A Self-Stabilizing Connected Components Algorithm

Piyush Sao
Georgia Institute of Technology
email: piyush3@gatech.edu

Richard Vuduc
Georgia Institute of technology
email: richie@gatech.edu

I. 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
- 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

Table I: 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 after iteration i	O(V)
CC^∞	The final connected component mapping upon algorithm completion.	O(V)
	Fault free	
H		O(V)

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.

II. Connected components

a) Label Propagation algorithm in Fault Free execution: The aim of label propagation algorithm is to mark every vertex in a given connected component with a predefined common label. This common label is same within each connected component, and different across different connected component. There are many choice for the common label. As a convention, we consider minimum vertex-id in the connected component as the final common label.

The label propgation algorithm keeps an array CC of current labels for all vertices. For each vertex v, its label $CC[v]$ is initalized with its vertex-id $CC[v] = v$. In every iteration, each vertex updates its label by calculating minimum label of all its neighbours 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 immidiate neighbourhood of v. Thus, through out the iteration minimum vertex id propagates to all the vertces in the connect component. The iteration converges when there are no more label changes in the graph.

Depending on the constrints, equation **??** can be implmented in two ways. In the first way, we use two different arrays to store CC^{i+1} and CC^{i}. We refer to this implementation is Sync LP algorithm. In an another way, we overwrite CC^{i+1} on CC^{i}. We refer to this version as Async LP algorithm. Depending on architecture and programming model, the two variants may have different performance characteristics. In subsequent discussion, we assume Async LP algorithm. Yet, our results are equally applicable to both instance of the LP algorithm.

Each iteration of LP, visits 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 async LP algorithm only takes a few more iteration than implementing full short cutting step, without the cost of short cutting.

III. Impact of hardware faults on LP algo

We term fault as any instance of hardware deviating from its expected behiour. Impact of such faults on the application can vary significantly depending on location of the fault. Based on impact of the faults, they can be classified into two braod categories: hard fault and soft faults. A hard fault causes the application to abort for instance failing of nodes and network fall into these categories. On the other hand, soft faults may not cause application to abort. Nevertheless, a soft fault can potentially lead to an incorrect result, which we term as failure. Bitflips in memory and latches are example of softfaults. A more detailed explaination of types of faults can be found elsewhere [Hommen,heroux].

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 intermidiate variables, without notifying the application. In such cases, application may arrive at an incorrect solution and yet, inform user as correct solution. Thus, silent data corruption can lead serous reliablity issues in the computing.

The importance of having algorithmic level fault tolerance has already been explored for a number of numerical computations[citation]. Graph computations are equally susceptible to softfautls as numerical computation. Researchers have started looking at resilient discrete computation only recently [cite]

IV. Self-stabilzing Algorithms

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[dijkstra]. An algorithm can also be viewed as a system with states and transitions. Its state is a subset of the intermediate variable which enables continued execution. 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[], we have shown that such abstraction may help us to construct resilient algorithms.

a) 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

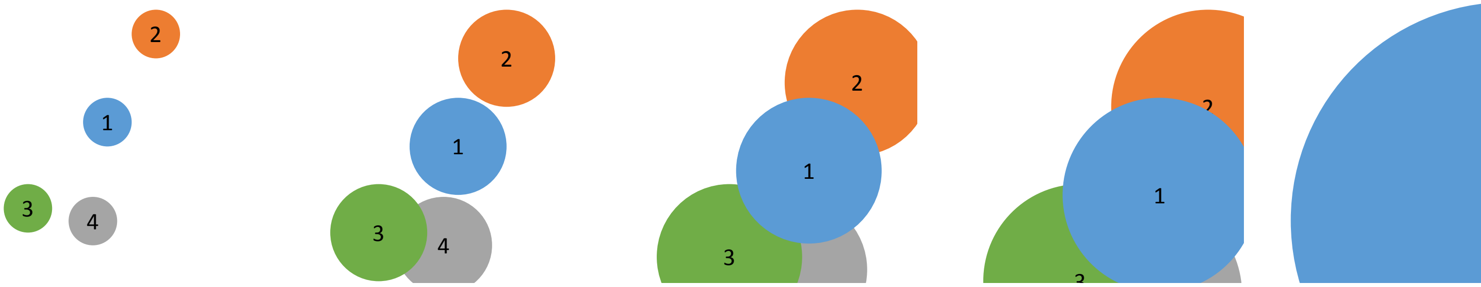


Figure 1: These sub-figures conceptually show how the connected component id propagates through the graph as time evolves. Subfigures represent snapshots of the algorithm at different times. For simplicity, this example assumes that all the shown vertices are connected. Initially the number of connected 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.

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.

b) 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.

V. Self-correcting Label Propagation Algorithm

In this section, we will describe the design of the self-stabilizing label propagation algorithm. We start with decribing the state of the algorithm and a condition for a valid state. However, the condition for valid state is not easy to verify computationally. To overcome this issue, in we augmented the data structure with redundancy to allow state verification. We use the augmented state as previously, and present new conditions for valid states. And finally we give efficient algorithm to verify these algorithms in cost effective manner.

State of LP algorithm: In algo[REF], the current connected component label array CC describes the state of LP algorithm. We use $CC^i[v]$ to denote the current label for vertex v in i-th iteration. We denote the final label of vertex v in fault-free execution as $CC^\infty[v]$.

Effect of fault in LP state:

Valid states:

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

A. Self-correcting Label Propagation Algorithm

We described self-correcting label propagation algorithm for synchronoous label propagation algorithm. Self-correcting algorithm corrects the current state using previous valid state.

To do so, we show that if CC^{i-1} is valid state then CC^i is valid if

$$CC^i[v] = CC^{i-1}[u] \ where \ u \in \mathcal{N}(v) \ \forall v \tag{2}$$

However, verifying eq[x] still requires $O(V+E)$ work, thus inefficient. However, self-correcting algorithm uses an auxilliary data structure called parent array. Parent of vertex v, denoted by $P[v]$, refers to vertex that

B. Self-stabilzing Label Propagation Algorithm

The main difference between self-correcting label propagation algorithm and self-stabilizing label propagation algorithm is that self-stabilizing label propagation algorithm doesn't require a previous valid state S^{i-1} to bring the algorithm to a valid state. We can formally pose this question as follows.

Problem Statement:: Given a graph $G = (V,E)$ and an arbitrary state for label propagation $S = (CC,P)$, determine if S is a valid state. If S is invalid, then construct a state S^*such that $S^* \approx S$, and S^* is a valid state.

1) Properties of LP state in fault free execution: We describe two key properties of a state S in fault free execution and later we prove that for any state that satisfy these properties is a valid state.

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

Self-correcting label propagation algorithm uses property 1 and 2, and additionally assumes that CC^{i-1} is a valid state.

Property 3 describes the range of valid values of $CC[v]$. In a synchronous label propagation algorithm, we have $CC^i[v] = CC^{i-1}[P(v)]$, and since label decreases over the iteration, thus latest $CC^i[P(v)] \le CC^{i-1}[P(v)]$. Therefore, $CC^i[P(v)] \le CC^i[v]$. While we do not show here in detail, this relation holds good in fault free execution of asynchronous case also.

Property 4 describes the structure of parent array. Make a gifure and explain.

2) Sufficient condtions for a valid LP state : We show that the former four conditions are sufficient for a valid state.

Theorem 2. *A label propagation state $S = \{CC, P\}$ is a valid state if for all vertices v, we have following conditions met:*

1) $CC[v] \le v$;
2) $P(v) \in \mathcal{N}(v)$;
3) $CC[P(v)] \le CC[v]$;
4) If $P(v) = v$, then $CC[v] = v$; 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.

Proof: If $H = (V, E_H)$ is a forest then, for any vertex v, the sequence $\{P(v), P^2(v), \ldots\}$ convereges to the root of the tree in the forest H. Let's denote the convergent of the sequence as $P^\infty(v)$.

Let $u = P^\infty(v)$. Since $P(u) = P(P^\infty(v)) = P^\infty(v) = u$, therefore from condition 4:

$$CC[u] = u \tag{3}$$

If condition-2 holds for all the vertices $\{v, P(v), P^2(v), \ldots\}$, then v and $P^\infty(v) = u$ are in same connected component. Therefore,

$$CC^\infty[v] = CC^\infty[u]. \tag{4}$$

Since label of u,will monotonically decrease in susequent iteration, therefore

$$u \ge CC^\infty[u] \tag{5}$$

From condition-3 $CC[v] \ge CC[P(v)] \ge CC[P^2(v)]...$

$$CC[v] \ge CC[P^\infty(v)] = CC[u] = u \tag{6}$$

Combining eq(x) and eq(y) and eq(z), we get $CC[v] \ge u \ge CC^\infty[u] = CC^\infty[v]$

$$CC[v] \ge CC^\infty[v] \tag{7}$$

Combining eq(x) with condition 1, we get

$$CC^\infty[v] \le CC[v] \le v$$

Eq(x) holds for all the vertices $v \in V$, therefore by Thm(1), $S = \{CC, P\}$ is a valid state. ■

Note that Thm(2) describes a set of sufficient condition. In general, there can be a state from which algorithm will converge to correct solution yet not satisfy Thm(2), leading to false positives. However, false-positives will only cause a small amount of additional computation, which is not a big concern in this case.

C. Verifying state validity

All the conditions of Thm(2) except the last one, can be verified locally for any vertex in $\mathcal{O}(1)$ operations. To verify $H = (V, E_H)$ is a forest, is equivalent to verifying that H does not have any loops, which is not a local operation.

1) Loop detection in H: In sequential case, we can use Tarzen's strongly connected component algorithm to find the loops in the graph H. Tarzen's strongly connected component algorithm requires $\mathcal{O}(V + E_H)$ operations. But since, $|E_H| = |V|$, therefore detecting loop in H requires $\mathcal{O}(V)$ operations in total, which is same as cost of other operations in sequential case.

In parallel case, Tarzen's strongly connected component algorithm (or any other algorithm based on BFS or DFS) are not suitable as: (a) they are inhrently sequential and have a depth of $\mathcal{O}(V)$; and (b) they can not be efficiently expressed in vertex centric programming model as label propagation algorithm. Therefore, we need a new loop detection algorithm which is parallel and can be expressed in vertex centric programming model.

```
Algorithm 1
Require: $\rho \ge 1$
Ensure: $X_k$
$NumChanges \leftarrow 1$
 $R(v) \leftarrow P(v)$
while $NumChanges > 0$ do
    for all vertex $v \in V$, Do in Parallel  do
        $r = \min(R(v), R(P(v)))$
        if $r \ne R(v)$ then
            $R(v) = v$
            $NumChanges = NumChanges + 1$
        if $v = R(v)$ then
            Report Loop
```

2) Parallel Loop detection in H: Our parallel loop detection is based on the idea that if v is a part of the loop in H, then v will re-appear in the sequence $\{P(v), P^2(v), \ldots\}$. Let's call the minimum element in the sequence $\mathscr{A}(v)$:

$$\mathscr{A}(v) = \min\{P(v), P^2(v), \ldots\}$$

If there is no loop in the, then v will not appear in the sequence $\{P(v), P^2(v), \ldots\}$, thus v can not be equal to $\mathscr{A}(v)$. Therefore, $v = \mathscr{A}(v)$ is only possible if there is a loop in H which consist of vertices $\{v, P(v), P^2(v), \ldots\}$. Thus to detect any loop in H, we calculate $\mathscr{A}(v)$ for all the vertices and compare it with v.

We calculate $\mathscr{A}(v)$ in parallel using pointer jumping technique. The Algm[x] describes the pseudocode for parallel loop detection in H. ADD more text here

Cost of loop detection: If there is no loop in H, then Algm[x] requires $\lceil log_2(h)\rceil+1$ iterations to converge, where h is the maximum height among all the trees in the forest H. If there are cycles in H, and if length of maximum cycle is $c$then, Algn[x] takes $\lceil log_2(c)\rceil+2$ iterations to detect it. So it need a total of $\max(\lceil log_2(h)\rceil+1, \lceil log_2(c)\rceil+2)$ iterations. In each iteration, we perform $\mathcal{O}(V)$ work. Thus total cost of loop detection is $\mathcal{O}(Vlog(V))$ as $c,\ h \leq |V|$.

D. Correction step

Correction step has two parts: detection of invalid component of states; and bring them to a correct state. Using Algm[x] and Thm[y], we design the correction step as follows. First we ensure that graph H is a proper subgraph of the input graph G, by verifying $P(v) \in \mathcal{N}(v)$ for all vertex v. If for some vertex $P(v) \notin \mathcal{N}(v)$, then we reset vertex v by setting $CC[v] = v$ and $P(v) = v$. After this we check, if a vertex v is a root of any tree, then $CC[v] = v$. It should be noted that in both check, if we find that state of v is invalid, it also means that state of all the descendent of v is also invalid. However, all the otehr vertices which have invalid state have not been informed yet. Once these checks and local correction are performed, we use a modified algorithm to check for loops and correct labels.

```
Algorithm 2
Require: $\rho \geq 1$
Ensure: $X_k$
$NumChanges \leftarrow 1$
$R(v) \leftarrow P(v)$
while $NumChanges > 0$ do
    for all vertex $v \in V$, Do in Parallel do
        $r = \min(R(v), R(P(v)))$
        if $r \neq R(v)$ then
            $R(v) = v$
            $NumChanges = NumChanges + 1$
        if $v = R(v)$ then
            Break the loop at $v$
            $P(v) = v$
Correction
$CC[v] = R(v)$
```

The Algm[z] works by when a vertex v finds loop $v = \mathcal{A}(v)$, then v has the minimum vertex id in the sequence $\{v, P(v), P^2(v), \ldots\}$. Thus, v must be root of the tree. Therefore, we reset the vertex v, by setting its parent to itself $P(v) = v$. Also when the loop detection converges, $\mathcal{A}(v)$ has the minimum vertex label among all its ancestors. Thus, the correct value of $CC[v]$ should be $\mathcal{A}(v)$. So at the end of loop detection step all vertex v will have aquired a valid value.

E. Fault-tolerant label propagation algorithm

Since the self-stabilization step has cost $Vlog(V)$, it is prohibitively expensive to be performed in every iteration.

Table II: 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

To ensure the self-stabilization property, previously[cite] we performed the correction step at regular interval. However, in case of label propagation algorithm, number of iteration is $\mathcal{O}(log(d))$, where d is the diameter of the graph. Thus, for many graph label propagation algorithm conerges in 5 to 10 iterations.

Due to these circumstanes, we only perform correction step when the label propgation algorithm reports convergence. If the correction step finds that the algorithm is in valid state, and algorithm has converged than algorithm has converged. If the correction step reports that algorithm is in an invalid state, then it tries to bring the algorithm back to valid state and restarts the label propagation iteration. However, this state is usually very close to the final state and it only takes a small fraction of iteration to converge[ref to result].

VI. 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].

A. Experimental set-up

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.

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.

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

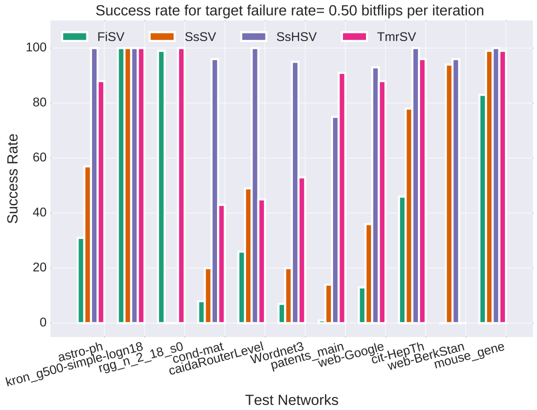


Figure 2: Success rate for different different test networks

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.

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.

B. 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

C. Convergence Test

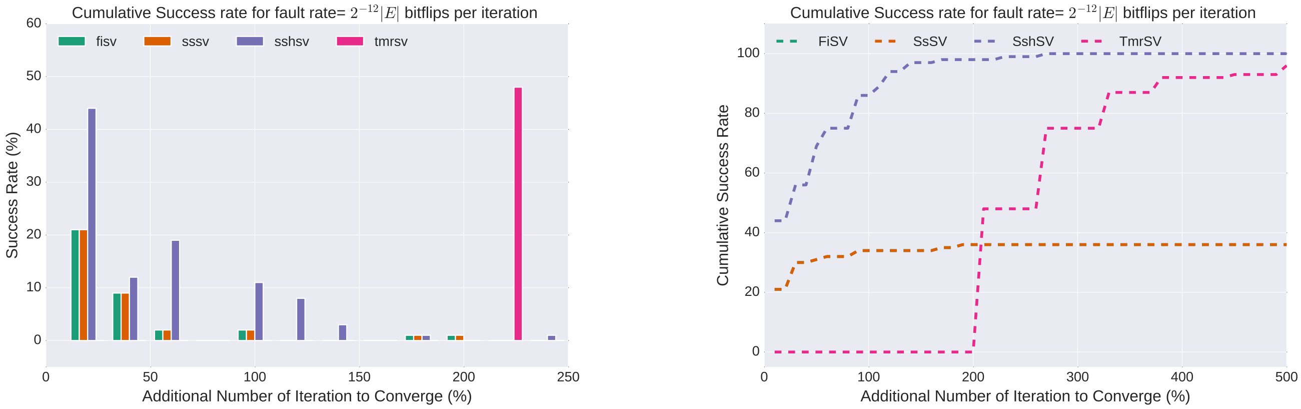
D. Overhead of Correction step

VII. Related work

VIII. Conclusion and Future Work

***c**onlusion*

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



(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

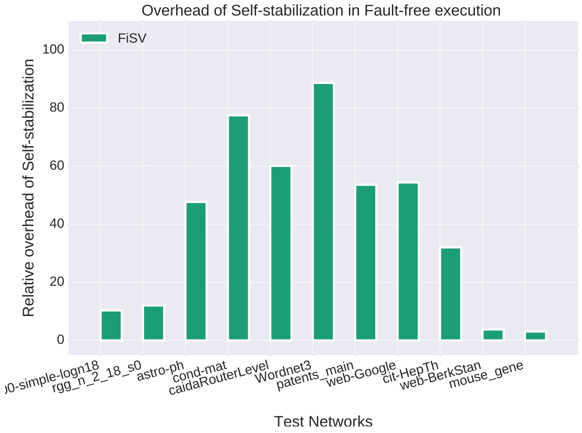


Figure 4: Overhead of correction step for different networks