Declarative Patterns for Imperative Distributed Graph Algorithms

Marcin Zalewski, Nicholas Edmonds, and Andrew Lumsdaine
{zalewski,ngedmond,lums}@indiana.edu
Indiana University
Bloomington, Indiana, USA

***Abstract*—We provide an abstraction for expressing graph algorithms in which the vertices and edges of the graph provide locality and communication structure and graph data are represented by property maps that associate vertices and edges to arbitrary user-defined data. Operations on the graph are expressed as patterns, which allow limited traversal of the graph and modification of property maps for the traversed fragments of the graph. Traversal is implicit, and is automatically computed from the pattern's access of property map values. Patterns are declarative, but they can be used in imperative algorithms by using strategies that run in epochs. Strategies are user defined programs that apply patterns in a certain way (e.g., we provide fixed_point, once, and Δ-stepping strategies), including chaining patterns in an arbitrary way. Patterns are applied in epochs, which provide synchronization across a distributed system, guaranteeing that all patterns have been applied by the end of an epoch.**

I. Introduction

A growing and increasingly diverse group of scientific disciplines including bioinformatics, social network analysis, and data mining utilize graph analytics at HPC scales. The expected sizes of the graphs involved far exceed the capabilities of most non-distributed systems. For example, the Graph500 benchmark [1] classifies graphs with 2^{26} vertices and 2^{30} edges (around 17GB) as *toy* graphs, graphs with 2^{32} vertices and 2^{36} edges (around 1TB) as *small* graphs, and only graphs with 2^{39} vertices and 2^{43} edges (around 140TB) as *large* graphs. Of course, not every problem requires such a large graph, but extending the limits opens new frontiers in computation.

Unfortunately, while many graph problems are relatively simple to state and the algorithms to solve them are short and elegant, the distributed implementations of these algorithms are often morphed into a difficult-to-read spaghetti code of communication primitives and optimizations. Furthermore, while algorithms that solve the same problem may share their core operations and differ only in the order in which these operations are applied, it is often difficult to reuse the common parts because the communication primitives, synchronization, and scheduling of a particular version of an algorithm is deeply embedded with the general reusable parts.

Every graph algorithm establishes some invariant by a combination of an abstract pattern of access to the graph, expressed in terms of vertices, edges, and some properties attached to both. The pattern is combined with an imperative part of scheduling, external data structures, and other non-abstract algorithm "tricks" that make the algorithm run fast, take a little less memory than other algorithms, or any other such non-functional property. There are two extremes in coding distributed graph algorithms: for maximum performance, the algorithm can strive for maximum control over low-level details, and, for maximum abstraction, graph algorithms can be synthesized from some abstract statement of a problem, plugging in implementation details such as schedulers. We propose an approach that falls in between: graph access patterns are expressed as abstract specification that is synthesized into the appropriate communication automatically, but the patterns provide customization points where a specific algorithm can be plugged in to provide scheduling, additional data structures, and any other details. Using these customization points, we provide basic algorithms as reusable *strategies*, allowing for abstract expression of algorithms while still providing control over details when necessary.

Essential to our approach is the basic assumption that it is not predictable which parts of the graph are colocated on a distributed machine. We assume that, in general, one cannot lock parts of a graph in a convenient and reasonably cheap way. However, we do assume that the graph representation is vertex-centric, allowing coordinated access at any given vertex. Consequently, our language for expressing graph patterns provides only limited guarantees for synchronization, and we also limit the ways in which graph may be accessed to simplify the translation to underlying communication framework. However, we found out that many if not most non-morphing[1] graph algorithms naturally fit into such view.

The backend for our framework is AM++ [2], an implementation of the Active Pebbles programming model [3]. AM++ provides fine-grained messaging, which fits perfectly and was indeed designed for the model of a graph of communicating vertices. Messages in AM++ are similar to other active messages models, but there are no restrictions on message handlers sending new messages, a common limitation in other models. Furthermore, AM++ provides termination detection, allowing a convenient implementation of our approach where vertices communicate with each other through messages. The approach we propose could be implemented with other communication frameworks, but AM++ is our primary target because of its explicit design goals of supporting fine-grained communication. Eventually, we plan to try our approach out in the ParalleX [4], but this is the subject of future work.

Next, we demonstrate our approach by example in Sec. II, we give the semantics in Sec. III, and we outline the translation of abstract algorithms to AM++ in Sec. IV.

II. Algorithms by Pattern

We begin by an informal overview of our approach, by applying it to two fundamental graph problems: single-source shortest paths (SSSP) and connected components (CC).

[1] Non-morphing algorithms do not modify graph structure.

```
// INITALIZATION
for (v in V)
  dist[v] = ∞;
dist[s] = 0;
```

```
// Δ-Stepping
B[0].add(s); i = 0;
while (!B.empty()) {
  while (!B[i].empty()) {
    v = B[i].pop();
    for (e in out_edges(v))
      if (relax(e))
        B[dist[v]/Δ].add(trg(e));
  }
  i++
}
```

```
// RELAXATION
bool relax(e) {
  d = dist[src(e)] + w[e];
  if (d < dist[v]) {
    dist[v] = d;
    return true;
  }
  return false;
}
```

```
// Fixed–Point
WorkSet WS; WS.add(s);
while (!WS.empty()) {
  v = WS.get();
  for (e in out_edges(v))
    if (relax(e))
      WS.add(trg(e));
```

v: vertex, e: edge, s: source, dist: distance map, B: buckets

Fig. 1: Pseudocode for simplified Δ-stepping [6] and a fixed-point SSSP.

A. SSSP

Given a graph $G(V, E)$ and a source vertex $s \in V$, the SSSP problem is to compute for each vertex v reachable from s the weight of the minimum-weight path with the weight of a path defined as a sum of the weights of all the edges in the path. The algorithm for solving SSSP are divided into the *label-setting* (e.g., Dijkstra [5]) and *label-correcting* algorithms (e.g., Δ-stepping [6]). Both kinds of algorithms operate on the *distance* of every vertex in the graph, with the distance of 0 for the source s, the distance of ∞ (by convention) for every vertex not yet reached from the source, and some tentative distance d for every vertex reached from the source. SSSP algorithms *relax* edges (u, v), setting the distance of v to $\mathrm{dist}[v] = \min\{\mathrm{dist}[v], \mathrm{dist}[u] + \mathrm{weight}[(u, v)]\}$ enough times to establishes the invariant $\forall v \in V \setminus \{s\}\colon \mathrm{dist}[v] = \min_{(u,v)\in \mathrm{in\_edges}(v)}(\mathrm{dist}[u] + \mathrm{weight}[u, v])$. The main difference between algorithms is the order and the number of relaxations performed. Label-setting algorithms *settle* the final distance of one vertex at every step, while label-correcting algorithms continuously improve labels of groups of vertices.

In Fig. 1, we show a variant of the Δ-stepping algorithm [6] along with a simple fixed_point algorithm. Δ-Stepping divides pending vertices into numbered buckets B, where a bucket B[i] contains a subset of vertices $\{v \in V\colon \mathrm{dist}[v] \in [i\Delta, (i + 1)\Delta)\}$. The source vertex s is added to the first bucket in the initialization stage, and then the buckets are successively relaxed, with new vertices added to appropriate buckets when relaxation succeeds. Δ-stepping can contain more optimizations such as relaxing heavy edges, which cannot insert more work into the current bucket, separately from light edges, which may add work to the current bucket. The iterative algorithm, on the other hand, simply relaxes the relevant edges in an unspecified order until no more relaxations succeed. Importantly, the two algorithms share the relax function that establishes the SSSP invariant.

The algorithms are presented using simple pseudocode, yet it is clear that to implement them in a distributed environment, the programmer needs to devise the communication for the algorithms. For example, how is relax performed in the two SSSP algorithms across the nodes of the system? Furthermore, it's clear that the relaxation procedure should be reusable between the actual implementations of the two algorithms, but it may be difficult to make it so if communication of the algorithm must be written by hand.

Fig. 2 shows the SSSP *pattern* as expressed in our framework. The basic abstraction in our approach are *property maps*, which assign values to vertices and edges. The SSSP pattern declares two property maps, one storing the pending distance of a vertex and the other the weights assigned to edges. The SSSP pattern contains one action, relax. Every action starts at some vertex, named v in relax. Actions may have one generator that generates edges or vertices, given the starting vertex. In relax, one of the three built-in generators is used to get access to outgoing edges of the vertex v. The generator allows for "fanning out" from the starting vertex to other parts of the graph. Next, expressions can be given names in the aliases section for readability. Aliases are not variables but just shortcuts used to refer to expressions. Finally, the conditions section of an action consists of a series of if statements. In relax, there is only one action. The condition checks whether the distance to the target trg(e) of the generated edge e is greater than the distance discovered along the edge from v. If the condition is true, the distance at the target of e must be corrected to the newly discovered better distance.

```
pattern SSSP {
  vertex_property dist : Distance ;
  edge_property weight : Distance ;

  relax(Vertex v) {
    generator: e in out_edges(v);
    aliases:
      d is dist[v] + weight[e];
      dist is dist[trg(e)];
    actions:
      if(dist > d) dist = d;
  }
}
```

Fig. 2: SSSP pattern.

Such a pattern allows high-level expression of the basic action of SSSP. Expressions in a pattern can consist of arbitrary C++ code with the restriction that no vertices and edges can be produced by other means than generators and property maps and that the structure of the graph cannot be changed. We describe the semantics of patterns and the restrictions on actions in more detail in Sec. III.

Given the SSSP pattern, we can write algorithms. The basic fixed point algorithm can be written by using a the fixed_point pattern:

```
using pattern SSSP;

for (v in V) dist[v] = ∞;
dist[s] = 0;
fixed_point(relax, {s});
```

The code may seem like "cheating." Where is the work set? How is the fixed point reached? We provide the fixed_point strategy as part of our system (along with some other strategies). The strategy is implemented as follows:

```
strategy fixed_point(action a, container vertices) {
  a.work(Vertex v) = {a(v); }
  epoch {
    for(v in vertices) a(v);
  }
}
```

The fixed_point strategy takes an action from some pattern and a set of vertices to run it at. Then, it uses one of the customization points provided by our framework to handle *dependencies*

detected by our framework. For the SSSP pattern, it is easy to see that the condition of the action in relax both reads from the dist property map and writes to it. Thus, executing the action successfully (i.e., the condition was true) may require rerunning the action for the vertex where the value of distance has changed. Our framework detects such dependencies (the details are discussed in Sec. IV), and allows the client code to intercept them. The work property of the action on Line 2 specifies what to do when a dependency is discovered. The property allows intercepting the vertex at which the dependency occurred. In the case of the fixed_point strategy, the action a is immediately run on the vertex. Finally, the action a is invoked on all vertices given as action's input. The action is run in an *epoch*, which ensures that all work started directly in the action and indirectly in the work hook is finished before the strategy exits. We discuss how epochs are implemented in Sec. IV.

The fixed_point strategy can be replaced with the Δ-stepping strategy. The strategy for Δ-stepping is more involved:

```
1  strategy delta(action a, container vertices, property–map m, delta Δ) {
2    buckets B; i = 0;
3    for(v in vertices) B.insert(v, m[v], Δ);
4    a.work(Vertex v) = {B.insert(v, m[v], Δ); }
5    while (!B.empty()) {
6      epoch {
7        while (!B[i].empty()) {
8          Vertex v = B[i].pop();
9          a(v);
10       }
11     }
12     if(B[i].empty()) i++;
13   }
14 }
```

The strategy takes a property map m and the Δ step. The property map is used to provide numeric values to the insert function of the buckets data structure that, along with Δ, provide the information necessary to insert a vertex into the appropriate bucket. The first step of the strategy inserts the starting vertices into the appropriate buckets. Then, the work of the action is set to insert the vertices where dependencies are discovered into appropriate buckets. The main loop of the strategy empties buckets until no vertices are left to work on. The inner loop empties one bucket. It does that in an epoch, because the work resulting from ongoing actions may insert vertices into the bucket after it tests empty. Therefore, epoch must be used to finish ongoing actions, and the bucket has to be tested again.

These examples demonstrate, in pseudo-code, the capabilities of our framework. The important ingredients are:

- **Patterns** that capture operations on the graph.
- **Strategies** that apply patterns in a specific way. Strategies rely on primitives provided by our framework, including *epochs* for applying patterns and performing all resulting work, *action work specification* that allows specific handling of work that results from automatically computed dependencies.

The actual strategies are more involved than shown in this section, providing complex data structures, managing multithreading, and so on. The Δ-stepping strategy, for example, has to provide a thread-safe buckets data structure. However, the primitives provided by our system remain simple. For example, epochs are provided across a distributed system with multiple ranks, each running multiple threads.

B. CC

The CC problem is one of the most fundamental problems for graphs. The task is to find groups of vertices in an undirected graph connected to each other by paths: if two vertices are connected by a path, they belong to the same component. Since all vertices in any path belong to the same component, the CC problem is to establish the invariant $\forall v, u \in V, \mathrm{adj}(u, v) \colon \mathrm{comp}[u] = \mathrm{comp}[v]$. There are many different algorithms (see [7] for a comparison of a few popular algorithms) for CC. Here, we present a parallel search algorithm to expose one additional feature of our framework that is not exercised by the SSSP algorithms in the previous section.

```
1  search(v) {
2    for (e in out_edges(v))
3      if(p[trg(e)] == NULL) {
4        p[trg(e)] = p[v];
5        search(trg(e));
6      }else conflicts.add(
7            p[trg(e)], p[v]);
8  }
9  // Parallel Search
10 conflicts = ∅;
11 for (v in V) p[v] = NULL;
12 for (v in V)
13   if(p[v] == NULL)
14     search(v);
15 rewrite_conflicts();
```

Fig. 3: Parallel search algorithm.

The basic parallel search algorithm is shown in Fig. 3. It is similar to the "well-known" [8] CC algorithm where components are found by depth-first or breadth-first search starting sequentially from the set of vertices that do not belong to any component. The parallel search version does the same, but it starts multiple searches in parallel, recording a conflict if two searches collide. In the final stage of the algorithm, when no vertices are left without a component assigned, the component numbers are efficiently rewritten. Rewriting does not require traversing the graph, and can be done solely on the component labels if the labels are ordered. Such algorithm is implemented in the Parallel Boost Graph Library [9].

```
1  pattern CC {
2    // Component parent
3    vertex_property prnt : Vertex;
4    // Component rewrite
5    vertex_property chng : Vertex;
6    cc_search(Vertex v) {
7      generator: u in adj(v);
8      aliases:
9        pv = prnt[v];
10       pu = prnt[u];
11       cpv = chng[pv];
12     actions:
13       if(pu == NULL)
14         pu = pv;
15       else if(pu < pv &&
16               pu < cpv)
17         cpv = pu;
18   }
19   cc_jump(Vertex v) {
20     aliases:
21       pv = chng[v];
22       ppv = chng[pv];
23     actions:
24       if(ppv < pv) pv = ppv;
25   }
26 }
```

Fig. 4: SSSP pattern.

Fig. 4 shows the pattern for a version of the parallel search algorithm. The pattern operates on two component maps, prnt (parent) to store component parents of vertices and chng (change) to store component rewrites. The cc_search action examines all neighbors of a vertex and (through the adj generator), and, in the first condition, it spreads the component of v to its neighbors that do not have a component assigned (NULL component). The second condition, evaluated only if the first one was false, records a conflict if a neighbor already has a "better" component (measured by total order on vertices). The existing rewrite is checked also to verify that the component is not already rewritten to a better component (the NULL component is assumed to be "the worst" component, and NULL should be the initial value of chng for all components). The

pseudo code for the complete algorithm uses the two patterns:

```
1  using pattern CC;
2  for(v in V) {
3    prnt[v] = NULL; chng[v] = NULL;
4  }
5  cc_search.work(Vertex v) {cc_search(v); }
6  epoch {
7    for(v in V)
8      if(prnt[v] == NULL) {
9        prnt[v] = v;
10       cc_search(v);
11       epoch_flush();
12   }
13 }
14 while(true) {
15   vs = {v in V | chng[v] != NULL};
16   if(!once(cc_jump, vs)) break;
17 }
18 rewrite_cc();
```

First, the property maps are initialized (Lines 2–4). Then, the work for the cc_search is changed to recursively apply the cc_search pattern, just like in the fixed_point strategy. After initialization, the cc_search pattern is applied in a loop in an epoch on Lines 6–13. That implementation may look strange, but we assume that invoking an action on Line 10 followed by epoch_flush, a primitive provided by our system, performs as much work as available. So, searches are started possibly on multiple nodes of a system, and, after each search is started, the system tries to perform as much work as possible when epoch_flush is called (the actual amount of work that will be performed depends on implementation, but we assume a good enough effort to perform enough work) before starting the next search. In the distributed setting, starting too many searches may lead to many remote accesses to record component conflicts, but it is important to note that our implementation based on AM++ allows *reductions*of unnecessary communication. For component collisions, the remote messages generated for updating of component rewrites are intercepted at the local node and compared to a cache of messages that were already sent. If a message with the same or a better rewrite is present in the cache, then the update does not have to be sent. Currently, reductions have to be specified by hand, but, in principle, reductions could be discovered for many classes of operations by analyzing patterns.

After the parallel searches are finished and conflicts are recorded, the final component rewrites are computed on Lines 14–17 by repeatedly applying the cc_jump action using the once strategy, provided as a part of our system. The once strategy performs an action at every vertex in the input set, recording if any assignments to property maps were performed. The cc_jump action checks every component root vertex to be rewritten if the rewrite target vertex is to be rewritten itself. If the target vertex is being rewritten to a "better" vertex, then the rewrite target is changed to that better vertex (pointer jumping, thus the name cc_jump). Finally, after all the pointers are updated, the components are rewritten by a rewrite_cc function that we do not specify here. This function has to simply rewrite component roots for all vertices based on the values in the chng property map, but since it is not a graph computation it is not particularly interesting in this context.

III. Semantics

Patterns express computations on directed graphs $G(V, E)$, where V is the set of vertices of type Vertex and E is a set of edges of type Edge. A pattern is a collection of vertex and edge property maps and of actions that can operate on these property maps. Every action begins at some vertex v, which is the input of the action. An action can only access vertices and edges through a limited set of generators and as values of property maps. An action consists of a sequence of conditions, each of which guards some property maps assignments (or, more generally, property map modifications).

⟨*pattern*⟩ ::= '`pattern`' ⟨*name*⟩ '`{`' ⟨*properties*⟩ ⟨*actions*⟩ '`}`'

A. Computational Model

We assume a distributed graph, where every node stores a portion of vertices and their outgoing edges. A bidirectional-graph, where "bidirectional" describes the storage model rather than a property of the graph, also stores incoming edges with a vertex. Nodes can apply further subdivision of work, for example, through threading. Our approach of using patterns does not explicitly depend on any of these assumptions, but it is designed to accommodate such a setup by limiting which vertices and edges can be used in any given computation. The implementation of our approach in AM++, described in Sec. IV, requires explicit treatment of system nodes and threads.

B. Property Maps

Property maps are the fundamental idea behind our approach. A property map associates vertices or edges with arbitrary values, including vertices and edges. Properties are specified according to the following grammar:

⟨*property*⟩ ::= ⟨*property-kind*⟩ ⟨*name*⟩ '`:`' ⟨*type*⟩ '`;`'
⟨*property-kind*⟩ ::= '`vertex-property`' | '`edge-property`'

Property kind specifies whether a map stores values for vertices or edges, and type describes the values stored in the property map. Accessing a property map is done by indexing the map with a vertex or an edge, depending on the property kind.

C. Actions

Actions describe the computation originating from a "source" vertex, which is the input to the action. Actions have the following form:

⟨*action*⟩ ::= ⟨*name*⟩ '`(`' '`Vertex`' ⟨*name*⟩ '`)`'
'`{`' ⟨*generator*⟩? ⟨*aliases*⟩* ⟨*condition*⟩ '`}`'

An action always takes one vertex argument, which can be given a name of choice (in our examples v). Then, an action has zero or one generators (signified by '`?`'), zero or more aliases, and at least one condition.

A generator allows the action to "fan out" from the origin vertex by accessing vertices or edges from a set obtained from property maps or from one of the three built-in sets adj, out_edges, or in_edges:

⟨*generator*⟩ ::= '`generator:`' ⟨*name*⟩ in ⟨*set-expr*⟩
⟨*set-expr*⟩ ::= ⟨*pmap-access*⟩ | ⟨*built-in-set-access*⟩
⟨*built-in-set*⟩ ::= '`in_edges`' | '`out_edges`' | '`adj`'

The type of ⟨*name*⟩ depends on the type of the elements of the set generated by ⟨*set-expr*⟩, which must either be Vertex or Edge. The set can only be obtained by accessing a property map

indexed by the origin vertex of the action or by using one of the built-in graph traversal functions returning in edges, out edges, or adjacent vertices.. The ⟨*name*⟩ used in the generator can be used in the remainder of the action. It is important to note that there can be only one generator, allowing only one level of "fan out." In general, there could be more generators, but most graph algorithms do not seem to need them, and multiple generators would greatly increase computational complexity of actions.

Aliases are simple shortcuts for expressions. For example, instead of writing prnt[v], the expression can be named as it is in Fig. 4. Aliases are not variables, and using an alias is the same as pasting in the expression it stands for. Aliases are simply a convenience feature, and have no impact on the semantics of actions.

Finally, the most important part of an action are the conditions. Conditions are essentially a chain of C++-like if-else statements where the boolean expressions must involve accessing property maps. The expressions can be arbitrary C++ expressions without side effects that modify property maps or the graph itself. Each if-else statement body can contain several modifications of property maps, where one property map is written to. Our definition of modification is purposefully vague, and we decide which property map value is modified by a simple rule of analyzing each modification expression from left to right, and treating the leftmost value as being modified and all other property map values as being read. This allows modifications such as the following:

```
1 property–map preds : std::set<Vertex> ;
2
3 // Somewhere in an action:
4 preds[v].insert(u);
```

In the above example, the preds (predecessors) property map stores a set of vertices, and a modification requires using the set interface.

The reads and modifications *are not* guaranteed to be synchronized in all but two cases. First, every modification, such as the one above, is guaranteed to be atomic. Thus, it is safe to call the insert function on the set of vertices. Furthermore, in every condition, the first modification is guaranteed to synchronize the reads of property values indexed with the same vertex that the modified property map value is indexed with. For example, in Fig. 2, the reading of dist in the condition is guaranteed to be synchronized with writing of dist in the modification statement. Writing of dist is not synchronized with the reading of dist[v], however, since v is a different vertex than trg(e). Basically, the rule of thumb is that the condition and the first modification will be evaluated at the same vertex (in terms of locality) and writing and reading of property maps at this vertex will be synchronized using some shared memory technique such as locking or atomic instructions.

An action modifies some property map values if any of its conditions evaluates to true. If an action not only modifies but also reads this value in one of its conditions or modification expressions, the vertex at which the property map value is modified is marked as dependent, and a work item is created for that vertex. The dependent vertex can be accessed by an algorithm by modifying the work hook provided by every action. By default, dependencies are simply ignored. In the fixed_point strategy, for example, the action is repeated for every dependent vertex.

D. Epochs

Patterns specify what to do at a single vertex. An algorithm applies patterns to a distributed graph, often following dependencies produced by each application of a pattern. Our system provides *epochs* to run a number of patterns and all their dependencies in a distributed setting. An epoch finishes (on all nodes, threads, and other parallel constructs used) only when all actions that were invoked and their dependencies have finished. Epochs are necessary coarse-grained synchronization mechanism for the fine-grained abstraction of actions. Indeed, epochs are also the basic concept in AM++, which we use to implement our approach.

An epoch is a scoping construct that can contain arbitrary code. In the implementation of the fixed_point strategy, for example, the input action is called on the input set of vertices within an epoch, and the only further work is done implicitly by the system following the dependencies of actions. In the delta strategy, a while loop is run within the epoch to empty a bucket and to start all the actions for the current level. In the implementation of CC parallel search algorithm, all vertices are traversed and the search action is called for all NULL vertices. The CC search algorithm uses the epoch_flush primitive to perform as many dependent actions as possible before moving on. We provide two such primitives:

- epoch_flush to perform as much work as possible with a reasonable system load, and
- try_finish to try to finish an epoch, exiting if no more actions are pending anywhere on the system.

The crucial difference between the two primitives is that epoch_flush only tries to perform as much work as possible and then hands the control to the calling code, and try_finish may move the control to the code outside of an epoch if there is no more work left. We have not shown how try_finish can be used, but it is useful for algorithms without coarse-grained synchronization. For example, we have implemented a distributed version of Δ-stepping where every thread on every node has its own local buckets. When a thread runs out of work locally, it tries to terminate the epoch, which succeeds if all other threads everywhere also run out of work and if no actions are still to be performed. If ending the epoch is unsuccessful, however, the thread goes back to its local bucket structure and tries to perform more work (its buckets can be filled while it tries to end the epoch).

IV. Implementation

Our framework for graph computation is based on the Active Pebbles [3] model implemented in AM++ [2](from now on, we refer to AM++ for both the Active Pebbles model and the AM++ implementation). The basic abstraction of AM++ are *active messages*, which are pieces of data that can be sent to any node of the system, and for which a *handler* is invoked. AM++ has several unique characteristics which, by design, make it particularly suitable for graph computations:

- Several message types can be created with arbitrary functions allowed as handlers. Handlers are statically typed, and

are the C++ mechanisms allow optimizations not possible with function pointers in more dynamic implementations.

- Handlers are not limited as in many other active message systems and can perform arbitrary computations and send any number of active messages.
- Messages are sent to nodes, and handlers are run per node. However, an object-based addressing scheme can be used to automatically compute the destination node from a message payload.
- AM++ provides built-in layers for message coalescing and caching. Coalescing greatly improves performance when large amounts of messages are sent (true for most large-scale graph computations). Caching allows to avoid unnecessary message sends and the corresponding handler calls in algorithms that produce potentially large amounts of repetitive work.
- Finally, one of the most important AM++ features for our system is *termination detection*. The epochs in our system map directly to AM++ epochs.

In the distributed setting of AM++, a vertex can be located at any node. In our implementation, all the outgoing and incoming edges are located on the same node as are the corresponding vertex and edge property values. Reading from and writing to property maps must be done at the nodes where the values are located. For our system, that implies that patterns must be translated to appropriate messages, where reading and writing of property maps is done in message handlers. Since different vertices involved in an expression may be located at different nodes, the messages need to gather the values necessary to evaluate an expression by forwarding the values in message payloads. When all the values necessary for evaluating an expression (e.g., a condition in an action) are gathered, the expression can be evaluated. Furthermore, because of this non-local gathering process, no locking is provided by the system beyond a single handler. Next, we outline how the AM++ messages are created for a pattern. Then, we discuss the implementation of locking and further implementation concerns that are not handled directly by our system.

A. Messages

An action consists of conditions and modification statements, each having access to the following:

- the input vertex (referred to by v from now on),
- vertices or edges produced by a generator (vertices referred to as u and edges as e from now on), and
- values retrieved from property maps.

Generators are simply loops run at the beginning of communication, repeating the communication as described below for every generated value. Two kinds of messages must be constructed for every condition and modification statement in a pattern:

- *gather messages* that traverse the values that are read in condition tests or modify statements, and
- *evaluation messages* that perform the actual condition tests and modifications.

Generation of these messages is based on constructing a dependency graph of these values where dependency is decided by locality.

Definition 1 (Locality). The locality of any value used in a pattern is described by the vertex that it is accessed at. The locality of the input vertex v, the generated edges e, and of the generated vertices u is the vertex v. The locality of a vertex or edge property access of the form p[x] is x if x is a vertex, and the locality of x if x is an edge. The locality of the special functions trg and src (target and source of an edge) is the locality of the edge they are applied to.

Definition 2 (Dependency Graph). The dependency graph stores dependencies between values. A directed edge (v1, v2) is added to the graph between the values v1 and v2 if v1 is the locality of v2.

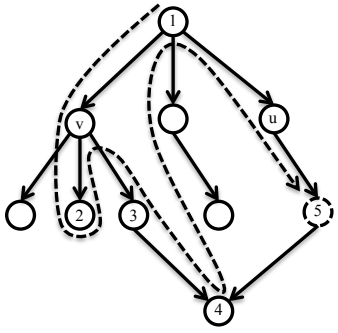

Fig. 5: An example of gather messages. The dashed line shows the most optimal order of gathering, and the dashed vertex shows the evaluation site.

To gather all the values necessary for evaluation of a condition, the graph needs to be traversed in some depth-first order. Fig. 5 shows an example where vertices 1 through 5 are necessary for evaluation. The values are collected in a depth-first traversal of the communication graph going through vertices 1, v, 2, back to v, then 3, 4, back to 1, then through u to 5. In this case it would be more efficient to proceed straight to vertex 3 from 2, without backing up to v. While this is indeed what we would do in practice, it can be considered an optimization, and it complicates the presentation of the algorithm. In Fig. 5, every "jump" between vertices corresponds to a message, totaling in 8 messages in this case. Each consecutive message's payload adds the values retrieved at every vertex on the traversed path, until all values are available at vertex 5. In general, given the exact associativity of all operations in an expression (e.g., through explicit parenthesizing or through the knowledge of associativity properties of operations), precomputed values of subexpressions can be carried in messages instead of all the components. Again, this is an optimization, and an important one for that matter. We do not discuss this optimization here, but we do apply it in our system.

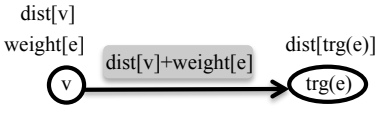

Fig. 6: One-message communication for the SSSP pattern (message content shown in gray).

While Fig. 5 is meant to show a rather general case, many actual algorithms are much simpler. Fig. 6 shows the communication graph for the SSSP pattern from Sec. II-A. The two values necessary to compute the new distance, dist[v] and weight[e], are local to the input vertex v. The subexpression dist[v] + weight[e] is precomputed at vertex v, and then sent as the payload of the message that computes the condition and performs the corresponding assignment when the condition is true at the vertex trg(e).

The general procedure for generating the communication for an action is to repeatedly apply the following procedure to all conditions:

1) Generate all the gather messages necessary to evaluate the condition.
2) Generate the evaluate message for the condition.
   a) If the condition evaluates to true, the evaluate message begins communication for its modification statement. The last modification statement begins the communication for the next non-else condition.
   b) If the condition evaluates to false, the communication for evaluation of the next condition is initiated.

To generate messages for a condition, the following steps are taken:

1) The localities required for evaluating the condition are found by analyzing all property map accesses.
2) The depth-first communication tree is pruned of edges that are not contained in a path to a required locality.
3) Gather messages are constructed using the pruned communication tree by traversing the tree in depth-first manner. Every message increases the payload of the previous message by the necessary values.
4) The final evaluate message is constructed to evaluate the condition.

The messages for modification statements are constructed similarly. However, before messages are constructed, modification statements are grouped together by the locality of values they modify. Still, the modifications are not reordered, so if modifications of values at different localities are interleaved, they will not be grouped. Furthermore, when the first group of modifications accesses values at a subset of localities of the localities accessed by condition, the group of modifications is merged into the condition. Instead gathering the values at the locality of the value to be modified, the condition gather message only forwards the locality itself. After all the other values have been gathered, a merged condition evaluation and value modification message is sent to the locality of the modified value. Then, the values with the final locality are read and the condition is evaluated. The same message handler performs the modifications if the condition evaluated to true. This is not a mere optimization, but, together with synchronization, this merging allows to ensure consistency of reads and writes of the modified value. For example, in the SSSP pattern as shown in Fig. 6, the evaluation of the condition and the modification happen at the same time at vertex trg(e), allowing the synchronization necessary for updating the distance. We perform similar merging between modification statements and the following conditions. If last the modification statement accesses the localities that are necessary for the following condition, the gather messages for that condition are elided. Similarly, if the previous condition is false, the next condition is evaluated right away if all the necessary values are available. The basic idea is to, first, save as much communication as possible, and, second, to allow as much synchronization as possible without distributed locking and additional communication, by evaluating consecutive conditions and modifications on the same node if the same values are accessed.

B. Synchronization

Synchronization is difficult to achieve on a distributed system. That's why we avoid built-in synchronization. However, many algorithms, such as our SSSP algorithms, are possible with synchronization on a single node. We guarantee synchronization on a single node when values are modified. If only one value is read and written, as in the SSSP relax pattern (communication shown in Fig. 6), synchronization is performed by atomic instructions where supported. If more than one value is accessed, synchronization is performed by locking.

The synchronization primitives are implemented through a *lock map* abstraction. The lock map has an interface for requesting a lock and for atomic instructions on property maps for the single-value case. Using generic programming techniques , we check if atomics are supported for a given type, and we revert to locking when they are not. The lock map abstraction allows to parameterize an algorithm by a locking scheme. Two examples of possible locking schemes are a single lock per vertex or a lock for a block of vertices, with a tradeoff between the coarseness of synchronization and the number of locks.

C. Dependencies

Dependencies allow fixed-point algorithms. In our current design, dependencies are implemented in a simple way. If a value is read anywhere in an action and the value gets modified in one of the conditions, the work hook specified by the programmer is called (e.g., in the fixed_point strategy in Sec. II-A).

D. Addressing

AM++ requires a node address for every message (see [3] for details). However, the address does not have to be provided explicitly. Instead, an address map can be provided for every message type for AM++ to use to compute the node address. The basic addressing is provided by the graph for vertices, where the node of a vertex can be obtained from the graph. This addressing can be used to derive node addresses for every message type that contains a vertex that is the destination of the message. In our system, every message contains the locality, which is a vertex, for which the message is destined, and address computation maps are automatically generated along with message types. The address maps are stateless, and simply extract the destination vertex from a message, computing the address from the graph address map.

V. Related Work

The Elixir system for synthesizing parallel graph programs [10] is perhaps the most similar to our work. An Elixir graph, similarly to our graphs, can have node and edge properties. The graph algorithm is a collection of rewrite patterns that match the shape and the properties of a graph fragment, and specify how those properties are to be rewritten, using Elixir's scheduling constructs. The actual program is automatically generated using work lists from the Galois system [11]. In contrast to Elixir, our system is designed for distributed graphs. Our patterns are similar to the Elixir patterns, but they always begin at a single vertex and they allow multiple conditions and corresponding rewrites. Instead of automatically generating programs, our

system generates communication, but the program has to be written explicitly, with the help of strategies.

Pregel [12] is an example of a bulk-synchronous framework for large-scale graphs. Pregel abstraction is expressed in terms of vertex programs that receive messages from other vertices at the beginning of a superstep, and send messages to other vertices at the end of superstep. The major difference between our system and Pregel is that Pregel provides a view of a single vertex only. Computations that require communication spanning multiple vertices, such as pointer jumping in our CC pattern, are more difficult to express given a single vertex view. On the other hand, Pregel provides an API for graph mutation, which our framework currently does not.

Distributed GraphLab [13] is an asynchronous system. It is also vertex-based, but instead of messages, computation is expressed through update functions. An update function $f(v, S_v) \to (S_v, T)$ gets vertex v and its scope S_v as input. The scope provides a consistent view at the vertex and its immediate neighbors. The output T is a set of vertices for which the update function should be eventually executed where, in general, the system is free to decide the order of execution of update functions. Just like Pregel, this model is limited to a vertex and its immediate neighborhood. The GraphLab model, however, provides a consistent view of that neighborhood making it easier to reason about correctness of an algorithm.

Combinatorial BLAS [14] is another, unique approach for expressing graph computation. Instead of directly operating on graphs, programs are expressed in terms of operations on sparse matrices. Breadth-first search, for example, is implemented as the repetitive multiplication of the transpose of the adjacency matrix of the graph with a rectangular matrix representing the current front of the search. In contrast to our work, the primary goal of [14] is to improve scalability of graph framework implementation, but it still (perhaps not incidentally) provides a BLAS based abstraction. The major difference between ours and theirs abstractions is that the BLAS abstraction is intended to uncover coarse-grained parallelism, while our abstraction uncovers fine-grained parallelism.

VI. Conclusions and Future Work

We have presented an abstraction for expressing graph computations in a distributed environment, based on the *graph as the communication network* of vertices and directed edges with data associated to the parts of the graph through *property maps*, with graph programs are expressed through *patterns* and *strategies* used to invoke patterns in imperative programs. Patterns can examine and modify the property maps, traversing the graph by using *generators* and vertex or edge values stored in a property map; generators provide the initial "fan out" capabilities, while property map vertex and edge value allow irregular communication necessary for a given algorithm. Patterns can be invoked in distributed setting, and each pattern execution may result in arbitrary invocation of other patterns. To manage such unstructured application of patterns, we provide *epochs*, which provide synchronization for patterns across nodes. Altogether, these elements allow for succinct declarative expression of the operations on the graph and the graph data along with imperative *support programs* for improving performance (e.g., delta stepping) or performing non-graph computations (e.g., final component rewriting in Fig. 3).

Our system is based on the implementation and active message ideas from AM++, providing an abstraction specifically designed for graphs and an AM++ implementation. Currently, our approach is not fully automated, and it requires manual translation of patterns into AM++ messages and constructs. We plan to implement a translator for patterns that will at least generate AM++ messaging code. Full integration with algorithm code into fully functional algorithm implementations will be more difficult just because of the need for an interface between the generated code and the code that is provided separately. We also plan to experiment with more algorithms to check if the current abstraction is powerful enough to express a variety of problems. One of the main extensions we already know to be necessary is graph mutation operations that are necessary for some graph algorithms. Also, we are currently investigating graph algorithms in the new emerging ParalleX model [4], and integrating our abstractions with that model will show whether our abstraction is general enough to move between underlying distributed computation models or if changes are necessary.

References

[1] "The graph 500 list," http://www.graph500.org/, Jun. 2013.

[2] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine, "AM++: A Generalized Active Message Framework," in *Proc. 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010, pp. 401–410.

[3] ——, "Active Pebbles: A Programming Model for Highly Parallel Fine-Grained Data-Driven Computations," in *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*. ACM, 2011, pp. 305–306.

[4] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications," in *Proc. 38th International Conference on Parallel Processing Workshops*, 2009, pp. 394–401.

[5] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[6] U. Meyer and P. Sanders, "Δ-stepping: A Parallelizable Shortest Path Algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, Oct. 2003.

[7] J. Greiner, "A Comparison of Parallel Algorithms for Connected Components," in *Proc. 6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM, 1994, pp. 16–25.

[8] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

[9] N. Edmonds, D. Gregor, and A. Lumsdaine, "Parallel Boost Graph Library," http://www.boost.org/libs/graph_parallel, Jun. 2013.

[10] D. Prountzos, R. Manevich, and K. Pingali, "Elixir: A system for synthesizing concurrent graph programs," in *Proc. ACM International Conference on Object-Oriented Programming Systems Languages and Applications*. ACM, 2012, pp. 375–394.

[11] "Galois system," http://iss.ices.utexas.edu/?p=projects/galois, Jul. 2013.

[12] 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 *Proc. ACM SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 135–146.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed Graphlab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[14] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.