, Aug. 1974. North Holland, Amsterdam.

[27] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *International Conference on Parallel Processing Workshops*, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3803-7. doi: 10.1109/ICPPW.2009.14. URL http://dx.doi.org/10.1109/ICPPW.2009.14.

[28] L. V. Kalé and M. Bhandarkar. Structured Dagger: A coordination language for message-driven programming. In *Euro-Par*, volume 1123–1124 of *Lecture Notes in Computer Science*, pages 646–653, Sept. 1996.

[29] L. V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/167962.165874.

[30] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *Architectural Support for Programming Languages and Operating Systems*, pages 233–243, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346311. URL http://doi.acm.org/10.1145/1346281.1346311.

[31] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009446. URL http://dx.doi.org/10.1109/TC.1987.5009446.

[32] D. Leijen and E. Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35(1):109–122, Dec. 1999. ISSN 0362-1340. doi: 10.1145/331963.331977. URL http://doi.acm.org/10.1145/331963.331977.

[33] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *International Conference on Management of Data*, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL http://doi.acm.org/10.1145/1807167.1807184.

[34] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, Sept. 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[35] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. du Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization*, pages 224–235, Apr. 2011. doi: 10.1109/CGO.2011.5764690.

[36] R. Newton, G. Morrisett, and M. Welsh. The Regiment macroprogramming system. In *Information Processing in Sensor Networks*, pages 489–498, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-638-7. URL http://doi.acm.org/10.1145/1236360.1236422.

[37] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. Design and evaluation of a compiler for embedded stream programs. In *Languages, Compilers, and Tools for Embedded Systems*, pages 131–140, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-104-0. URL http://doi.acm.org/10.1145/1375657.1375675.

[38] T. Ottosen and N. Groves. *The Boost.Range Library*. Boost, 2012. http://www.boost.org/libs/range/doc/index.html.

[39] R. Paterson. A new notation for arrows. *SIGPLAN Not.*, 36(10):229–240, Oct. 2001. ISSN 0362-1340. doi: 10.1145/507546.507664. URL http://doi.acm.org/10.1145/507546.507664.

[40] S. Salihoglu and J. Widom. GPS: A Graph Processing System. Technical report, Stanford University. URL http://ilpubs.stanford.edu:8090/1039/.

[41] S. Sogame. *Oven*, 2007. http://p-stade.sourceforge.net/oven/doc/html/index.html.

[42] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Practical Aspects of Declarative Languages*, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18377-5. URL http://dl.acm.org/citation.cfm?id=1946313.1946334.

[43] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. ISSN 1040-6042. Reprinted in C++ Gems, ed. Stanley Lippman.

[44] T. L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004. URL http://osl.iu.edu/~tveldhui/papers/2004/dissertation.pdf.

[45] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pages 256–266, New York, NY, USA, 1992. ACM. ISBN 0-89791-509-7. doi: http://doi.acm.org/10.1145/139669.140382.

[46] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358. Berlin: Springer-Verlag, 1988. URL citeseer.ist.psu.edu/wadler90deforestation.html.

[47] Z. Wan and P. Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5):242–252, May 2000. ISSN 0362-1340. doi: 10.1145/358438.349331. URL http://doi.acm.org/10.1145/358438.349331.

[48] S. Weeks. Whole-program compilation in MLton. In *Workshop on ML*, page 1, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. doi: http://doi.acm.org/10.1145/1159876.1159877.

[49] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. AM++: A generalized active message framework. In *Parallel Architectures and Compilation Techniques*, Sept. 2010.

[50] 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.