Self-stabilizing Connected Components

Piyush Sao, Christian Engelmann
Oak Ridge National Laboratory
{ saopk, engelmannc} @ornl.gov

Srinivas Eswar, Oded Green, Richard Vuduc
Georgia Institute of Technology
{ seswar3, ogreen, richie } @gatech.edu

***Abstract*—For the problem of computing the connected components of a graph, this paper considers the design of algorithms that are resilient to transient hardware faults, like bit flips. More specifically, it applies the technique of *self-stabilization*. A system is self-stabilizing if, when starting from a valid or invalid state, it is guaranteed to reach a valid state after a finite number of steps. Therefore on a machine subject to a transient fault, a self-stabilizing algorithm could recover if that fault caused the system to enter an invalid state.**

We give a comprehensive analysis of the valid and invalid states during label propagation and derive algorithms to verify and correct the invalid state. The self-stabilizing label-propagation algorithm performs $\mathcal{O}(V \log V)$ additional computation and requires $\mathcal{O}(V)$ additional storage over its conventional counterpart (and, as such, does not increase asymptotic complexity over conventional label propagation). When run against a battery of simulated fault injection tests, the self-stabilizing label propagation algorithm exhibits more resilient behavior than a triple modular redundancy (TMR) based fault-tolerant algorithm in 80% of cases. From a performance perspective, it also outperforms TMR as it requires fewer iterations in total. Beyond the fault tolerance properties of self-stabilizing label-propagation, we believe, they are useful from the theoretical perspective; and may have other use-cases.

I. Introduction

We are interested in the problem of designing an algorithm to compute the connected components of a graph in a way that is tolerant of transient hardware faults, such as bit flips. When such a fault occurs, it could bring the algorithm into an arbitrary state, which might be invalid. Our approach is based on the idea of *self-stabilization*, which originally arose in distributed control [15]. A self-stabilizing system is one that, beginning in an arbitrary state, whether valid or invalid, nevertheless reaches a valid state after a finite number of steps. Thus, a self-stabilizing algorithm can be resilient to transient faults. Previously, we derived self-stabilizing iterative numerical solvers [34], assuming a *selectively reliable* execution model, meaning that one can periodically invoke an expensive but fault-free correction step (Section II). Here, we extend these ideas to iterative but discrete algorithms, using *label propagation* methods for connected components as a first test case.

Label propagation methods are iterative in nature and attractive in practice owing to their inherent parallelism. Each vertex begins with its id as a label. In every subsequent iteration, a vertex updates its label by finding the minimum label among its neighbors and itself. The algorithm terminates when labels stop changing. The correctness of the label propagation algorithm depends on the hardware being entirely fault-free.

In a preliminary study, we explored how augmenting label propagation with an additional data structure could help test whether a given iteration of label propagation executed correctly [33]. Besides a label value, each vertex would maintain the id of the neighbor responsible for that value. With this information, it is possible to check the correctness of an iteration far more cheaply than, say, re-executing the entire iteration. We showed that this mechanism could efficiently detect and correct a corrupted state, even at extremely high fault injection rates. However, it also relied, critically, on the previous state being valid. Consequently, if the algorithm started from an invalid state, then the scheme could not recover. By discovering a variant of the algorithm that is self-stabilizing, which recall may begin in an arbitrary state, this paper improves significantly upon that prior work.

Formally, consider a graph $G = \{V, E\}$ and a collection of labels L, one for each vertex $v \in V$. We also keep the vertex-id of its neighbor that passed its current label, that we call *parent* array and denote by P. So the state of the algorithm is $S = \{L, P\}$. Our goals will be to address two questions.

- Starting from some arbitrary state S, can label propagation eventually converge to the correct solution?
- Given an arbitrary state S (valid or invalid), can we efficiently construct a guaranteed valid state $\tilde{S}$ that is "close" to S?

The first problem is equivalent to finding a path between the vertex v and its label $L[v]$, for every vertex $v \in V$. To our knowledge, no efficient algorithm exhibits the same parallelism as the label propagation algorithm besides recomputing $L[v]$. For the second problem, we do not consider going to the trivial valid initialization state, i.e., $L[v] = v,\ P(v) = v, \forall v \in V$, as a satisfactory solution. Such an approach throws away a lot of useful information in the state S and requires significant computational effort. The requirement of "closeness" reflects the idea that a corrupted state may still be mostly correct.

The first contribution of this paper is identifying a set of sufficient conditions on the state S for it to be valid. We seek conditions that can also be verified efficiently and exhibit the same parallelism as the label propagation algorithm. Our second contribution is an efficient correction step that can bring any arbitrary state S to a guaranteed valid state. The net result is the first known self-stabilizing connected component algorithm (Section IV).

We empirically evaluate the resilience of our method in the presence of injected faults (Section V). Our evaluation shows that the self-stabilizing label propagation algorithm is more resilient to faults than another fault-tolerant label propagation algorithm based on triple modular redundancy (TMR). Specifically, we see this improvement for seven out of our eight test input networks. Furthermore, our self-stabilizing label propagation algorithm incurs a maximal overhead of 40% in our experiments, compared to the overhead of the TMR method, which is at least 200%. Beyond the specific case of connected components for static graphs, we suggest potential extensions (Section VII).

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

II. Faults and Algorithmic States

In this section, we briefly discuss the fault and reliability models and the concept of self-stabilization, which are important for subsequent discussion.

A. Faults in Computing Systems

The term fault denotes any instance of abnormal hardware behavior. We distinguish between *hard* and *soft* faults. A hard fault causes an application to end prematurely. The failure of a computes node or network falls into this category. In contrast, soft faults may not terminate the application but can lead to an incorrect result. A bit flip that occurs in memory or in a latch is an example of a soft fault. Interested readers can find a more detailed discussion on faults elsewhere [20].

This paper concerns only soft faults. A particularly insidious manifestation of a soft fault is silent data corruption. Silent data corruption occurs when a soft fault leads to corruption of a wide range of intermediate variables without notifying the application. In such cases, the application may arrive at an incorrect result. Thus, silent data corruption can lead to serious reliability issues.

B. Selective Reliability Model of Execution

We assume the *selective reliability model* of faulty execution [20]. In this model, there are two modes of computation: a) an *unreliable mode* that is cheap (in time, storage, or other relevant performance metric), and b) a *reliable mode* that is presumably slow or otherwise costly. One prefers to perform as much computation as possible in unreliable mode.

C. Self-stabilizing Iterative Algorithms

We discuss how an abstraction of an algorithm as a state machine can help us to make the algorithm more resilient.

1) States of an Algorithm: An iterative algorithm can be viewed as a system with states and transition rules. Its *state* is a subset of the intermediate variable that enables continued execution.

2) Definition: Valid and invalid states: The state of the algorithm is said to be *valid* if the algorithm will converge to a correct solution in fault-free execution starting from this state. If the algorithm does not converge to a correct solution from its current given a fault-free execution, then the state is said to be *invalid*. This abstraction has helped us to construct resilient algorithms [34], [35], [33].

3) Impact of hardware fault on algorithms state: Given a fault-free execution, from beginning to end, the state of the algorithm should be valid at all times. However, once faults are introduced the above is typically not correct. Consider a soft fault such as a single bit flip on a value in L. This could lead to a vertex pointing to an incorrect connected component. A subsequent fault-free execution, after this one error, will lead to an incorrect solution. Our principle for designing a more resilient algorithm is based on the idea of augmenting the algorithm so that it can bring itself to a valid state, by design.

Our basic recipe for creating a fault-tolerant algorithm is as follows[34]. First, we identify the *state* of the algorithm: valid or invalid. We then augment the algorithm with a mechanism that can bring the state of the algorithm back to a valid state. We distinguish between following two principles for bringing the algorithm to a valid state:

4) Self-stabilizing algorithm: Formally, we regard a system as being self-stabilizing if starting from any arbitrary *state*, it comes to a valid state in a finite number of steps [15]. If a fault causes an algorithm to reach an invalid state, a self-stabilizing algorithm will come to a valid state in a finite number of iterations. Subsequently, the algorithm will converge to the correct solution if all the following computations do not encounter any fault.

5) Self-correcting algorithm: A self-correcting algorithm can bring itself to a valid state by correcting its state with information of a previous valid state. In contrast to self-stabilization, self-correction must restart itself from a valid state. In reality, it is not such a limitation as most algorithms by design start from a valid state. On the other hand, the self-correcting algorithm can be more efficient than the self-stabilizing algorithm, as it can meaningfully exploit information of a previous valid state. We have previously developed a self-correcting algorithm for connected-components[33], which partly motivated this paper's search for an even more robust self-stabilizing variant instead.

III. Connected components

Table I: Symbols used in the fault free label propagation algorithm and in the new fault tolerant algorithm.

Symbol	Description	Size
V	Vertices in the graph	$\mathcal{O}(V)$
E	Edges in the graph	$\mathcal{O}(E)$
d	Diameter of the graph	
$adj(v)$	Adjacency list for vertex $v \in V$	
$\mathcal{N}(v)$	$v \cup adj(v)$	
L	Label array	$\mathcal{O}(V)$
L^i	Label array after iteration i	$\mathcal{O}(V)$
L^∞	The final and correct label array upon.	$\mathcal{O}(V)$
P	Parent Array	$\mathcal{O}(V)$
E_H	Set of directed edged $E_H = \{v \to P(v)\ \forall v \in V\}$	$\mathcal{O}(V)$
H	Propagation graph $H = \{V, E_H\}$	$\mathcal{O}(V)$
$\mathcal{P}(v)$	Set of ancestors of v $\mathcal{P}(v) = \{P(v), P^2(v), \ldots\}$	
$\mathcal{R}(v)$	$\mathcal{R}(v) = P^\infty(v)$	
$\mathcal{A}(v)$	$\mathcal{A}(v) = \min \mathcal{P}(v)$	

Given an undirected graph $G = (V,\ E)$, a connected-component C is a subset of its vertices V that satisfy the following conditions: a) there is a path between any two vertices of C in the graph G; and b) there are no paths between any vertex in C and $V - C$. We would like to find all the connected components G. This problem is known as graph connected component problem.

We focus on the highly parallel label propagation algorithm for computing graph connected components. We start with a brief review of how label propagation works in the case of no faults.

A. Fault-free label propagation algorithm

Each vertex in the graph is labeled with a connected component, this denoted by the array L. Initially, each vertex is placed in its connected component and is labeled with its vertex id. Iteratively, all vertices in the graph inspect their neighbors and check if any of them have a label smaller than their existing label and if so the vertices replace their values. Hence the term label propagation. Fig. 1 shows conceptually how the labels propagate through the graph. The algorithm completes when every vertex in the graph has acquired its final label. In Fig. 2, we show a graph component, initial label of vertices (Fig. 2a), and final label of vertices (Fig. 2b). Specifically, the label of given iteration is computed as follows:

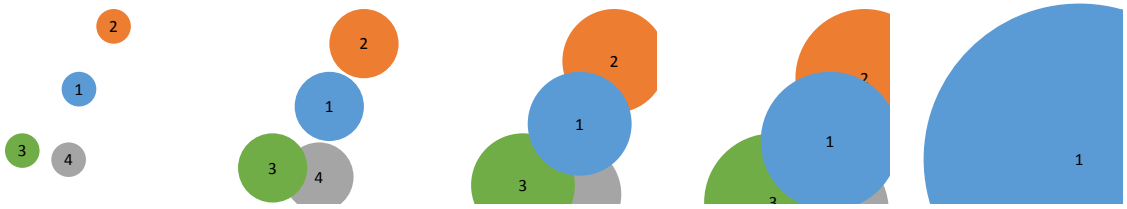


Figure 1: These sub-figures conceptually show (from left to right) how the connected component id propagates through the graph as time evolves. The subfigures represent snapshots of the of the connected component labeling at different times. For simplicity, this example assumes that all the shown vertices are connected. Initially, the number of connected components is equal to the number vertices. Fig. 1a depicts the initial state in which each vertex is in its component. Figs. 1b to 1d illustrate that some vertices belong to the same connected component yet may need many label updates (in either the same iteration or a separate iteration). Fig. 1e is the final state in which there is a single connected component.

Algorithm 1 Label propagation algorithm

Require: $G = (V, E)$
1: **function** CONNECTEDCOMPONENTS(G)
2: **for** each $v \in V$ **do**
3: $L^0[v] \leftarrow v; P[v] \leftarrow v;$ ▷ Initialization
4: $S \leftarrow \{L, P\}$
5: $Nc \leftarrow |V|,\ i \leftarrow 0$
6: **while** $Nc > 0$ **do**;
7: $\{S, Nc\} \leftarrow$ LABELPROPAGATIONSWEEP(G, S)
8: $i \leftarrow i + 1$
9: **return** S
10: **function** LABELPROPAGATIONSWEEP(G, S)
11: $Nc \leftarrow 0$
12: **for** each $v \in V$ **do**
13: **for** each $u \in E(v)$ **do**
14: **if** $L[u] < L[v]$ **then**
15: $L[v] \leftarrow L[u]$
16: $P[v] \leftarrow u$
17: $Nc \leftarrow Nc + 1;$
18: **return** $\{S, Nc\}$

(a) Starting Labels (b) Final Labels

Figure 2: We show the initial (Fig. 2a)and the final labels (Fig. 2b)of a single graph component. The vertex-id is shown inside the node, and its label is shown in green beside it. In Fig. 4, we use this component to illustrate different cases of state corruption.

$$L^{i+1}[v] = \min_{u \in \mathcal{N}(v)} L^i[u], \tag{1}$$

where $\mathcal{N}(v) = \{v, adj(v)\}$ denotes the immediate neighborhood of v. The pseudocode for label propagation appears in Algorithm 1. Observe that Algorithm 1 includes an additional data structure P, which is not part of the standard label propagation algorithm. Specifically, P is an array used to denote the parent of each vertex, where the parent $P[v]$ is the vertex responsible for the most recent change in the label of v.

We consider two ways to implement Eq. (1):

1) Using two different arrays (L^{i+1} and L^i) to store the labels in two consecutive iterations. We refer to this approach as the synchronous-LP algorithm.
2) Using a single array, L, where the values overwrite each other. Given the nature of parallel execution, we refer to this version as asynchronous-LP algorithm, since the values can be updated concurrently.

Depending on a given architecture and programming model, these two variants may perform differently. In the later discussion, we focus on the asynchronous-LP algorithm; which is also the more challenging of the two. Even so, our results are applicable to both instances of the label propagation algorithm.

Complexity Analysis of Label Propagation Algorithm: In each iteration of label propagation, we visit all the vertices and edges once and thus costs $\mathcal{O}(V + E)$. A total of $\mathcal{O}(d)$ iterations are needed to converge, where d is the diameter of the graph. Short-cutting can be used to bound the number of iteration to $\mathcal{O}(\log d)$ [37]. But in practice, the asynchronous-LPalgorithm only takes a few more iterations than implementing a full short-cutting step while avoiding the cost of short-cutting.

B. Self-correcting Label propagation Algorithm

In [33], we presented a self-correcting version of the Label propagation Algorithm. Here we extend the results of [33] to construct a self-stabiliziing version. Here, we briefly discuss self-correcting Label propagation Algorithm relevent to subseqeunt discussion.

1) Impact of faults in label propagation algorithm: A hardware fault can change the state of the algorithm. We term such undesirable changes as state-corruption. A corrupt label $L[v]$, depending on the corruption, may be ignored, become corrected in later iterations, or may propagate to all other vertices, causing the algorithm to give an incorrect result. We distinguish between the following three cases of label corruption:

1) $L^i[v] > v$;
2) $L^\infty[v] \leq L^i[v] \leq v$; and
3) $L^i[v] < L^\infty[v]$.

Here $L^\infty[v]$ is the final and correct label of vertex-v.

The first type of corruption is easy to identify, as we know that label of any vertex should not increase. In the third type of corruption, the vertex v has acquired a label that is smaller than the minimum vertex-id in its component. Such a corruption will propagate to all other vertices in the component. We have previously shown that for the second type of corruption, the label propagation algorithm would converge to the correct solution [33].

Theorem 1. ***Valid States:*** *The label array L is in a valid state that, given a fault-free execution from the current state onwards, the following is true for all vertices: $L^{\infty}[v] \leq L[v] \leq v$ [33].*

2) Self-correcting label propagation algorithm: The problem with theorem 1 is that it defines the validity of the state based on the correct output. Therefore, it cannot be used to verify the validity of current state. In our previous work, we presented the following set of conditions that can be used to check the validity of the current state [33]. That check must be performed synchronously, where each vertex maintains two label ids: one for the current iteration and one from the previous iteration.

Theorem 2. *Given a valid state for the previous iteration, L^{i-1}, the current label array L^i is valid if for all vertices v, L^i satisfies these conditions:*

1) $L^i[v] \leq v$;
2) $L^i[v] = L^{i-1}[P(v)]$; and
3) $P(v) \in \mathcal{N}(v)$.

The conditions of theorem 2 can be verified in $\mathcal{O}(1)$ time for any vertex v [33]. Thus, we can check validity for all vertices in $\mathcal{O}(V)$ time. We can cheaply recompute the labels for the vertices that do not satisfy theorem 2 using the valid previous state $L^{i-1}[v]$.

theorem 2 assumes and can only work correctly when the previous state L^{i-1} is valid. In the *self-correcting* label propagation algorithm, we detect and correct invalid labels in every iteration by using theorem 2 to ensure that the algorithm is in a valid state.

3) Limitations of Self-correcting label propagation algorithm: The *self-correcting* algorithm can only work when we have the copy of previous valid labels L^{i-1}. Thus, the *self-correcting* label propagation algorithm will not work with asynchronous-LP formutation.

In some cases, such as dynamic graphs, we would like to start the algorithm from a previously calculated state. For such states, we do not know whether they are valid for the changed graph. The *self-correcting* algorithm cannot work in such cases either.

IV. Self-stabilizing Label Propagation Algorithm

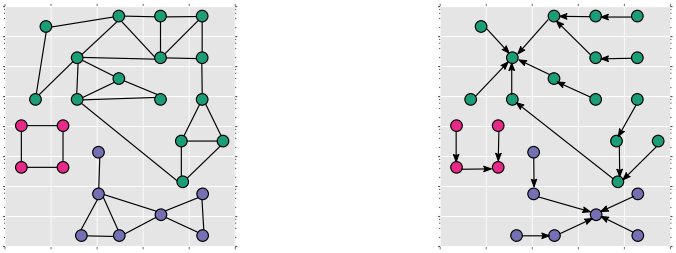
In this section, we show a set of conditions that are sufficient to verify the validity of a state, theorem 3. Theorem 3 is an extension of theorem 2, however, it does not require a previous valid state, i.e. $L^{i-1}[v]$ in theorem 2, to check the validity of the state. Using theorem 3 we construct a correction step described in Section IV that we use to bring the algorithm to a valid state.

A. Self-stabilizing Validity Conditions

Given an arbitrary state $S = \{L, P\}$, can we determine whether it is valid or not? To answer this, we present an extended version of theorem 2 that does not assume anything about prior iterations.

The key idea here is, that information about past iterations are already present in $S = \{L, P\}$. To put it concretely, we present following properties of the parent array P.

1) Relationship between $L[v]$ and $L[P(v)]$: In the fault-free execution of label propagation algorithm, label of a vertex $L[v]$ is always greater than or equal to its parent's label $L[P(v)]$. Consider the state of the label propagation algorithm during iteration i. A vertex v acquires its label from its parent $P(v)$ in some iteration $j \leq i$. So the current label of vertex v is $L^i[v] = L^j[P(v)]$. In the label propagation algorithm, labels of any vertex can only decrease as the iteration progresses. Thus, $P(v)$'s current label $L^i[P(v)]$ must be smaller than or equal to its earlier value $L^j[P(v)]$. So, $L^i[v] = L^j[P(v)] \geq L^i[P(v)]$.



(a) Input graph G (b) Propagation graph H

Figure 3: An input graph and its propagation graph.

$$L[v] \geq L[P(v)]. \tag{2}$$

2) $L[v] = v$ if and only if $P(v) = v$.: If any vertex is the parent of itself, i.e. $P(v) = v$ then $L[v] = v$. In Algorithm 1, all the vertices are initialized with $P(v) = v$ and $L[v] = v$. If a vertex never changes its label during label propagation iterations, then it will keep its parent and label. Conversely, if a vertex has changed its value then $L[v] < v$ and $P(v) \neq v$. Moreover, a changed $P(v) \neq v$ will never revert back to $P(v) = v$ as $P(v)$ only changes when $L[v]$ is changed. So $P(v) = v$ implies v obtained a label from itself, which is not possible. So, in the fault-free execution of Algorithm 1 we must have the following for all vertices v:

$$L[v] = v \iff P(v) = v. \tag{3}$$

Definition 1. Propagation Graph H: For a given state $S = \{L, P\}$ of execution of Algorithm 1 with input $G = \{V,\ E\}$, *Propagation Graph* H is the directed graph defined by $H = \{V,\ E_H\}$ where the edge set E_H consists of directed edges $v \to P(v)$ for all $v \in V$.

3) Structure of the Propagation Graph H: In the fault-free execution of Algorithm 1, the propagation graph H does not contain any cycles besides self-loops. Thus, H consists of many trees. When the algorithm is converged, H has one tree for each component in the graph. A path from any vertex to the root of its tree is also the path by which root's label propagated to that vertex, albeit in reverse. Fig. 3 shows an example of the propogation graph generated for a particular input.

Now we can formally state and prove the following set of conditions, which are sufficient to assert validity of any arbitrary state S.

Theorem 3. *Starting from state $S = \{L, P\}$, Algorithm 1 will converge to correct solution in subsequent fault-free execution, if S satisfies the following conditions.*

1 $L[v] \leq v$ for all $v \in V$;
2 $P(v) \in \mathcal{N}(v)$ for all $v \in V$;
3 $L[P(v)] \leq L[v]$ for all $v \in V$;
4 $L[v] = v \iff P(v) = v$ for all $v \in V$; and
5 Ignoring self-loops propagation graph H is a forest.

We use the theorem 1 to prove theorem 3. Following notation is helpful for remaining discussion.

$$\mathcal{R}(v) := \text{the root of tree containing } v$$

We already have $L[v] \leq v$ from Condition 1. We prove $L[v] \geq L^{\infty}[v]$ in following three parts:

$$\begin{aligned} L[v] &\geq L[\mathcal{R}(v)]; \\ L[\mathcal{R}(v)] &\geq L^{\infty}[\mathcal{R}(v)]; \text{ and} \\ L^{\infty}[\mathcal{R}(v)] &= L^{\infty}[v]. \end{aligned}$$

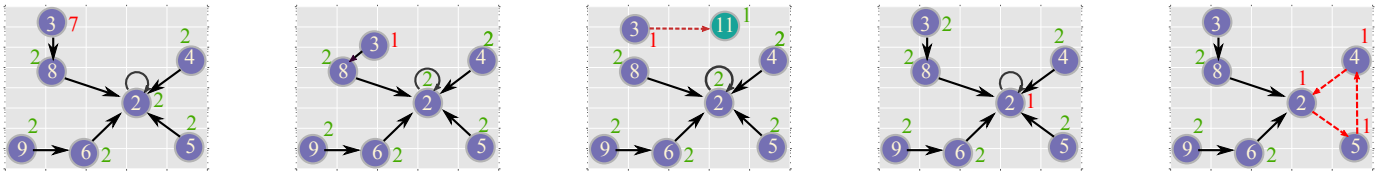


Figure 4: Different Corruption Scenarios: For the component shown in Fig. 2, we show a few possible cases of state corruption. The Fig. 4c shows a situation that can occur when the vertex-3 incorrectly reads its adjacency list and acquire a label, from a vertex that's not its neighbor. Such cases can be detected by verifying $P(v) \in \mathcal{N}(v)$. In Figs. 4a and 4b, we show a case where the label of the vertex three is corrupted. In the case of Fig. 4a, it will get corrected automatically, therefore requires no detection or correction. While in the case of Fig. 4b, this corruption will propagate to all vertices in the component. The corruption of this type can be detected by verifying $L[v] \geq L[P(v)]$. In Fig. 4d, the label of the root of the tree is corrupted. We can detect this corruption from the 4th condition. Finally, Fig. 4e shows the case when there is a cycle in the in the propagation graph H. We can only detect such corruptions by checking if there are any cycles in the propagation graph H.

Proof: From Condition 5, if H is a forest, then each tree in H will have a root node $\mathcal{R}(v)$. $\mathcal{R}(v)$ is the limit of the sequence $\{P(v),\ P^2(v), \ldots, \}$, therefore

$$\mathcal{R}(v) = P^{\infty}(v) \tag{4}$$

will hold for all vertex v.

1) Condition 3 implies label of any vertex v is greater than or equal to its parent's label. We can apply this condition on the parent itself to get $L[P(v)] \geq L[P^2(v)]$. Applying Condition 3 repeatedly, we get that the label of vertex v is greater than or equal to label of all its ancestors. Specifically, we are interested in the $P^{\infty}(v) = \mathcal{R}(v)$. Therefore,

$$L[v] \geq L[\mathcal{R}(v)] \tag{5}$$

2) Since the root of the tree $\mathcal{R}(v)$ is the parent of itself: $\mathcal{R}(v) = P(\mathcal{R}(v))$. Condition 4 implies that the label of the root $L[\mathcal{R}(v)]$ is root itself:

$$L[\mathcal{R}(v)] = \mathcal{R}(v) \tag{6}$$

Furthermore, for any vertex $v \geq L^{\infty}[v]$. Applying it to $\mathcal{R}(v)$, we get

$$\mathcal{R}(v) \geq L^{\infty}[\mathcal{R}(v)]. \tag{7}$$

3) Condition 2 implies v and $P(v)$ are in the same component. Similarly, $P(v)$ and $P^2(v)$ are in the same component. Verily, v and $\mathcal{R}(v)$ are in the same component in G and will have same final label. Thus we have:

$$L^{\infty}[v] = L^{\infty}[\mathcal{R}(v)] \tag{8}$$

So if all conditions of theorem 3 hold then we must have

$$L^{\infty}[v] \leq L[v] \leq v \quad \forall v \in V. \tag{9}$$

Hence, from theorem 1 we conclude that if all conditions of theorem 3 hold for a state $S = \{L, P\}$, then S is valid. □

Now we describe how we use theorem 3 to verify whether a state is valid in parallel. Conditions 1-4 theorem 3, may be verified *locally* and in parallel. By local we mean using only information to which a vertex has immediate access. However, Condition 5 requires cycle detection, which need global information.

The traditional algorithms for finding a cycle in a directed graph are not well suited for vertex-centric programming. In the sequential case, one can use Tarjan's strongly connected component algorithm [38], or breadth-first search (BFS), or depth-first search (DFS). These algorithm run at $\mathcal{O}(V + E)$ cost. In the graph H, the number of edges and vertices are equal: $V = E$. So the cost of these algorithms will be $\mathcal{O}(V)$. Since the other checks of theorem 3 cost $\mathcal{O}(V)$, the $\mathcal{O}(V)$ cost of cycle detection is, in theory, acceptable. However, due to the limited available parallelism or inability to express these cycle detectors easily in the vertex-centric model, we prefer not to use them to verify the condition 5.

B. Parallel Cycle Detection

Figure 5: We show how the corruption shown in Fig. 4e is detected. In Fig. 5, we show the $\mathcal{A}(v)$ for all the vertices. Since, $\mathcal{A}(v) = v$ for $v = 2$, we detect a cycle which $v = 2$ is the minimum element. Therefore, v resets its parent to itself and other members update their label with $L[v] = \min\{v, \mathcal{A}(v)\}$ (Fig. 5b).

We present a new algorithm for detecting cycles in the directed graph where each vertex has degree one such as H. The fundamental idea of our algorithm is that, if there are no cycles in the graph H, then a non-root vertex v cannot be an ancestor of itself.

Consider the following sequence $\mathcal{P}(v)$ of ancestors of a non-root vertex v

$$\mathcal{P}(v) = \left\{P(v),\ P^2(v), \ldots\right\}. \tag{10}$$

If there are no cycles in H, then $\mathcal{P}(v)$ will converge to root of the tree that v belongs to. But if v is a part of a cycle, then $\mathcal{P}(v)$ will become periodic and cycle through all the vertex in the cycle indefinitely. So the calculation of $\mathcal{P}(v)$ may not be used for detecting cycles in H.

Instead of $\mathcal{P}(v)$, we consider the minimum vertex-id among all the vertex that are ancestors of vertex v. We denote it as $\mathcal{A}(v)$:

$$\mathcal{A}(v) = \min \mathcal{P}(v) = \min\left\{P(v),\ P^2(v), \ldots\right\}. \tag{11}$$

In the fault-free case, both $\mathcal{A}(v)$ and $\mathcal{P}(v)$ will converge to the root of the tree. But unlike $\mathcal{P}(v)$, the calculation of $\mathcal{A}(v)$ will converge even if there is a cycle in H.

1) Cycle Detection using $\mathcal{A}(v)$: If there are no cycles in H, then a non-root vertex v cannot be equal to $\mathcal{A}(v)$. Thus, if we find a non-root vertex such that $v = \mathcal{A}(v)$ then following two condition holds:

1) v is a vertex in a cycle in H; and
2) v has the smallest vertex-id in the cycle.

Algorithm 2 Cycle Detection Algorithm

```
1: function LOOPDETECT(G = {V, E}, S = {L, P})
2:   Nc ← 1
3:   A(v) ← P(v)                                   ▷ Initialization
4:   while Nc > 0 do
5:     for all vertex v ∈ V do
6:       if A(v) ≠ min{A(v), A(P(v))} then
7:         A(v) ← min{A(v), A(P(v))}
8:         Nc ← Nc + 1
9:       P(v) ← P(P(v))                           ▷ Contraction
10:      if v = A(v) then
11:        Report Cycle
12:        P(v) ← v                               ▷ Break cycle
13:  return {A, S}
```

2) Calculating $\mathcal{A}(v)$: The calculation of $A(v)$ is equivalent to 'min-prefix' operation, and can be done in parallel using pointer jumping techniques such as Wyllie's list ranking algorithm [40]. We show the pseudocode in Algorithm 2. We initialize $A(v)$ for each vertex by $P(v)$. In every iteration, we update $A(v)$ as $A(v) = min(A(v), A(P(v))$, and update the $P(v) \leftarrow P^2(v)$.

C. Self-stabilizing Label-propagation Algorithm

We have not yet described how do we correct a state if the verification of theorem 3 fails. Before doing so, we consider two questions: a) in which order should we verify the conditions of theorem 3? and b) how frequently should we verify these conditions?

Correcting P: For each vertex v, we verify if $P(v) \in \mathcal{N}(v)$. If $P(v) \notin \mathcal{N}(v)$, then we reset the parent of the vertex v by setting $P(v) = v$. This step ensures that the propagation graph H is valid. In other words, if any two vertices v and u are connected in H, then they are connected in G as well.

Correcting L: For each vertex v, we calculate $\mathcal{A}(v)$ using Algorithm 2. Now we calculate new labels for each vertex as follows

$$L[v] = \min(v, \mathcal{A}(v))$$

Additionally, if there are any cycles, i.e. $v = \mathcal{A}(v)$, we reset the parents of those vertices: $P(v) = v$.

Convergence detection: We can show that after correcting L and P, the algorithm is in a valid state. We reliably run one iteration of the label-propagation algorithm to check if the algorithm is converged. If the algorithm is not converged then we restart the algorithm with the calculated valid state.

We show the complete self-stabilizing algorithm in Algorithm 3. Note that unlike Algorithm 1, Algorithm 3 can start with any arbitrary state S. Optionally, we can introduce an additional parameter, which is the period of self-stabilization, F. (Therefore, the frequency of self-stabilization is $1/F$). The Algorithm 3 has two loops that we denote by the outer and the inner-loop. The outer loop is responsible for bringing the algorithm to a valid state. The rest of outer-loop beyond the inner-loop executes either when the inner-loop reports convergences (i.e., $Nc = 0$) or when the iteration index i has reached a multiple of period F. When that happens, then we execute the self-stabilization step to bring the algorithm to a valid state, and using this valid state, we verify the convergence again. The self-stabilization step consists of three parts:

- verifying $P(v) \in \mathcal{N}(v)$;

Algorithm 3 Self-stabilizing Label Propagation Algorithm

```
Require: Input graph G = (V, E)
Require: Input state S = (L, P)
Require: Frequency of self-stabilization F
1: Nc ← |V|, i ← 0, Converged = 0
2: while Converged = 0 do                          ▷ Outer Loop
3:   while Nc > 0 and i ≢ 0 mod F do;               ▷ Inner Loop
4:     {S, Nc} ← LABELPROPAGATIONSWEEP(G, S)
5:     i ← i + 1
   Verifying State Validity
6:   for each v ∈ V do
7:     if P(v) ∉ N(v) then
8:       P(v) ← v
9:   {A, S} ← LOOPDETECT(G, S)
10:  for each v ∈ V do
11:    L[v] ← min(v, A(v))                         ▷ Update label
   Check Convergence
12:  {S, Nc} ← LABELPROPAGATIONSWEEP(G, S)
13:  if Nc = 0 then
14:    converged = 1
15: return S = {L, P}
```

- calculating $\mathcal{A}$ to detect and break cycles; and
- update label by $L[v] = \min(v, \mathcal{A}(v))$.

Once it has reached a valid state, we finally execute one label-propagation iteration to detect the convergence. The inner loop is the same as Algorithm 1. Typically the inner-loop forms the bulk of the computation. In the selective reliability model, we execute the inner-loop in unreliable mode and the self-stabilization step and final convergence detection in the reliable mode.

D. Asympotic Correction Cost

The costs of various corrections steps are as follows.

- In [33], we have shown that $P(v) \in \mathcal{N}(v)$ can be verified in $\mathcal{O}(V)$.
- Cycle detection via Algorithm 2 converges in $\mathcal{O}(\log h)$ steps and each step performs $\mathcal{O}(V)$ operations. Here h is the height of the propagation graph. Ideally we would expect h to be equal to the diameter of the graph like in the fault-free case. But in the worst case, h may be as large as $\mathcal{O}(V)$. Therefore, the cycle detection using Algorithm 2 can cost $\mathcal{O}(V \log V)$. One may replace Algorithm 2 with work-optimal algorithms such as [2] which bring the worst-case cost down to $\mathcal{O}(V)$. Yet, despite being asymptotically worse, for our specifc case, we found Algorithm 2 to be faster than [2]
- $L[v]$ is updated in constant time for each vertex resulting in $\mathcal{O}(V)$ cost.

Therefore the cost of a self-stabilizing step is $\mathcal{O}(V \log V)$ but can theoretically be as low as $\mathcal{O}(V)$. The total cost of the label propagation algorithm is $\mathcal{O}((V + E) \log d)$. So verification at every step is computationally prohibitive. Therefore we only verify if a label propagation algorithm's state is valid whenever it reports convergence.

E. Online Corrections

Verifying if the algorithm is in a valid state only when the algorithm is converged will be inefficient in cases where fault propagates to other vertices very quickly. But verifying all the conditions of theorem 3 after every iteration is computationally expensive. We overcome this issue by performing checking only a subset of conditions of theorem 3 every iteration instead of all conditions and correct them as soon as they are detected so it stops its propagation.

We can verify all the condition but the cycle-detection in $\mathcal{O}(V)$ cost for all vertices. In addition, we check for 2-cycles. In contrast to detecting cycles of all sizes, 2-cycles for any vertex can be check by verifying $v = P(P(v))$. If any of the conditions are not satisfied for any vertex, then we reset those vertices. We call these checks and correction as *online corrections*. Performing online corrections will not necessarily bring the algorithm into a valid state. Yet it can detect some of the faults and correct them before they propagate to other vertices. Performing online-corrections can significantly reduce the number of iterations to converge to correct solution.

V. Empirical Evaluation

We performed a series of experiments to test the robustness of the self-stabilizing algorithms in the Section IV. We focus on evaluating its overhead and convergence property in the presence of soft faults.

Recall, our self-stabilization algorithm can be used with either asynchronous-LP and synchronous-LP variants (Section III). We focus only on asynchronous-LP case here, which is the more challenging of the two (as a single fault can propagate to many vertices in the same iteration). For the sake of brevity, in this paper, we focus only self-stabilizing algorithms. As such we do not compare against the self-correcting algorithm which may not be able to return to a stable and valid state. We do note that from a performance perspective, the self-correcting algorithm outperforms the self-stabilizing algorithm as it requires fewer checks. From a parallel perspective, both algorithms have an equal amount of parallelism.

A. Experimental setup

Fault injection methodology: To simulate faults, we inject bit-flips in the memory operations. Specifically, we inject bit-flips in the two primary memory operation in the label propagation iteration: a) in reading the adjacency list, and b) reading labels of neighbors. Bit-flip in an array-index value may cause memory segmentation error and abort. So, we guard susceptible array-index variable with a range check. If the variable is out of range; then we change it to a random value within the range.

Using this methodology, we can control the number of bit-flips injected into the data. We test a wide range of bit-flips from a single bit-flip for one in every 4k memory reads and up to a bit-flip for one out of every 32 memory operations. These are considerably high bit-flip rates even for unreliable systems. In practice, the bit flip is thousands to millions of times lower than the ones we report for. Our goal by using the extremes values serves two purposes: 1) to show that our algorithm can perform well even on the most unreliable of systems and 2) a single bit-flip once in every billion or trillions operations might not lead to a faulty execution for the graphs that we use (though we do expect even low rates may prove to be problematic for trillion vertex graphs).

Testing framework: All our algorithms are implemented in the *C* programming language and compiled with the Intel C compiler version 17.0.0 with *O*3 optimization flags. We ran our experiments on a dual socket Ivy- bridge with 2×8cores running at 2.66GHz. This testbed had 128GB DRAM capacity and 12MB of L3 cache.

Networks: Table II list the networks used in our experiments. These networks were selected as they represent a diverse set of real-world applications and problems. They have different sparsity patterns, densities, degrees, and component distribution.

B. Competing algorithms

In our experiments, we compare the baseline (an asynchronous-LP variant of Algorithm 1) algorithm with the following fault-tolerant algorithms:

Table II: List of the networks used in our experiments. These were taken from[14].

Name	(V)	$\frac{V}{E}$
astro-ph	16706	14.5
cond-mat	16726	5.6
caidaRouterLevel	192244	6.3
Wordnet3	82,670	1.6
patents_main	240,547	2.3
cit-HepTh	27,770	12.7
bauru5727	40,366	3.6
cnr-2000	325,557	9.9

1) Triple modular redundancy (TMRLP) - discussed below
2) Self-stabilizing label propagation Algorithm (SSLP) described in Algorithm 3
3) Self-stabilizing label propagation Algorithm with online correction (SSLP-OC): SSLP augmented with online corrections described in Section IV-E

Lastly, we note that the *Baseline* algorithm is prone to faults though it can correct some of the errors by itself. Also, TMRLP is not a self-stabilizing algorithm nor does it ensure a correct result in the end.

The TMRLP *Algorithm:* The TMRLP algorithm works as follows. Three independent instances of the label propagation algorithm are executed, each one with individual state and set of data structures. In unreliable execution, each of three algorithms executes a single iteration of the label-propagation algorithm. After the label-propagation, the label $L[v]$ is checked across the different instances for all the vertices in the graph. If all the three $L[v]$ values differ then we reset that label to: $L[v] = v$. If two copies of $L[v]$ are same, and only one is different, then we correct the different value. If all three values are identical, then we do not change the label for any copy. The TMRLP algorithm performs an additional $2\times$ redundant computation and needs an additional $2\times$ storage.

C. Failure testing analysis

We say that an algorithm has *failed* when it gives an incorrect result. Failure tests aim to quantify the frequency of the failure event, independent of any incurred overhead. We do so by executing each algorithm many times for a given network at a given fault injection rate in simulated random fault environment. Specifically, we report statistics for 100 different executions. We compare the four algorithms on the success rate at different fault injection rates. We vary the fault injection rates from $\{2^{-12}, 2^{-11}, \ldots 2^{-5}\}$ bit-flips per memory operation per iteration of label-propagation algorithm.

If we do not limit the number of iterations, both the self-stabilizing variants, the SSLP and the SSLP-OC algorithm, will give correct results. But, this doesn't reflect the practical use of the algorithm. Thus, in our experiments, we limit the number of iteration to 100. If an algorithm doesn't converge by 100 iterations, it aborts and reports failure.

We show the results of the failure-test in Fig. 6, where each subfigure represents the failure rate for a different network. In each subfigure, the abscissa depicts the fault injection, from 2^{-12} to 2^{-5} bit-flips per edge. Note the abscissa is log-scale. The ordinate shows the percentage of times an algorithm failed for a given fault injection rate. So, a lower curve is preferable as this means fewer tests failed.

In 7 out of 8 cases, both SSLP and SSLP-OC are more resilient than the baseline and the TMRLP . In the top row of Fig. 6, our new self-stabilization outperforms both baseline and TMRLP for all four

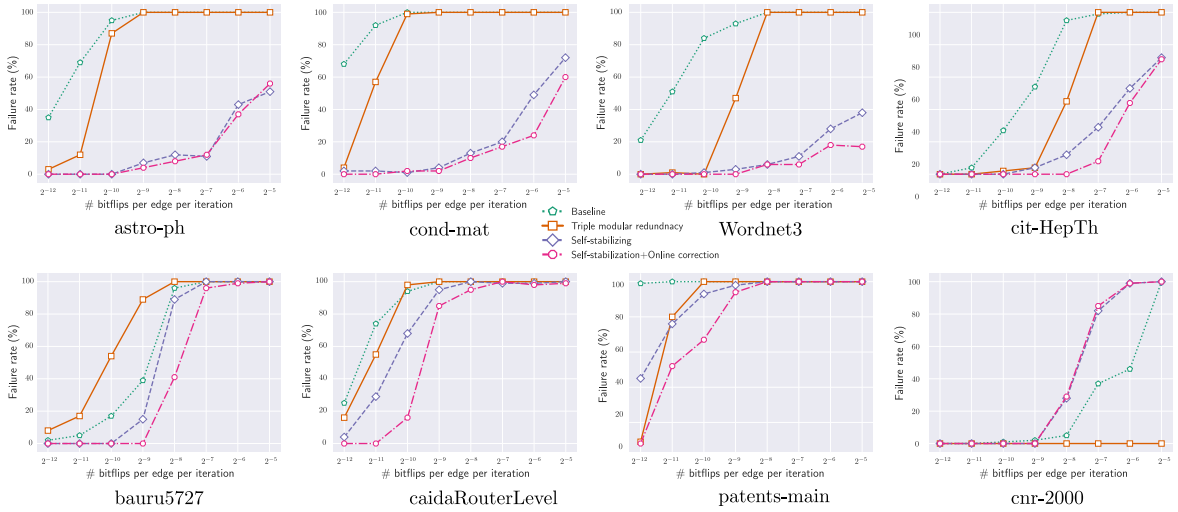


Figure 6: Failure rates (y-axis) with respect to fault injection rate (bitflips per iteration edge) for different networks.

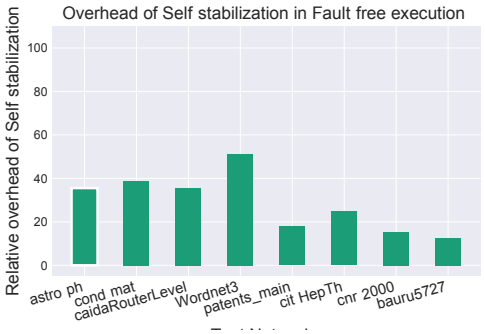


Figure 7: Overhead of the correction step for the self stabilizing algorithms in comparison with a fault-free execution. Overhead is measure in the additional execution time required by our self stabilizing algorithm over a fault-free execution. For most networks the overhead is less than 40% and on average is close to 30%

networks. In the bottom row of Fig. 6 we see that the four algorithms behave similarly, yet have slightly different *breaking-points* fault rates. We have yet to determine the cause of the behavior for cnr-2000 and why the new self-stabilizing algorithms are not performing as well as TMRLP and the baseline.

D. Overhead of Correction step

The self-stabilization step adds 10-40% time to the execution of label-propagation algorithm - shown in Fig. 7. The overhead show in Fig. 7 includes both the self-stabilization step and one label propagation iteration, which we perform to detect convergence. The cost of self-stabilization step is $\mathcal{O}(V \log V)$, whereas the cost of the entire label-propagation algorithm is $\mathcal{O}((V+E)\log V)$. Thus, the relative cost of self-stabilization is higher for the networks where $\frac{E}{V}$ is small. For instance, the highest overhead we incur is for *wordnet3*, which also has the smallest $\frac{E}{V}$ among the matrices we tried. Even in the worst case, self-stabilization overhead is lower than redundancy based fault tolerant approach such as TMRLP , where the smallest memory and computation overhead is 200%.

VI. RELATED WORK

Since Dijkstra's formalization of self-stabilizing systems, the concept has been applied primarily in distributed computing when analyzing stability [16]. In this setting, algorithms concern networking protocols and consensus protocols [7], [1]. In one of the seminal works in this area Awerbuch and Varghese [5] show that local algorithms can be made self-stabilizing by maintaining a log of the state transitions. They also introduce the first end-to-end communication protocol [4] using simple local checking and corrections. This work along with [1] introduced the notion that the protocol should maintain sufficient information at each node so that illegal states can be detected *locally* by every node. Local checking and correction is a desirable property, as it avoids a costly global synchronization for achieving self-stabilization. Sensor networks are also a distributed system where faults may occur frequently, and self-stabilization is a preferred way to achieve fault-tolerance and stability [7].

Some graph algorithms already have a self-stabilizing counterpart [22], [18], [12], [9], [26], [28], [13]. Breadth-first search is another method for the connected-component problem. Huang and Chen [9] describe a self-stabilizing algorithm for constructing breadth-first trees. Jacob et al. [26] invented a self-stabilizing algorithm to maintain skip graphs which are used extensively in peer-to-peer networks. They can recover the "skip graph" from an invalid state in $\mathcal{O}(\log^2 V)$ synchronous rounds of reliable computing. Recently, a fault correcting framework based on self-stabilization has been added to the distributed memory D-Galois graph analytics system [13]. They classify various types of correction classes for various graph algorithms and derive recovery steps for each class.

While connected components is an algorithm covered in Phoenix the recovery method developed is adapted to the distributed memory setting and not for shared memory corrections outlined here. They handle the case when entire hosts fail and algorithm has to be computed on the remaining hosts. They design a recovery process by which only the failed hosts' computations are recalculated rather than recomputing every machines work since the time of failure. This is done by rerunning the full LP algorithm and resetting the labels of the failed vertices to the initial label. We improve on this approach by resetting to $\min(v, \mathcal{A}(v))$ which can save some computations. Our approach further optimizes the cost of checking validity and the correction steps which are not considered in Phoenix. We believe the contributions in this work can be used to augment Phoenix to speed up corrections in the distributed setting as well.

The idea of self-stabilization for designing fault-tolerant high-performance computing algorithms is relatively recent. One reason

may be that the principle of self-stabilization is a weak form of fault-tolerance: it guarantees a return to a valid state but does not bound the overall time for computation. Nevertheless, there have been a few successful demonstrations for fault-tolerant scientific computing [34], [35], [11], [32], [8], [19], [3].

Besides the self-stabilization framework, there are many other ways to add fault-tolerance to an algorithm. A popular one, which arises in linear algebra, is linear checksum-based redundancy, both for dense [41], [27], [30], [29], [21], [17] and for sparse computations [36], [10]. Typically, the checksum-based method can be very efficient at a given fault rate; however, the overhead usually increases linearly with increasing fault rate. For hard faults, another popular choice is to use checkpoint and restart schemes, which can be combined with methods for soft faults [31], [6].

Besides the algorithm-based methods, many compiler and runtime system methods can automate the process of adding resilience to a computation [25], [24], [39], [23]. We view these as complementary to algorithm-specific methods like the ones proposed in this paper.

VII. Conclusion and Future Work

This paper's self-stabilizing method for connected components goes significantly beyond the self-correcting algorithm presented previously [33]. In particular, self-correcting algorithms need a previously correct and valid state to start from when recovering from a transient fault, whereas the new self-stabilizing method does not need a previously valid point. Moreover, unlike the self-correcting algorithm, our new algorithm is able to detect an invalid state even for the asynchronous algorithm, which does not have a previous state to resort to when an error occurs. Thus, self-stabilization adds an important new capability.

Our experiments show that the self-stabilizing algorithm performs well in the face of extremely high error rates (i.e., when up to 1 out of every 32 reads being faulty). In most cases, our new algorithms outperformed TmrLp , a triple redundancy-based approach, both in their ability to reach a valid and correct solution and also in computational and storage overhead. While TmrLp introduces a 200% overhead to the execution time, our self-stabilizing algorithm on average introduces an average of 30% overhead.

Future Work

While we showed how to apply self-stabilization to label propagation in connected components, we still do not know if there are other algorithms that can benefit from a self-stabilizing formulation. Our self-stabilizing mechanism doesn't assume anything about the state, i.e., an invalid state due to a fault can be arbitrarily bad. However, it is possible that under a different fault model, one can narrow down the set of valid and invalid states considerably and design correction step only for this set.

Additionally, the self-stabilization mechanism can be "re-purposed." In particular, it could help update components in the case of a *dynamic* graph, where the structure and connectivity of the graph are undergoing change. From the lens of self-stabilization, any such live updates make an existing state invalid. Thus, if a correction step can be found, one need not re-run the algorithm from scratch; instead, one can use the correction to obtain some valid state for the new graph prior to re-running, which may accelerate convergence to an updated solution using fewer iterations than starting from scratch. While we did not treat this idea here, the key observation is that the self-stabilization, as a design principle, may have uses that go far beyond resilience.

VIII. Acknowledgement

This work is supported by the Early Career Research Program of the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, with program managers Lucy Nowell and Robinson Pino, under contract number DE-AC05-00OR22725.

References

[1] Afek, Y., Kutten, S., and Yung, M. Memory-efficient self stabilizing protocols for general networks. In *International Workshop on Distributed Algorithms* (1990), Springer, pp. 15–28.

[2] Anderson, R. J., and Miller, G. L. Deterministic parallel list ranking. In *Aegean Workshop on Computing* (1988), Springer, pp. 81–90.

[3] Anzt, H., Dongarra, J., and Quintana-Ortí, E. S. Tuning stationary iterative solvers for fault resilience. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (2015), ACM, p. 1.

[4] Awerbuch, B., Patt-Shamir, B., and Varghese, G. Self-stabilization by local checking and correction. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on* (1991), IEEE, pp. 268–277.

[5] Awerbuch, B., and Varghese, G. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on* (1991), IEEE, pp. 258–267.

[6] Benoit, A., Cavelan, A., Robert, Y., and Sun, H. Multi-level checkpointing and silent error detection for linear workflows. *Journal of Computational Science* (2017).

[7] Cappello, F. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications 23*, 3 (2009), 212–226.

[8] Carpenter, G. A., and Grossberg, S. The art of adaptive pattern recognition by a self-organizing neural network. *Computer 21*, 3 (1988), 77–88.

[9] Chen, N.-S., Yu, H.-P., and Huang, S.-T. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters 39*, 3 (1991), 147–151.

[10] Chen, Z. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 167–176.

[11] Coleman, E., and Sosonkina, M. Self-stabilizing fine-grained parallel incomplete lu factorization. *Sustainable Computing: Informatics and Systems* (2018).

[12] Collin, Z., and Dolev, S. Self-stabilizing depth-first search. *Information Processing Letters 49*, 6 (1994), 297–301.

[13] Dathathri, R., Gill, G., Hoang, L., and Pingali, K. Phoenix: A substrate for resilient distributed graph analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, ACM, pp. 615–630.

[14] Davis, T. A., and Hu, Y. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS) 38*, 1 (2011), 1.

[15] Dijkstra, E. W. Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 41–46.

[16] Dolev, S. *Self-stabilization*. MIT press, 2000.

[17] Du, P., Bouteiller, A., Bosilca, G., Herault, T., and Dongarra, J. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices 47*, 8 (2012), 225–234.

[18] Ghosh, S., and Karaata, M. H. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing 7*, 1 (1993), 55–59.

[19] Göddeke, D., Altenbernd, M., and Ribbrock, D. Fault-tolerant finite-element multigrid algorithms with hierarchically compressed asynchronous checkpointing. *Parallel Computing 49* (2015), 117–135.

[20] Hoemmen, M., and Heroux, M. A. Fault-tolerant iterative methods via selective reliability. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society* (2011), vol. 3, p. 9.

[21] Huang, K.-H., and Abraham, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers 100*, 6 (1984), 518–528.

[22] Huang, S.-T., and Chen, N.-S. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters 41*, 2 (1992), 109–117.

[23] HUKERIKAR, S., DINIZ, P. C., AND LUCAS, R. F. Programming model extensions for resilience in extreme scale computing. In *European Conference on Parallel Processing* (2012), Springer Berlin Heidelberg, pp. 496–498.

[24] HUKERIKAR, S., DINIZ, P. C., LUCAS, R. F., AND TERANISHI, K. Opportunistic application-level fault detection through adaptive redundant multithreading. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on* (2014), IEEE, pp. 243–250.

[25] HUKERIKAR, S., TERANISHI, K., DINIZ, P. C., AND LUCAS, R. F. An evaluation of lazy fault detection based on adaptive redundant multithreading. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE* (2014), IEEE, pp. 1–6.

[26] JACOB, R., RICHA, A., SCHEIDELER, C., SCHMID, S., AND TÄUBIG, H. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proceedings of the 28th ACM symposium on Principles of distributed computing* (2009), ACM, pp. 131–140.

[27] JOU, J.-Y., AND ABRAHAM, J. A. Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures. *Proceedings of the IEEE 74*, 5 (1986), 732–741.

[28] KORMAN, A., KUTTEN, S., AND MASUZAWA, T. Fast and compact self stabilizing verification, computation, and fault detection of an mst. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (2011), ACM, pp. 311–320.

[29] LUK, F. T., AND PARK, H. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing 5*, 2 (1988), 172–184.

[30] LUK, F. T., AND PARK, H. Fault-tolerant matrix triangularizations on systolic arrays. *IEEE Transactions on Computers 37*, 11 (1988), 1434–1438.

[31] MANDAL, P. S., AND MUKHOPADHYAYA, K. Self-stabilizing algorithm for checkpointing in a distributed system. *Journal of Parallel and Distributed Computing 67*, 7 (2007), 816–829.

[32] OBORIL, F., TAHOORI, M. B., HEUVELINE, V., LUKARSKI, D., AND WEISS, J.-P. Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on* (2011), IEEE, pp. 144–153.

[33] SAO, P., GREEN, O., JAIN, C., AND VUDUC, R. A self-correcting connected components algorithm. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale* (2016), ACM, pp. 9–16.

[34] SAO, P., AND VUDUC, R. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (2013), ACM, p. 4.

[35] SAO, P. K. *Scalable and resilient sparse linear solvers*. PhD thesis, Georgia Institute of Technology, 2018.

[36] SHANTHARAM, M., SRINIVASMURTHY, S., AND RAGHAVAN, P. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM international conference on Supercomputing* (2012), ACM, pp. 69–78.

[37] SHILOACH, Y., AND VISHKIN, U. An o (logn) parallel connectivity algorithm. *Journal of Algorithms 3*, 1 (1982), 57–67.

[38] TARJAN, R. Depth-first search and linear graph algorithms. *SIAM journal on computing 1*, 2 (1972), 146–160.

[39] WADDEN, J., LYASHEVSKY, A., GURUMURTHI, S., SRIDHARAN, V., AND SKADRON, K. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 73–84.

[40] WYLLIE, J. C. The complexity of parallel computations. Tech. rep., Cornell University, 1979.

[41] YAO, E., ZHANG, J., CHEN, M., TAN, G., AND SUN, N. Detection of soft errors in lu decomposition with partial pivoting using algorithm-based fault tolerance. *International Journal of High Performance Computing Applications 29*, 4 (2015), 422–436.