A Self-Stabilizing Connected Components Algorithm

Piyush Sao
Georgia Institute of Technology
Atlanta, GA
piyush3@gatech.edu

Richard Vuduc
Georgia Institute of Technology
Atlanta, GA
richie@gatech.edu

1. INTRODUCTION

We would like to reliably compute connect components in a graph in presence of soft faults. Connected components is an important graph kernel with applications in community detection, metagenomics and parallel processing of a graph. The increased number of computing elements have also increased the rate of hardware malfunctioning, also known as faults. Soft faults, which do not cause the application to crash, give incorrect results to the computation, often masquerading as correct results. Soft-faults pose a great threat to the reliability of highly parallel computing systems. It is widely accepted that dealing with faults, is no longer just a hardware issue. Algorithm designer must redesign algorithm so that it may, at the very least, differentiate between incorrect and correct results. However, the challenge is, very often it is not possible to check the correctness of the results, without recomputing the solution. Under these constraints, how can we design a connected component algorithm which gives a correct output in spite the presence of soft-faults?

Self-correction para

- Introduce self-correction LP (prior to this)
- what is self-correction
- define valid state and invalid state:
- limitations of self-correction LP

Prior to this, we presented a fault tolerant label propagation algorithm (sclp) based on ideas of self-correction in [cite]. Both self-correcting and self-stabilzing algorithm share the notion of state, and distinguish between valid and invalid states. Briefly, a state of an algorithm is a subset of intermediate variables that allows to resume the algorithm. A state is called valid if an algorithm starting the state will converge to correct solution in fault free execution, otherwise invalid. In a fault free execution, an algorithm starts from a valid, remains in a valid state throughout the execution, and finally converge to a valid solution state. By contrast, in presence of hardware faults may bring the algorithm to a invalid state and eventually algorithm will fail. A self-correcting algorithm works by bringing the algorithm to a valid state by assuming it started from a previously known valid state.

Self-stabilization para

- Introduce self-stabilization:
- advantage of SS over SC
- SSLP as alternative to SCLP

Self-stabilization was introduced by Dikstra in 1973 in context of distributed control. Briefly, an algorithm is self-stabilizing if starting from any arbitrary state, valid or invalid, algorithm comes to a valid state in finite number of steps. We previously showed potential of self-stabiling algorithm for constructing fault tolerant numerical iterative algorithms [cite]. Self-stabilization is arguably more desirable property as it does not assume anything about history of execution. However, self-stabilized formulation of every algorithm may not exists.

how self-stab LP works

- idea of correction step
- difficulty in constructing a correcting step: vertex centric
- correction step
- advantage of correction step

To construct a self-stabilizing label propagation algorithm, we first analyzed states of the label propagation algorithm and developed a set of sufficient conditions that ensures if the state is valid. We present an efficient correction step that verifies if the state is valid, and brings it to a valid state if the state is invalid. Since label propagation converges in very few iterations, we only run the correction step when the algorithm reports convergence. If algorithm is not in a valid state when it reports convergence then the correction step constructs a valid state and restart the computation from this valid state. Running label propagation from this valid state takes significantly fewer iteration to converge, which is significantly efficient to restarting the algorithm.

summary of results

- what is overhead of correction step
- overhead
- comparison parameter

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Algorithm 1 Label propagation algorithm

Require: $G = (V, E)$
1: Initialization ;
2: **for** each $v \in V$ **do**
3: $CC^0[v] \leftarrow v; P[v] \leftarrow v;$
4: $NumChange \leftarrow |V|,\ i \leftarrow 0$
5: **while** $NumChange > 0$ **do**;
6: $NumChange \leftarrow 0$
7: **for** each $v \in V$ **do**
8: **for** each $u \in E(v)$ **do**
9: **if** $CC^i[u] < CC^{i+1}[v]$ **then**
10: $CC^{i+1}[v] \leftarrow CC^i[u]$
11: $P[v] \leftarrow u$
12: $NumChanges \leftarrow NumChanges + 1;$
13: $i \leftarrow i + 1$

Table 1: Symbols used in the fault free LPalgorithm and in the new fault tolerant algorithm.

Symbol	Decription	Size
V	Vertices in the graph	O(V)
E	Edges in the graph	O(E)
$adj(v)$	Adjacency list for vertex $v \in V$	
CC	Connected component array	O(V)
CC^i	Connected component array $_i$ after iteration	O(V)
CC^∞	The final connected component mapping upon algorithm completion.	O(V)
	Fault free	
H		O(V)

• result

Our self-stabilizing label propagation algorithm requires $\mathcal{O}(V)$ additional storage and correction step requires $\mathcal{O}(Vlog(V)$ computation, which is asymptotically a smaller fraction of cost of label propagation algorithm $\mathcal{O}((V + E)log(V))$. We tested fault tolerance properties of self-stabilizing label propagation algorithm a number of representative test problems. Self-stabilizing label propagation algorithm is significantly more efficeint than existing fault tolerance techniques such as triple modular redundency (TMR). In particular, Self-stabilizing label propagation algorithm takes only 20% additional iteration even in presence of 2^{-9} bit flips per memory iteration.

2. CONNECTED COMPONENTS

Given a undirected Graph $G = (V,\ E)$, a connected-component C is a subset of vertex V which satisfies the following. A) there is a path between any two vertices of C in the parent 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 in a graph. This problem is known as graph connected component problem.

We focus on the highly parallel label-propagation algorithm for the graph connected-component problem. We describe the working of label propagation algorithm relevant to our discussion later.

Label Propagation algorithm in Fault Free execution.

The label propagation algorithm marks each vertex with a label that identifies its component. The identifying label can be the minimum vertex-id in the component. The label-marking process is iterative. In every iteration, each vertex computes the minimum label amongst itself and its neighbors. So the minimum label propagates to all the vertex in the component. The algorithm stops when every vertex in the graph has acquired its final label.

The label propagation algorithm keeps an array CC of current labels of all vertices. For each vertex v, its label $CC[v]$ is initialized with its vertex-id $CC[v] = v$. In every iteration, each vertex updates its label by calculating minimum label of all its neighbors and itself:

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

Where $\mathcal{N}(v) = \{v, adj(v)\}$ is the defined as immediate neighbourhood of v. So the minimum vertex-id propagates to all the vertices in the connected component. The iteration converges when there are no more label changes in the graph. We show the pseudo-code for label propagation algorithm in Algorithm 1. In Algorithm 1, we also update the *parent* array $P[v]$, which tracks the vertex which causes the last change in the label of v.

We can implement Equation (1) can in two ways. In a first way, we use two different arrays to store CC^{i+1} and CC^i. We refer to this implementation is synchronous-LP algorithm. In another way, we overwrite CC^{i+1} on CC^i. We refer to this version as asynchronous-LP algorithm. Depending on architecture and programming model, the two variants may have different performance characteristics. In the subsequent discussion, we assume asynchronous-LP algorithm. Yet, our results are equally applicable to both instances of the LPalgorithm.

Cost of Label-propagation Algorithm: Each iteration of LP, we visit all the vertex and edges once and thus costs $\mathcal{O}(V + E)$. The LP algorithm requires $\mathcal{O}(d)$ iterations to converge, where d is the diameter of the graph. We may use short-cutting to bound the number of iteration to $\mathcal{O}(log(d))$ [insert citation]. However, in practice asynchronous-LPalgorithm only takes a few more iteration than implementing full short cutting step, without the cost of short cutting.

3. FAULTS IN COMPUTING SYSTEMS

We term fault as any instance of hardware deviating from its expected behavior. Impact of such faults on the application can vary depending on the location of the fault. Based on the impact of the faults, they can be classified into two broad categories: hard fault and soft faults. A hard fault causes the application to terminate prematurely. For instance, failing of nodes and network fall into this category. On the other hand, soft faults may not cause the application to abort. Even so, a soft fault can lead to an incorrect result, which we term as the failure. Bitflips in memory and latches are examples of soft faults. Interested readers can find a more detailed discussion on faults elsewhere [1].

In this paper, we only deal with soft faults. A particularly insidious manifestation of soft-faults is silent data corruption. Silent data corruption occurs when a soft fault leads to corruption of entire intermediate variables, without notifying the application. In such cases, the application may arrive at an incorrect solution and yet, report it as correct solution. Thus, silent data corruption can lead to serious reliability issues in the computing.

The importance of having algorithmic level fault tolerance has already been explored for a number of numerical com-

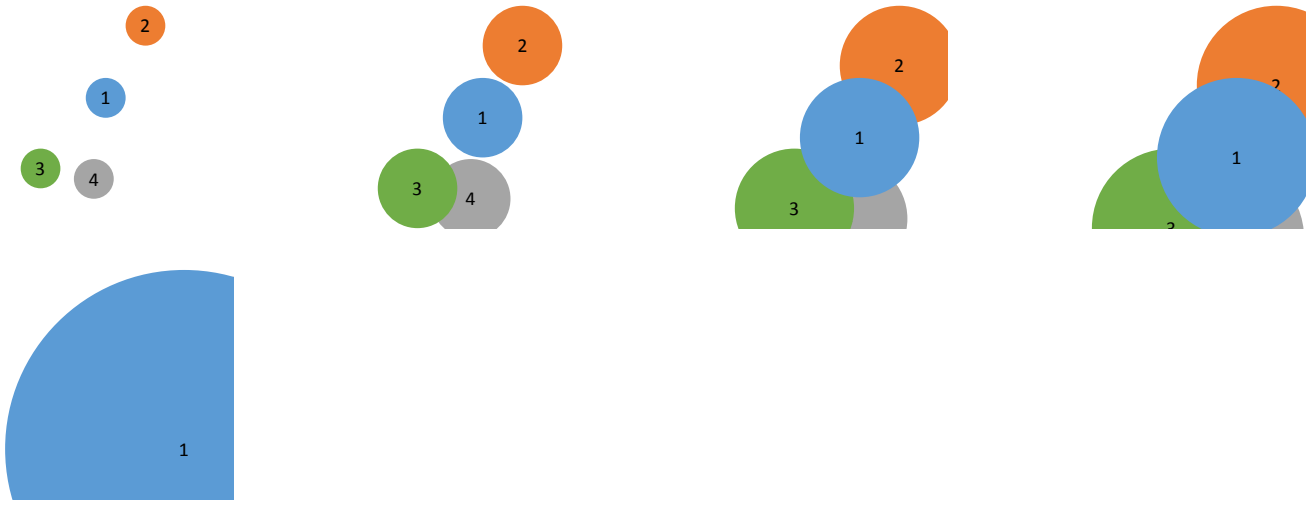

Figure 1: These sub-figures conceptually show how the connected component id propagates through the graph as time evolves. Subfigures represent snapshots of the algorithm at different times. For simplicity, this example assumes that all the shown vertices are connected. Initially the number of connected components is equal to the number vertices. (a) Depicts the initial state in which each vertex is in its own component. (b)-(d) depict that some vertices belong to the same connected component yet may require multiple label updates (in either the same iteration or a separate iteration). (e) is the final state in which there is a single connected component.

putations (see Section 8). Graph computations are equally susceptible to soft faults as numerical computation. Researchers have started looking at resilient discrete computation only recently (**1. cite**)

4. ALGORITHMS AS STATE-MACHINES

In this section, we discuss how abstraction of an algorithm as a state machine may help us to make them more resilient.

An iterative algorithm can also be viewed as a system with states and transition rules. Its *state* is a subset of the intermediate variable which enables continued execution. For instance, the vector CC in Algorithm 1 alone is sufficient to restart the algorithm from any checkpoint. In general, the state of the algorithm can have or can be augmented to have some form of redundancy built in it. Algorithm 1, the variable P acts as an useful redundancy.

Valid and invalid states.

A state of the algorithm is said to be in a valid state if the algorithm will converge to a correct solution in fault-free execution starting from this state, otherwise invalid. In previous work **2. cite**, we have shown that such abstraction may help us to construct resilient algorithms.

Impact of hardware fault on algorithms state.

If an algorithm is well suited for a given problem then in a fault-free execution, its state remains valid during entire computation. However, soft-fault such as bit flip can corrupt the intermediate variables and, can potentially bring it to an invalid state. Subsequent fault-free execution will lead to an incorrect solution, thus failure, unless it is brought to a valid state by some mechanism. Our principle for designing fault-tolerant algorithm is based on the idea of augmenting the algorithm so that it can bring itself to a valid state, by design. We distinguish between following two principles for bringing the algorithm to a valid state.

Self-stabilizing algorithm.

Formally, a system is said to be self- stabilizing, if starting from any arbitrary *state*, it comes to a valid state in a finite number of steps[2]. 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 later computations do not encounter any fault.

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 algorithm, a self-correcting must start from a valid state. In reality, it is not such a limitation as most algorithms by design starts 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. In [cite], we presented a self-correcting of SYNC version of label propagation algorithm.

5. VALID AND INVALID STATES OF LP ALGORITHM

5.1 Impact of faults in LP algorithm

Theorem 1. ***Valid States:*** *A connected component array CC is a valid state—i.e., a fault-free execution of algorithm starting from CC will converge to the correct solution—if, for all vertices v, $CC^{\infty}[v] \leq CC[v] \leq v$ ([3]).*

5.2 Self-correcting LP algorithm

The problem with Theorem 1 is that it defines the validity of the based on the final and correct output. Thus, it can not be used to check the validity of the state. In our previous work [3], we presented the following set of conditions that can be used to check the validity.

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

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

The conditions of the Theorem 2 can be verified in $\mathcal{O}(1)$ time for any vertex v [3]. Thus, we can check validity for all vertices in $\mathcal{O}(V)$ time. We can cheaply recompute the labels for the vertices which do not satisfy Theorem 2 using the valid previous state $CC^{i-1}[v]$.

The Theorem 2 assumes and can only work correctly when the previous state CC^{i-1} is valid. In the *self-correcting* label propagation algorithm, we verify conditions of Theorem 2 after every iteration to ensure that the algorithm is in a valid state; and Theorem 2 can be used to detect and correct next invalid states.

5.2.1 Limitations of Self-correcting LP algorithm

The *self-correcting* algorithm can only work when we have the copy of previous valid labels CC^{i-1}. Thus, the *self-correcting* LPalgorithm will not work with ASYNC formutation of the LPalgorithm.

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.

5.3 Self-stabilizing Validity Conditions

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

The key idea here is, that information about past iterations are already present in $S = \{CC, p\}$. Specifically, the P contains the information about how a label has propagated to vertex v. To put it concretely, we present following properties of P.

5.3.1 Relation between $CC[v]$ and $CC[P(v)]$

In the fault-free execution of LPalgorithm, label of a vertex $CC[v]$ is always greater than orequal to its parent's label $CC[P(v)]$. Any vertex v acquires its label from its $P(v)$ in some iteration $j \leq i$. So the current label of vertex v is $CC^i[v] = CC^j[P(v)]$. In the LPalgorithm, labels of any vertex can only decrease as the iteration progresses. Thus, the current $P(v)$'s current label $CC^i[P(v)]$ must be smaller than or equal to its ealier value $CC^j[P(v)]$. Therefore, $CC^i[v] = CC^j[P(v)] \geq CC[P(v)]$.

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

Additionally, if any vertex is the parent of itself, i.e. $P(v) = v$ then $CC[v] = v$. In Algorithm 1, all the vertices have initialized with $P(v) = v$ and $CC[v] = v$. If a vertex never changes it's label during LP iterations, then it will retain its parent and label. Conversely, if a vertex has changed its value then $CC[v] < v$ 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 $CC[v]$ is changed. So $P(v) = v$ implies v obtained a label from itself, which is not possible. Therefore, in fault free execution of Algorithm 1 we must have following for all vertices v:

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

Definition 1. Propagation Graph H: For a given state $S = \{CC, P\}$ for 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 $v \in V$.

5.3.2 Structure of the Parent Graph H

In the fault-free execution of Algorithm 1, the propagation graph H does not contain any loops besides self-loop. Thus, H consists of multiple trees. The label of roots of each tree is a *local- minima* of labels. When the iteration is converged, H is one tree of 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.

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

Theorem 3. *Starting from state $S = \{CC, P\}$, the Algorithm 1 will converge to correct solution for graph $G = \{V, E\}$, if S satisfies the following conditions.*

1. $CC[v] \leq v$ *for all* $v \in V$;
2. $P(v) \in \mathcal{N}(v)$ *for all* $v \in V$;
3. $CC[P(v)] \leq CC[v]$ *for all* $v \in V$;
4. $CC[v] = v \iff P(v) = v$ *for all* $v \in V$; *and*
5. *Directed graph described by parent array $H = (V, E_H)$, where $E_H = \{(v, P(v)), \forall v \in V\}$ describes a forest.*

6. SELF-STABILIZING LABEL PROPAGATION ALGORITHM

In this section, we describe how do we use the Theorem 3 to verify whether a state is valid. Specifically, the challenge is in verifying the condition-5 of Theorem 3 in parallel. Further, we also need a mechanism to bring the algorithm to a valid state if we detect an invalid state.

Except for the condition-5 of the Theorem 3, we can verify all other conditions *locally* and in parallel. Here, the term local means by only using information that a vertex has access to. Condition 1-4 can be verified for any vertex by looking at it's and it's neighbor's label and parent. Whereas we can not detect a loop just by looking at a vertex and it's neighbors.

The traditional algorithms for finding a loop in a directed graph are not well suited for vertex-centric programming. The problem of finding the loop in a directed graph is also

Table 2: List of the matrices

Name	(V)	$\frac{V}{E}$
Kron_simple_500 log 18	262,144	80.7
rgg(2,18)	262,144	11.8
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
web-Google	916,428	5.5
cit-HepTh	27,770	12.7
web-berkstan	685,230	11.0
mouse-gene	45101	642.2

known as the problem of finding strongly connected components in a graph. In sequential case, one can use Tarzen's strongly connected component algorithm, or breadth search first(BFS), or depth search first (DFS)

$$CITE$$

. 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 $\mathcal{O}(|V|)$. As other checks of Theorem 3 costs $\mathcal{O}(|V|)$ so $\mathcal{O}(|V|)$ cost of loop detection is, in theory, acceptable. Yet due to limited parallelism and inability to express in the vertex-centric model, we can not use these to verify the condition-5 of the Theorem 3.

7. EMPIRICAL RESULTS

We performed a series of experiments to test the robustness of the self-stabilizing algorithms in the sec[x]. We focus on evaluating overhead and convergence property in presence of soft faults.

While there is not much different in applying self-stabilization to Async and Sync vriants, we focus only on Async case here. Async case is considerably more difficult than Sync case, as in Async case a single fault can propagate to multiple vertices in a single iteration. Moreover, Async case keeps only a single copy of the state, so self-correction approach taht works well on Sync case, will not work on Async case[1].

7.1 Experimental set-up

7.1.1 Test-bed

We prototyped our implementation in C and compiled with Intel C compiler (insert compiler version) with $O3$ optimization flages. We ran our experiments in a dual socket Ivy-bridge with 2×8cores running at 2.66GHz. This testbed had 128GB DRAM and 12Mb of L3.

7.1.2 Test-networks

We choose networks from real applications with diverse sparsity pattern, density, degree distribution and component distribution. In table[x], we list the networks along with some properties.

7.1.3 Fault injection methodology

We inject bitflips in memory operation to simulate faults.Specifically, we inject bitflips in two main memory operation: a) reading adjacency tree, and b) reading labels of neighbours. Bitflip in an array-index value may cause memory segmentation error and abort. Therefore, we gaurd susceptible array-index variable with a range check. If the variable is out of range, we change it to random value within range.

[1]We compared the self-correcting and the self-stabilizing algorithm for Sync algorithm, and self-correcting label propagation performs definitively better.

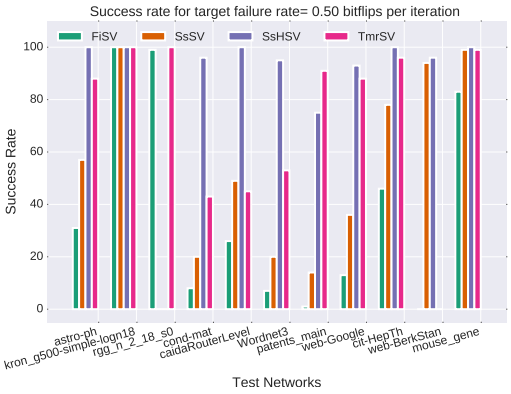

Figure 2: Success rate for different different test networks

7.1.4 Competing algorithms

In our experiments, we compare the following representative fault tolerant algorithms:

1. Baseline: Algm(x)
2. TMR: Triple modular redundancy
3. SsLP: Self-stabilizing LP iteration without any checks.
4. HSsLP: Self-stabilizing LP iteration with unreliable checks.

7.2 Failure Test

The aim of failure test is to quantify the frequency of the event when an algorithm fails to give the correct results, indepedent of any incurred overhead. This is done by executing each algorithm multiple times for a given network in simulated random fault environment with different seeds for random number generation. We compare the four algorithms based on the success rate at a specific fault rate. The fault rate is chosen from $\{2^{-5}, 2^{-6}, \ldots 2^{-20}\}$ bitflips per memory operation at which TMR fails for roughly 50% of the trials.

Note that SS SSH LP algorithm will almost always give correct results if we do not limit number of iterations to converge. However, 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 iteration, it aborts and reports failure.

In fig(x), we show the success rate of different algorithm for different graphs at TMR50. Note that this fault inject rate so high that traditional redundancy based fault tolerance algorithm will fail in ˜50% of the cases. In fig(x), we see that SShSv has a success rate of more than 90% in 9 out of eleone cases. SShSv is better than TMR in 8 out of 11 cases.

There are two graphs kron_g and mouse_gene where all the algorithm have very good success rate. This is typically the case when the graph is relatively dense. RGG is one random graph where Baseline and TMR have 100% success

rate but both SsSv and SShSV performs extremely poorly. This usually happens when a graph has a lot of tree like structure. In such cases even though sssv and sshsv have converged to correct solution, they may still find a 2-loop

7.3 Convergence Test

7.4 Overhead of Correction step

8. RELATED WORK

Here goes the related work [4].

9. CONCLUSION AND FUTURE WORK

conlusion

- what did we do
- what are implications
- where do we go from here

10. REFERENCES

[1] M. Hoemmen and M. A. Heroux, "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*, vol. 3, p. 9, 2011.

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

[3] P. Sao, O. Green, C. Jain, and R. Vuduc, "A self-correcting connected components algorithm," in *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, pp. 9–16, ACM, 2016.

[4] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 69–78, ACM, 2012.

(a) Histogram of iteration distribution

(b) Cumulative success rate with respect to addition iteration to converge

Figure 3: Success rate for different different test networks

Figure 4: Overhead of correction step for different networks