Families of Graph Algorithms: SSSP Case Study

Thejaka Amila Kanewala[1,2(✉)], Marcin Zalewski[2], and Andrew Lumsdaine[2,3]

[1] School of Informatics and Computing, Indiana University, Bloomington, IN, USA
thejkane@indiana.edu

[2] Pacific Northwest National Laboratory, Seattle, WA, USA
{marcin.zalewski,andrew.lumsdaine}@pnnl.gov

[3] University of Washington, Seattle, WA, USA

Abstract. Single-Source Shortest Paths (SSSP) is a well-studied graph problem. Examples of SSSP algorithms include the original Dijkstra's algorithm and the parallel Δ-stepping and KLA-SSSP algorithms. In this paper, we use a novel Abstract Graph Machine (AGM) model to show that all these algorithms share a common logic and differ from one another by the order in which they perform work. We use the AGM model to thoroughly analyze the family of algorithms that arises from the common logic. We start with the basic algorithm without any ordering (Chaotic), and then we derive the existing and new algorithms by methodically exploring semantic and spatial ordering of work. Our experimental results show that new derived algorithms show better performance than the existing distributed memory parallel algorithms, especially at higher scales.

Keywords: Single-source shortest paths (SSSP) · Distributed-memory graph algorithms

1 Introduction

Given a graph problem, how many ways can it be solved in? In this paper, we consider the seemingly simple problem of *single-source shortest paths* (SSSP), where the task is to find the shortest path from a source vertex s to every other vertex in the graph. A number of sequential algorithms exist. The well-known Dijkstra's algorithm [3] is "work optimal", where vertices are ordered in a priority queue based on their distance from the source s, and every edge is traversed only once. Work optimality, however, comes at a cost of limited parallelism and extensive synchronization. Subsequent development concentrated on relaxing the strict ordering of the Dijkstra algorithm to make more work available in parallel at the cost of some "wasted work" that has to be invalidated and repeated. For example, the Δ-stepping [9] algorithm groups vertices into Δ-sized *buckets*, based on their distances from the source s, giving an approximation of Dijkstra ordering. Vertices in a bucket are processed in parallel, and picking an appropriate Δ ensures the right balance between parallelism and wasted work.

DOI: 10.1007/978-3-319-64203-1_31

The *KLA-SSSP* [6] algorithm is similar, but it uses topological distances instead of shortest path distances from the source s to order work into buckets[1].

Table 1. Orderings in SSSP algorithms.

Algorithm	Ordering
Dijkstra's	Global priority queue
Δ-stepping	Global distance equivalence classes defined by Δ
KLA	Global topological distance equivalence classes defined by k
Chaotic	None

Algorithm 1. The SSSP relax function

```
1: **Input:** Task $(v, d)$, distances $D$
2: **if** $d < D(v)$ **then**
3:    $D(v) \leftarrow d$
4:    $\forall v_n \in \text{neighbors}(G, v)$ :
5:       $\text{Task}(v_n, d_v + \text{weight}(v, v_n))$
```

In both Δ-stepping and KLA-SSSP, processing of the buckets inserts implicit synchronization points, since processing of a bucket cannot begin until all previous buckets are finished. The *Chaotic SSSP* does away with synchronization altogether by processing all the vertices in parallel in an arbitrary order, resulting in maximum available parallelism at the cost of more wasted work.

Despite the variety of algorithms, analysis reveals that they are all based on the same core logic of *relaxation*, as shown in Algorithm 1. Relaxation takes as the input a vertex-distance pair and a distance map (D), and produces more vertex-distance pairs if the distance was improved. These newly produced pairs are further relaxed, and the algorithms differ by how these relaxations are ordered (Table 1). In this paper, we methodically investigate this similarity between the seemingly different SSSP algorithms. To do that, we model the algorithms using the *Abstract Graph Machine* (AGM) [7]. An AGM represents a graph algorithm as two distinct components: the *processing function* that models the core functionality of the algorithm and an ordering of the work tasks that define the characteristics of the algorithm (as in Table 1, for example). The work ordering relation of an AGM is defined based on one or more attributes of work tasks. For the SSSP algorithms, it can be the distance in vertex-distance pair, or it can be an additional attribute introduced specifically for a given algorithm (see the next section for details). The work ordering relation is a *strict weak ordering*. It divides work into ordered *equivalence classes*, where work within an equivalence class is unordered and can be executed in parallel, but work within separate equivalence classes is executed according to an ordering induced by the strict weak ordering relation.

We present AGM models for all the algorithms listed in Table 1, and we show that they change by the way in which they order work. Then we show that new algorithms can be developed by methodically discovering new orderings. We introduce *extended AGM* (EAGM) that incorporates information about spatial distribution into algorithms modeled in AGM. With EAGM, we develop

[1] KLA-SSSP with single-hop buckets is equivalent to the Bellman-Ford algorithm [1].

variations of algorithms presented in Table 1 with additional ordering at different spatial levels of architecture such as node (process), numa (non-uniform memory access) region, and thread, resulting in *nine* different SSSP algorithms. We compare the weak scaling performance of the new algorithms with existing distributed memory parallel algorithms and also with the SSSP algorithm in PowerGraph [5] and in Parallel Boost Graph Library (PBGL) [4] for a performance base line. Our results show that some of the new variations of SSSP algorithms perform better than the well-known algorithms, especially at large scales.

In summary, our main contributions are generalizing SSSP algorithms using the AGM formulation, an approach to generate variations of primary distributed SSSP algorithms listed in Table 1 using EAGM, and experimental evaluation showing that algorithm variations generated by EAGM specification perform better compared to the well-known algorithms listed in Table 1.

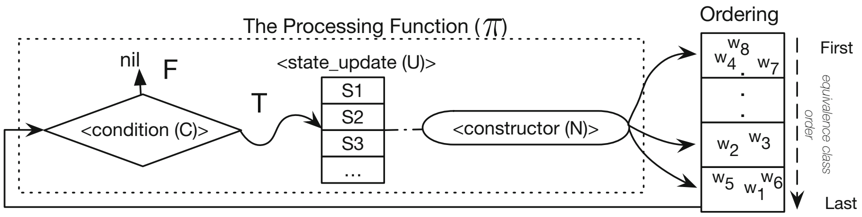

Fig. 1. An overview of the Abstract Graph Machine execution.

2 Abstract Graph Machine (AGM)

The *Abstract Graph Machine* (AGM) approach captures the core logic of an algorithm and the semantic work ordering that impacts the performance of the algorithm. Such principled approach allows discovery of families of algorithms by varying work ordering where it affects the performance but not semantics of an algorithm. In this section, we introduce AGM framework, and in Sect. 3 we apply it to the SSSP algorithms.

At the heart of AGM is the *processing function* that captures the logic of an algorithm. A processing function takes a single *workitem*, the smallest unit of work performed in the algorithm, and it generates zero or more new *workitems* from the input *workitem*. The processing function can access the graph and per-vertex and per-edge state when computing new *workitems*. The set of all the *workitems* generated by an algorithm is denoted using *WorkItem*. In other words, the *WorkItem* represent all the *workitems* generate during the whole lifetime of the algorithm's execution. The order of execution of *workitems* generated by processing functions is dictated by a strict weak ordering relation defined on the *WorkItem*. Figure 1, shows an overview of the AGM. An AGM consists of a definition of a *Graph*, a definition of a *WorkItem* set, a set of *states*, a *processing function*, a *strict weak ordering* relation, and of an *initial workitem set*.

Graph Definition. The graph definition takes the form, $G = (V,\ E,\ vmaps,\ emaps)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. *vmaps* is a set of functions each of the form $f : V \longrightarrow X$, and *emaps* is another set of functions each of the form $f : E \longrightarrow X$. For example, a weighted graph is represented as $G = (V, E, \{\}, \{weight : E \longrightarrow \mathbb{R}\})$.

The Set, *WorkItem.* A *workitem*($\in$ *WorkItem*) is a *tuple*. The first element of the tuple is a vertex, and the remainder are the state and the ordering attribute values. For example, the Chaotic SSSP algorithm stores a vertex and a distance in a *workitem*. The size of the tuple is determined by the states and the ordering attributes used in the AGM formulation of a given algorithm. The *workitem* tuple elements are accessed using the *bracket operator*; e.g., if $w \in WorkItem$ and $w = \langle v, p_0, p_1 \ldots, p_n \rangle$, then $w[0] = v$, $w[1] = p_0$, $w[2] = p_1$, and so on. The *workitem* data (i.e., the tuple elements) are used by the processing function to generate new work items and update state.

States. An AGM maintains state values as *mappings*. The domain of the state mappings is the set V. The co-domain depends on the possible values that can be held in states. For example, in Dijksta's SSSP algorithm the state mapping is $distance : V \longrightarrow \mathbb{R}$. In AGM terminology, accessing a state value associated with a vertex (or edge) "v" is denoted as "mapping[v]" (e.g., distance[v]).

Processing Function. A processing function $\pi : WorkItem \longrightarrow \mathbb{P}(WorkItem)$[2] takes a *workitem* as an argument and produces zero or more *workitems*. The body of the processing function consists of a set of *statements (Sts)*. A statement contains a *condition* $C : WorkItem \longrightarrow Bool$ based on input *workitem*, an *update to states* $U : WorkItem \longrightarrow Bool$, and a constructor $N : WorkItem \longrightarrow \mathbb{P}(WorkItem)$ describing how new *output workitems* should be constructed. The condition C is evaluated first. If it evaluates to *True*, state update U is evaluated. If both are *True*, the constructor N is invoked. The C indicates whether a St is applicable to a *workitem*, and the U evaluates to *True* if states are changed when processing the input *workitem*. The N of a statement is evaluated only if its C and U both evaluate to *True*. States are not explicit parameters to the processing function, and changes to state are treated as *side effects*. An implementation of an AGM must ensure that state updates happen atomically. To define a processing function statement, the condition (C), state update (U), and the *workitem* constructor (N) must be provided.

We use notation loosely based on set-comprehension notation to represent processing functions. The format of a processing function $\pi : WorkItem \longrightarrow \mathbb{P}(WorkItem)$ takes the form $\pi(w) = \{\{w_n | \langle N(w) \rangle, \langle U(w) \rangle, \langle C(w) \rangle\}, \ldots\}$. In this notation, w_n is the new *workitem* generated by N from the input *workitem* w. U and C represent the state update and the condition. This notation describes one statement. For processing functions where there are more than one statements, the notation can be duplicated for each statement and separate each

[2] We denote a powerset of a set A as $\mathbb{P}(A)$.

statement using a comma. Note that we use angle brackets ($\langle \ldots \rangle$) to deliminate parts of processing function. This is not a standard notation in set comprehension, but it makes parts of the processing function clear. Furthermore, we will provide the U part of AGM as a side effect, but we will treat it as a boolean (*True* when the side effect occurs, *False* if it does not).

Strict Weak Ordering Relation. The *workitems* generated by a processing function are ordered according to a strict weak ordering relation (represented using $<_{wis}$) defined on *WorkItem*, which induces equivalence classes on *workitems*. The *workitems* in an equivalence class are not comparable to each other, but any two *workitems* in different equivalence classes are. In the AGM, the *workitems* belonging to the same equivalence class can be processed by the processing function in parallel, but *workitems* belong to different equivalence classes are ordered according to the ordering on equivalence classes.

Initial Work Item Set. The initial *workitem* set contains *workitems* that represent the input to the algorithm. For example, for SSSP, the initial set of *workitems* will contain the *workitem* corresponding to the source vertex s.

The AGM. Having defined all supporting concepts we now give the definition of an AGM in Definition 1.

Definition 1. *An* Abstract Graph Machine *(AGM) is a 6-tuple (G, WorkItem, Q, π, $<_{wis}$, S), where*

1. $G = (V, E, vmaps, emaps)$ *is the input graph,*
2. $WorkItemSet \subseteq (V \times P_0 \times P_1 \cdots \times P_n)$ *where each P_i represents a state value or an ordering attribute,*
3. *Q - Set of states represented as mappings,*
4. $\pi : WorkItem \longrightarrow \mathbb{P}(WorkItem)$ *is the processing function,*
5. $<_{wis} : WorkItem \times WorkItem$ *- Strict weak ordering relation on workitems,*
6. $S \subseteq WorkItem$ *- Initial workitem set.*

3 SSSP Algorithms in AGM

In this section, we present AGM models for algorithms discussed in Table 1. To specify these models, we need to provide AGM elements from Definition 1. First, we provide the input graph, the *WorkItem*, the set of states, the processing function, and the initial *workitem* set. Then, we show that adding different orderings to the AGM models, we get existing distributed SSSP algorithms.

The input graph for the SSSP problem is a weighted graph: $G = (V, E, vmaps = \{\}, emaps = \{weight\})$. The basic *workitem* includes a vertex and its distance and *WorkItem* for SSSP is defined as $WorkItem^{sssp} \subseteq (V \times Distance)$. The basic *workitem* is extended by additional ordering attributes when necessary (e.g., in KLA-SSSP). The set of states includes a single mapping *distance* for storing the distance from the source vertex. The *distance* mapping is defined as $distance : V \longrightarrow \mathbb{R}^*_+$. The processing function for SSSP changes the distance state if the input *workitem*'s distance for a given vertex is less than what is already stored for that vertex in the distance map. The list of adjacent vertices of a given vertex are accessed through the $neighbors : V \longrightarrow \mathbb{P}(V)$ function. The basic (it will be extended with additonal functionality for some algorithms) processing function for the SSSP graph problem is defined in Definition 2.

Definition 2. $\pi^{sssp} : WorkItem^{sssp} \rightarrow \mathbb{P}(WorkItem^{sssp})$

$$\pi^{SSSP}(w) = \begin{cases} \{w_n | \langle w_n[0] \in neighbors(w[0]) \\ \qquad \text{and } w_n[1] = w[1] + weight(w[0], w_n[0]) \rangle, \\ \quad \langle distance(w[0]) \longleftarrow w[1] \rangle, \\ \quad \langle if \;\; w[1] < distance(w[0]) \rangle \} \end{cases}$$

The SSSP processing function (π^{sssp}) has a single statement. The statement is executed only if the input *workitem*, *w*s' distance is less than the value stored in the *distance* map for the vertex in the *workitem* ($w[0]$, the first element of the *workitem* tuple). Constructor of the statement specifies how to construct the new *workitem* w_n. The processing function defines the core logic that needs to be achieved by any SSSP algorithm. Some of the algorithms discussed in Table 1 extend this definition because of the way they order *workitems*.

Chaotic SSSP. The Chaotic SSSP algorithm does not order *workitems*. Therefore, the strict weak ordering relation is defined in such a way that no two *workitems* are related (defined in Definition 3).

Definition 3. $<_{ch}$: $WorkItem^{sssp} \times WorkItem^{sssp}$ *is a binary relation where* $\forall w_1, w_2 \in WorkItem^{sssp} : w_1 \not<_{ch} w_2$.

This relation induces only one equivalence class, and all the *workitems* in this class can be executed in parallel. The AGM model for Chaotic SSSP algorithm is given in Proposition 1. The presented AGM uses the strict weak ordering defined in Definition 3.

Proposition 1. *Chaotic Algorithm is an instance of an AGM where*

1. $G = (V, E, vmaps = \{\}, emaps = \{weight\})$ *is the input graph,*
2. $WorkItem = WorkItem^{sssp}$,
3. $Q = \{distance\}$ *is the state (initially* $\forall v \in V, distance(v) = \infty$*),*
4. $\pi = \pi^{sssp}$,
5. *Strict weak ordering relation* $<_{wis} = <_{ch}$,
6. $S = \{<v_s, 0>\}$ *where* $v_s \in V$ *and* v_s *is the source vertex.*

Dijkstra's SSSP. The Dijkstra's SSSP algorithm globally orders *workitems* by their associated distances (Definition 4).

Definition 4. $<_{dj}$: *$WorkItem^{sssp} \times WorkItem^{sssp}$ is a relation where* $\forall w_1, w_2 \in WorkItem^{sssp} : w_1 <_{dj} w_2$ *iff* $w_1[1] < w_2[1]$.

The AGM formulation for Dijkstra's SSSP is same as the AGM formulation in Proposition 1 except for the strict weak ordering. In $<_{dj}$, two *workitems* belong to the same equivalence class if they have the same distance. In general, the equivalence classes generated by $<_{dj}$ are small, hence the parallelism available in Dijkstra's SSSP algorithm is limited.

***Δ*-Stepping Algorithm.** *Δ*-Stepping [9] SSSP algorithm arranges vertex-distance pairs into distance *buckets*) of size $\Delta \in \mathbb{N}$ and executes buckets in order. Within a bucket, vertex-distance pairs can be executed in any order. Processing a bucket may produce extra work for the same bucket or for successive buckets. The strict weak ordering relation for *Δ*-stepping algorithm is given in Definition 5. As for Dijkstra's algorithm, *Δ*-stepping AGM is as in Proposition 1 with ordering replaced by $<_{\Delta}$.

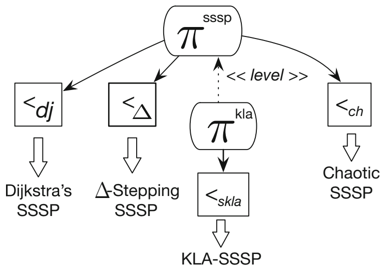

Fig. 2. Summary of AGMs for SSSP algorithms.

Definition 5. $<_{\Delta}$: *$WorkItem^{sssp} \times WorkItem^{sssp}$ is a relation where* $\forall w_1, w_2 \in WorkItem^{sssp} : w_1 <_{\Delta} w_2$ *iff* $\lfloor w_1[1]/\Delta \rfloor < \lfloor w_2[1]/\Delta \rfloor$.

KLA-SSSP Algorithm. The *K-Level Asynchronous* (KLA) paradigm [6] processes vertices up to k topological levels asynchronously (k can be varied). Correspondingly, the KLA-SSSP AGM orders *workitems* by their *level*. To do this, *workitems* include an additional *ordering attribute*. The KLA-SSSP *WorkItem* is defined as $WorkItem^{kla} \subseteq V \times Distance \times Level$ where $Level \subseteq \mathbb{N}$. The processing function also is extended to populate the level attribute (Definition 6).

Definition 6. $\pi^{kla} : WorkItem^{kla} \longrightarrow \mathbb{P}(WorkItem^{kla})$

$$\pi^{kla}(w) = \begin{cases} \{w_n | \langle w_n[0] \in neighbors(w[0]) \\ \qquad \text{and } w_n[1] = w[1] + weight(w[0], w_k[0]) \\ \qquad \text{and } w_n[2] = w[2] + 1\rangle, \\ \quad \langle distance(w[0]) \longleftarrow w[1]\rangle, \\ \quad \langle if\ w[1] < distance(w[0])\rangle\} \end{cases}$$

The *workitems* within consecutive k levels can be executed in parallel. The strict weak ordering relation for KLA-SSSP is given in Definition 7. The AGM for KLA-SSSP algorithm is as AGM in Proposition 1 except for the processing function, which is replaced with π^{kla}, and for the strict weak ordering, which is replaced with $<_{skla}$ defined in Definition 7.

Definition 7. $<_{skla}$: *$WorkItem^{kla} \times WorkItem^{kla}$ is a relation where* $\forall w_1, w_2 \in WorkItem^{kla} : w_1 <_{skla} w_2$ *iff* $\lfloor w_1[2]/K \rfloor < \lfloor w_2[2]/K \rfloor$.

Family of SSSP Algorithms. The SSSP AGMs are summarized in Fig. 2. Dijkstra's, Δ-stepping, and Chaotic algorithms share the same processing function but with different orderings. Both Dijkstra's algorithm and Δ-stepping algorithm use distance to define their strict weak orderings. KLA-SSSP uses levels to order *workitems*. The only difference between π^{sssp} and π^{kla} is that π^{kla} has logic to update level attribute in newly generated *workitems*. In Fig. 2, we represent this with a dashed arrow to indicate that π^{kla} is an *extended* version of π^{sssp}. Because all the algorithms use the same processing function (with ordering extension for KLA-SSSP), they form an *algorithm family*.

4 Extended Abstract Graph Machine

AGMs are abstract and independent of implementation details. However, distributed graph algorithms are strongly impacted by properties of the distributed architecture they run on. To capture that impact, we introduce *extended* AGM (EAGM) that represents *spacial distribution* on a distributed memory platform. Currently, we recognize 4 hierarchical levels of distribution that roughly match modern distributed systems (arrows indicate inclusion):

$$\begin{tikzcd}\boxed{\textbf{Global}} \arrow[r] & \boxed{\textbf{Process}} \arrow[r] & \boxed{\textbf{Numa}} \arrow[r] & \boxed{\textbf{Thread}}\end{tikzcd}$$

Given the spatial hierarchy, we use EAGMs to specify *spatial orderings* for AGM graph algorithms. Spatial orderings apply non-semantic ordering on *workitems* throughout the spatial hierarchy of a distributed machine. The ordering at the **Global** level is the same as in the underlying AGM, keeping the semantics of an AGM intact. Since the global ordering maintains the equivalence classes of AGM, *workitems* can be further ordered at the lower levels of the hierarchy. For example, two different EAGM spatial orderings for Δ-stepping are $<_{\Delta} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{ch}$ and $<_{\Delta} \rightarrow <_{ch} \rightarrow <_{ch} \rightarrow <_{dj}$ where each ordering corresponds to the EAGM level (the orderings are as defined in the previous section). The first spatial ordering enforces $<_{\Delta}$at the global level, but leaves execution in buckets unordered ($<_{ch}$). The second spatial ordering applies Dijkstra's ordering at the **Thread** level ($<_{dj}$), which means that *workitems* at every thread are ordered in a priority queue as they reach the thread in the spatial distribution. In summary, an EAGM consists of an AGM and a spatial architecture hierarchy annotated by spatial orderings.

Table 2. Thread ordered, numa ordered and process ordered EAGMs for Δ-stepping, KLA and Chaotic AGMs.

	`buffer`	`threadq`	`numaq`	`nodeq`
Δ-stepping	$<_{\Delta} \downarrow <_{ch} \downarrow <_{ch} \downarrow <_{ch}$	$<_{\Delta} \downarrow <_{ch} \downarrow <_{ch} \downarrow <_{dj}$	$<_{\Delta} \downarrow <_{ch} \downarrow <_{dj} \downarrow <_{ch}$	$<_{\Delta} \downarrow <_{dj} \downarrow <_{ch} \downarrow <_{ch}$
KLA-SSSP	$<_{kla} \downarrow <_{ch} \downarrow <_{ch} \downarrow <_{ch}$	$<_{kla} \downarrow <_{ch} \downarrow <_{ch} \downarrow <_{dj}$	$<_{kla} \downarrow <_{ch} \downarrow <_{dj} \downarrow <_{ch}$	$<_{kla} \downarrow <_{dj} \downarrow <_{ch} \downarrow <_{ch}$
Chaotic	$<_{ch} \downarrow <_{ch} \downarrow <_{ch} \downarrow <_{ch}$	$<_{ch} \downarrow <_{ch} \downarrow <_{ch} \downarrow <_{dj}$	$<_{ch} \downarrow <_{ch} \downarrow <_{dj} \downarrow <_{ch}$	$<_{ch} \downarrow <_{dj} \downarrow <_{ch} \downarrow <_{ch}$

In Table 2, we apply Dijkstra's strict weak ordering relation (Definition 4) to spatial hierarchy levels of Process (`nodeq`), Numa (`numaq`), and of Thread (`threadq`) to derive EAGMs for algorithms in Table 1. The *buffer* represents the original algorithm without spatial level orderings. The table shows orderings for each combination of ordering and AGM, where the ordering chain corresponds to the archtectural hierarchy given at the beginning of this section. Each EAGM generates a variation of the main algorithm defined by its corresponding AGM. By methodical application of spatial ordering, we derive a family of SSSP algorithms. In the next section, we evaluate the performance of different EAGMs.

5 Experiments and Results

In this section, we implement and compare the weak scaling performance of each derived EAGM in Table 2. In addition, we also compare the performance of the EAGMs to the performance of SSSP algorithms available in two well-known graph processing frameworks PowerGraph [5] and PBGL [4].

Weak scaling performance is measured on two types of synthetic *R-MAT* [2] graphs: RMAT1 graphs with R-MAT parameters $A = 0.57$, $B = C = 0.19, D = 0.05$ and with edge weights ranging 0–100, and RMAT2 graphs with R-MAT parameters $A = 0.5$, $B = C = 0.1, D = 0.3$ and with edge weights 0–255. All experiments were carried out on Cray XE6/XK7 nodes, each with 2 AMD Opteron Abu Dhabi CPUs (for total of 32 cores), and 64 GB of memory per node (4 numa domains, 2 per CPU).

The algorithms are implemented in AM++ [14], a light-weight active messaging framework. Graph vertices are equally distributed among distributed processes and in-node graph structure is stored in *compressed sparse row* format. Disjktra's orderings is implemented using concurrent priority queues at the process and the numa levels, and using standard priority queue at the thread level.

5.1 Scaling Results

The weak scaling results are presented in Figs. 3, 4 and 5. Experiment results for basic AGMs are represented using the `buffer` designator. As in Table 2, EAGMs with thread-level, node-level and numa-level Dijkstra orderings are represented using `threadq`, `nodeq` and, `numaq` designators. We tested the performance of Δ-stepping EAGMs for three delta values ($\Delta = 3, 5, 7$) and KLA-SSSP EAGMs with three k values ($k = 1, 2, 3$). In the following, we discuss results in detail.

Δ-Stepping Variations. The basic Δ-stepping (`buffer`) algorithm performs the best in-node (up to 32 cores). Since no communication is involved, the additional ordering provided by the other implementations does not provide a sufficient benefit for its overhead. In general, the `threadq` variation is the fastest in the distributed setting for both RMAT1 and RMAT2 graph inputs. The `nodeq` and the `numaq` variations perform better with increasing deltas, but they are not competitive with the `buffer` implementation.

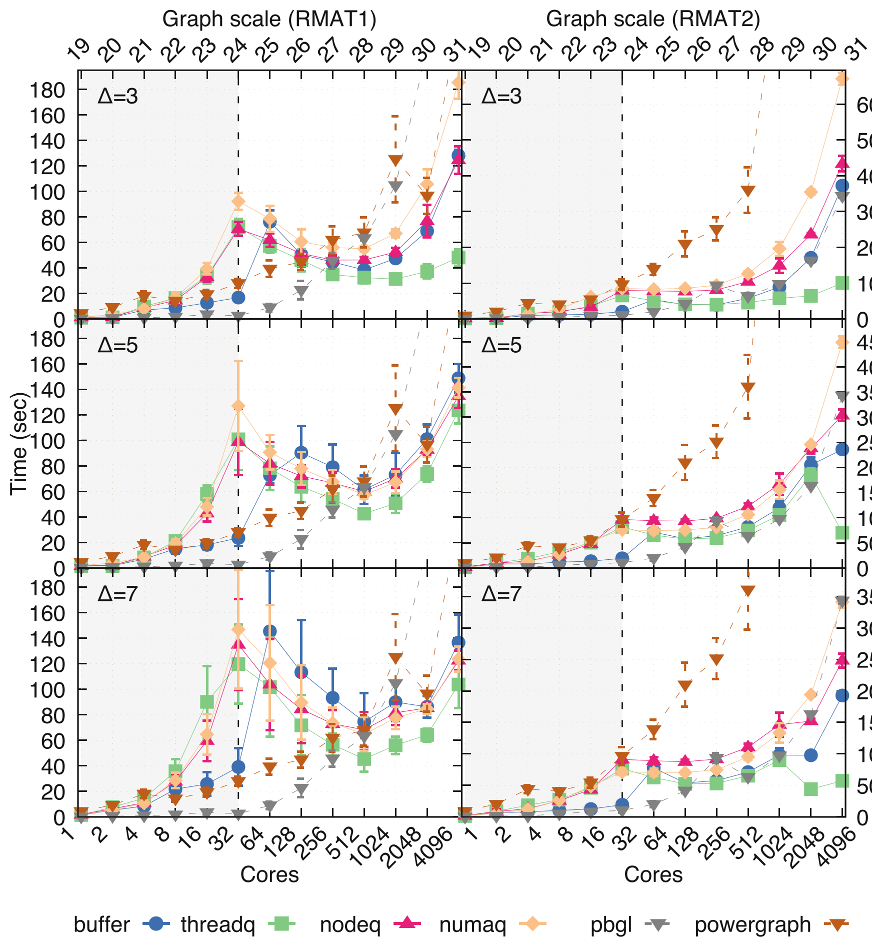

Fig. 3. Timing results of Δ-stepping. Shaded region indicates single node runs.

For RMAT1 graph inputs, PoweGraph shows better distributed performance for small scale graphs. However, for larger graph inputs, PowerGraph does not scale well. All the Δ-stepping EAGMs outperform PowerGraph at higher scales, especially for RMAT2. The `threadq` EAGM shows better performance than PBGL on RMAT2 graphs, and for RMAT1 graphs, all EAGMs outperform PBGL.

In summary, while in-node performance is dominated by the basic Δ-stepping algorithm (excluding PowerGraph and PBGL results), the distributed execution shows significant improvement with the `threadq` EAGM. Although the `numaq` and `nodeq` variations provide more ordering than the `threadq` variation, the overhead of the concurrent ordering reduces the performance of `numaq` and `nodeq`.

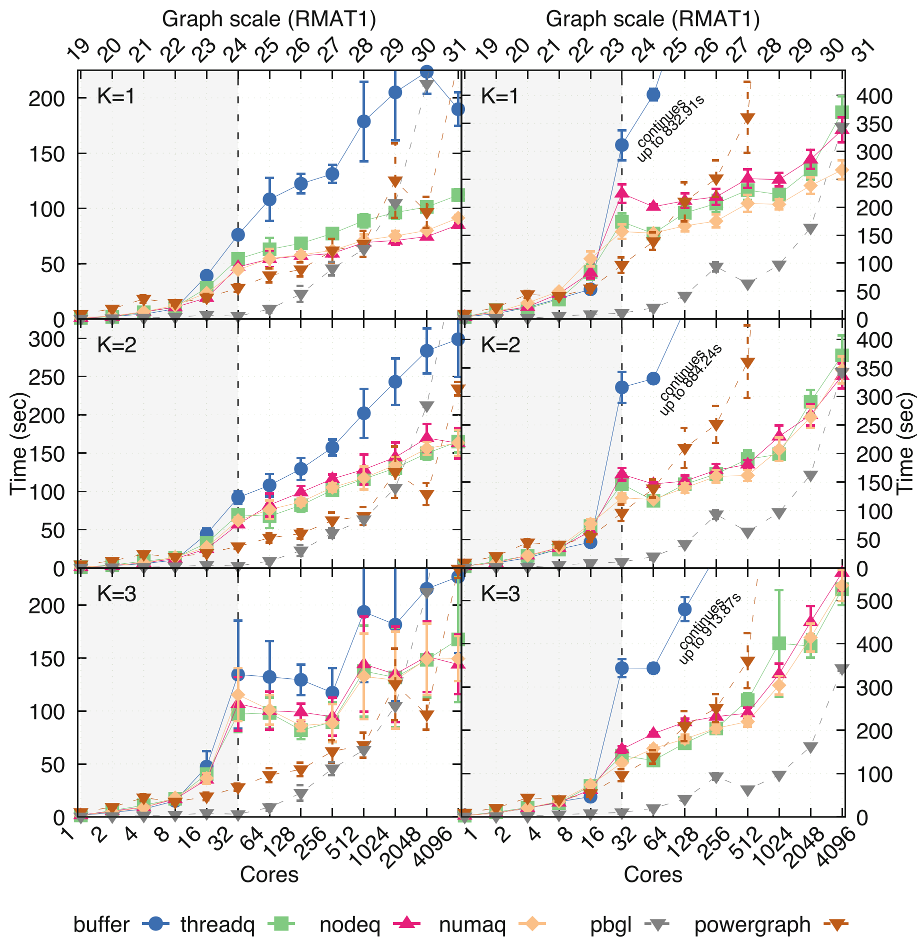

Fig. 4. Timing results of KLA. Shaded region indicates single node runs.

KLA Variations. KLA variations show different performance characteristics than Δ-stepping. For KLA, the `nodeq` and the `numaq` variations perform the best at scale, with $K = 1$. At greater K values, the performance of `threadq` is comparable to `nodeq` and `numaq`, but, in absolute terms, the performance at higher K values is worse than at $K = 1$. The `numaq` and `nodeq` provide the best potential ordering by ordering the most items. The overheads are kept at bay because at $K = 1$ all the writes to the next level's queue occur before all the reads. For higher K values, writes and reads get more mixed, and the advantage of `numaq` and `nodeq` becomes less pronounced. In KLA, for both RMAT1 and RMAT2 inputs, all EAGM variations (`threadq`, `nodeq` and `numaq`) perform better compared to the basic `buffer` variation.

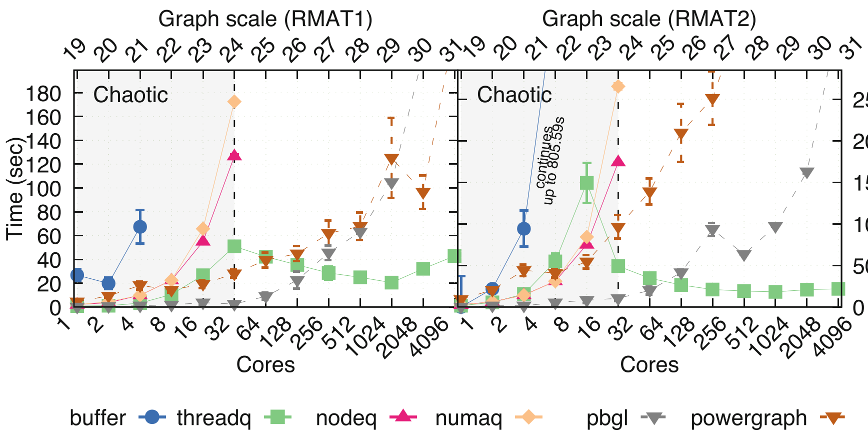

Fig. 5. Timing results of the Chaotic EAGM. Shaded region indicates single node runs.

For RMAT1 graph inputs, PowerGraph outperforms almost all the KLA EAGMs. However, PowerGraph execution time tends to increase with the scale, but KLA-SSSP EAGM variations tend to scale well with the increasing scale. For RMAT2 graph inputs, all the KLA-SSSP EAGM variations, except for `buffer`, outperform PowerGraph in distributed execution. However, for RMAT2, PBGL outperforms almost all EAGMs, and `numaq` and `nodeq` tend to perform better at higher scales with $K = 1$. All the EAGMs show better performance than PBGL for RMAT1 graphs.

Chaotic Variations. For chaotic EAGMs, the thread-level ordering shows good performance, specially in distributed execution. For RMAT2, `threadq` weak scales almost perfectly in distributed execution. In addition, the `threadq` variation outperforms GraphLab and PBGL for both RMAT1 and RMAT2 graphs in distributed execution. Furthermore, the `threadq` Chaotic EAGM is faster than all other EAGMs in terms of absolute performance, demonstrating how the structured (E)AGM approach may result in new, highly performant algorithms.

6 Related Work

Abstract Models–For shared memory systems ordering in graph algorithms is studied as schedulers. Nguyen and Pingali synthesized concurrent schedulers in [11]. Pingali et al. ([12] and [13]) discussed a data-centric formulation, that treats graphs as abstract data types, called *operator formulation*. Ordering is achieved using an operator called “ordered set iterator”. The AGM formulations' processing function works at the *workitem* level while operator formulation is applied on the graph. In addition, their ordering formulation is different from the AGM ordering formulation.

Spatial Ordering–Though, parallel processing in SSSP is well studied, spatial level orderings for SSSP problem are not common. Pingali et al.has done research on shared memory systems with Galois [10] scheduler, OBIM [8] that include spatial characteristics of the machine. In summary, shared memory models may not extend to distributed memory immediately due to cost factors that are significant in distributed memory than in shared memory (e.g., low compute/communication ratios, overhead of barriers, overhead of subgraph computations etc.,). The AGM model is designed to minimize such overheads.

7 Conclusions

Using the AGM abstraction, we showed that existing distributed graph algorithms; Dijkstra's SSSP, Δ-stepping SSSP and KLA-SSSP has the same processing logic but with different orderings. These orderings generate different equivalence class either based on distance or based on the level. We also showed, proposed EAGM model generates more fine-grained orderings at less synchronized spatial levels. Results of our experiments showed that some of the generated algorithms perform better compared to standard distributed memory, parallel SSSP algorithms under different graph inputs.

Acknowledgments. This research is based upon work supported by the National Science Foundation under grant 1319520. Access to computational resources was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU was also supported in part by Lilly Endowment, Inc. Significant part of this work was performed while the authors were affiliated with Indiana University.

References

1. Bellman, R.: On a routing problem. Technical report, DTIC Document (1956)
2. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. In: SDM, vol. 4, pp. 442–446. SIAM (2004)
3. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
4. Edmonds, N., Breuer, A., Gregor, D., Lumsdaine, A.: Single-source shortest paths with the parallel boost graph library. In: The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ, pp. 219–248 (2006)
5. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: OSDI, vol. 12, p. 2 (2012)
6. Harshvardhan, Fidel, A., Amato, N.M., Rauchwerger, L.: KLA: a new algorithmic paradigm for parallel graph computations. In: Proceedings of 23rd International Conference on Parallel Architectures and Compilation, pp. 27–38. ACM (2014)
7. Kanewala, T.A., Zalewski, M., Lumsdaine, A.: Abstract graph machine. arXiv preprint arXiv:1604.04772 (2016)

8. Lenharth, A., Nguyen, D., Pingali, K.: Priority queues are not good concurrent priority schedulers. The University of Texas at Austin, Department of Computer Sciences. Technical report TR-11-39 (2011)
9. Meyer, U., Sanders, P.: δ-stepping: a parallelizable shortest path algorithm. J. Algorithms **49**(1), 114–152 (2003)
10. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Proceedings of 24th ACM Symposium on Operating Systems Principles, pp. 456–471. ACM (2013)
11. Nguyen, D., Pingali, K.: Synthesizing concurrent schedulers for irregular algorithms. In: ACM SIGPLAN Notices, vol. 46, pp. 333–344. ACM (2011)
12. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Méndez-Lojo, M., et al.: The tao of parallelism in algorithms. ACM SIGPLAN Not. **46**(6), 12–25 (2011)
13. Prountzos, D., Manevich, R., Pingali, K.: Elixir: a system for synthesizing concurrent graph programs. ACM SIGPLAN Not. **47**(10), 375–394 (2012)
14. Willcock, J.J., Hoefler, T., Edmonds, N.G., Lumsdaine, A.: AM++: a generalized active message framework. In: Proceedings of 19th Internatational Conference on Parallel Architectures and Compilation Techniques, pp. 401–410. ACM (2010)