A Fault Tolerant Connected Component Algorithm

Piyush Sao, Oded Green, Chirag Jain, Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology
{piyush3,ogreen3,cjain3,richie}@gatech.edu

ABSTRACT

This paper presents a fault tolerant label propagation based graph connect component algorithm, that can converge to correct solution even in presence soft transient faults.

First, we develop a set of conditions on state variables of the algorithm that ensures convergence to correct solution. We augment the algorithm with additional state variable to make verification of these conditions becomes computationally tractable. Finally, we add a local correction scheme which ensures the algorithm comes to a valid state after a faulty execution.

We present experimental results on wide range of matrices to show that the new algorithm can withstand high fault rates, with minimal storage and computational overhead. Additionally, we present analytical models for overhead of fault detection and correction.

Keywords

Fault-tolerance, Graph-algorithm, Connected Component, Bit-flip

1. INTRODUCTION

2. FAULT MODEL

3. SV ALGORITHM FOR GRAPH CONNECTED COMPONENT

We use Siloach and Vishkin algorithm (cite) for calculating connected component of the graph, as the baseline algorithm and modify the algorithm to make it fault tolerant. The! SValgorithm is based on so-called "label-propagation" techniques. It is embarrassingly parallel, thus it can be efficiently implemented on highly concurrent shared memory (cite) and distributed memory architectures.

The SValgorithm is described in alg. 1. By convention, connected component id of any vertex is minimum vertex-id in its connected component. For all the vertices in the graph, SValgorithm maintains the current connected-component id array CC, which will contain final connected component id at the end of computation.

DOI: 10.1145/1235

Algorithm 1 Label propagation algorithm

```
Require: $G = (V, E)$
 1: Initialization ;
 2: for each $v \in V$ do
 3:     $CC[v] = v; CCp[v] = v;$
 4: end for
 5: $NumChange = |V|$
 6: while $NumChange > 0$ do;
 7:     $NumChange \leftarrow 0$
 8:     MemCpy($CCp$,$CC$,$|V|$)
 9:     for each $v \in V$ do
10:         for each $u \in E(v)$ do
11:             if $CCp[u] < CC[v]$ then
12:                 $CC[v] \leftarrow CCp[u]$
13:                 NumChanges = NumChanges+1;
14:             end if
15:         end for
16:     end for
17: end while
```

In addition, to felicitate concurrent execution, a connected component id from previous iteration is also stored in array CCp. In the beginning of computation, CCp array for each vertex is initialized to its vertex id. In each SViteration, each vertex traverses its adjacency list, and computes the minimum component-id of all its neighbors and stores it in the CC array. Thus, eventually minimum component id propagates to all the vertices in the given connected component. We keep track of number of changes in CC array occur in any iteration. The SValgorithm terminates when no vertex changes its component-id in an SViteration.

Each iteration requires $\mathcal{O}(|V| + |E|)$ computations, since the algorithm accesses all vertices and their respective adjacencies. The maximal length of propagation is limited by the graph diameter d. As such, the total time complexity of the algorithm is $\mathcal{O}(d.(V + E))$. Relative to alg. 1, there is a shortcut that can reduce the number of iterations to $d/2$. However, this does not change the asymptotic time complexity of the algorithm, and we do not consider it further

Conceptually, the component labels propagate as fig. **??** depicts. Initially (a), four of the components have minimal labels locally; these labels propagate gradually, and the label of a given node may change several times, (b)-(e), possibly even within the same iteration. Eventually, the algorithm reaches a final state (e) where for a fully-connected graph there will be a single connected component.

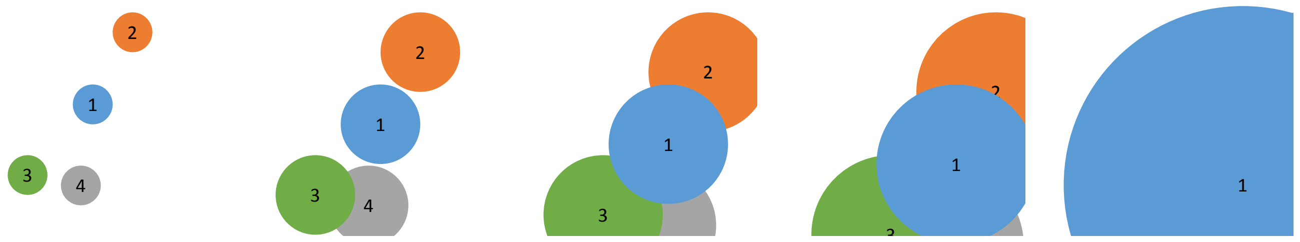

Figure 1: These sub-figures conceptually show how the connected component id propagates through the graph as time evolves - each subfigure is for a different iteration of the algorithm. This example assumes that all vertices are connected and for simplicity shows only the connected components are 1 through 4. 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.

Algorithm 2 Fault Tolerant Label propagation algorithm

```
Require: $G = (V, E)$
 1: Initialization ;
 2: for each $v \in V$ do
 3:     $CC[v] = v; CCp[v] = v, H[v] = v;$
 4: end for
 5: $NumChanges = |V|$
 6: while $NumChanges > NumCorrections$ do;
 7:     $NumChanges \leftarrow 0$
 8:     MemCpy($CCp$,$CC$,$|V|$)
 9:     for each $v \in V$ do
10:         for each $u \in E(v)$ do
11:             if $CCp[u] < CC[v]$ then
12:                 $CC[v] \leftarrow CCp[u]$
13:                 $H[v] \leftarrow u$
14:                 $NumChanges = NumChanges + 1;$
15:             end if
16:         end for
17:     end for
18:     Fault detection and correction
19:     $NumCorrections = 0$
20:     for each $v \in V$ do
21:         if $CC[v] \neq CCp[H[v]] | CC[v] > v | H[v] \notin E(v)$ then
22:             $NumCorrections = NumCorrections + 1;$
23:             for each $u \in E(v)$ do
24:                 if $CCp[u] < CC[v]$ then
25:                     $CC[v] \leftarrow CCp[u]$
26:                     $H[v] \leftarrow u$
27:                 end if
28:             end for
29:         end if
30:     end for
31: end while
```

4. FAULT TOLERANT SV ALGORITHM

As a motivation for proposed algorithm, we analyze effects of silent data corruption in the SV algorithm on convergence of the algorithm.

4.1 Fault Propagation

In presence of faults, the CC vector may get corrupted. Corruption of CC vector may result in slower convergence, converging to incorrect results, or converging to a result which may be considered technically-correct, however it may not be correct from convention of output. For instance, in presence of fault final connected component id is second minimum, as oppose to minimum according to our convention. We consider such cases as incorrect results, as consumer application of the algorithm may use properties of defined convention.

To analyze propagation of data corruption in CC vector, consider any vertex v, and it's CC value in the i-th iteration as $CC^i[v]$. Let $CC^\infty[v]$ be the final CC value of v in case of fault free execution. Suppose a fault occurs and due to the fault $CC^i[v]$ is changed to $\hat{CC}^i[v]$. Now depending on value of $\hat{CC}^i[v]$, the data corruption may propagate, or will be corrected in a later iteration. We consider following two cases:

Case 1: $\hat{CC}^i[v] > CC^\infty[v]$.

In this case, since minimum CC in the connected component remains unchanged, $\hat{CC}^i[v]$ will be eventually overwritten by $CC^\infty[v]$. However, it may still delay convergence and if v is minimum vertex id in the connected component, i.e. $CC[v] = CC^\infty[v]$, connected component will converge to second minimum vertex-id, thus giving incorrect result.

Case 2: $\hat{CC}^i[v] < CC^\infty[v]$.

In this case, minimum CC in the connected component is changed and, thus $\hat{CC}^i[v]$ will propagate to all other vertex, resulting in converging to incorrect results.

THe difficulty is, looking at CC array it is not possible to determine if: a) a fault has occurred that needs corrections; b) algorithm will converge to correct answer. To felicitate, determining if the algorithm is in valid state, we introduce another vector that we call *parent* vector denoted by H. For each vertex, the H vector stores the vertex that caused the last change in its CC value. For instance, if in a given iteration, $CC[v] > CCp[u]$, then SValgorithm updates $CC[v] \leftarrow CCp[u]$, and we correspondingly we update $H[v] \leftarrow u$. We present following theorem that establishes validity of a state vector.

Theorem 1. *Given a graph $G = (V, E)$ and a arbitrary state vector $S = (CC, H)$, the alg. 1 starting from S will converge to correct solution defined by [insert ref to definition], if S satisfies following condition:*

1. *Parent of any vertex v is either v itself or it is one of neighbor: $H(v) \in \{v, E(v)\}$;*
2. *if $H(v) = v$, then $CC(v) = v$;*
3. *$CC[v] \leq v$;*
4. *$CC[v] \geq CC[H[v]]$;and*
5. *There are no loop except self-loops in the directed graph G^* defined by $G^* = (V, E^*)$, where $E^* = \{(v, H(v)) \forall v \in V\}$.*

4.2 Fault tolerant SVAlgorithm

In principle thm. 1 can be used to construct a fault detection and correction scheme. Im theory, such a fault detection and correction scheme may not require detection and correction step to be executed in every iteration, and its frequency can be a tuning parameter. However, there are a few challenges in realizing such a detection and correction step. First, while condition 1,2, and 3 can be verified using $\mathcal{O}(1)$ computation for each vertex , verifying condition 4 would require $\mathcal{O}(d)$ computation for each vertex [**?**]. Moreover, correcting the state variables would be even more costly.

This problem can be overcome if we assume that $S^i = (CC^i, H^i)$ vector from previous iteration was correct. If S^i satisfies conditions from thm. 1, and a faulty SViteration is executed to compute $s^{i+1} = (CC^{i+1}, H^{i+1})$:

$$(CC^{i+1}, H^{i+1}) \leftarrow fSV(G, CC^i)$$

then, there will not be any loop as long as S^{i+1} satisfies condition 1, 2 and 3 of thm. 1. Thus, fault detection and correction becomes computationally more tractable.

To detect a fault, after each faulty SViteration, we check condition 1, 2 and 3 of thm. 1. Checking for condition 2, 3, and 4 require $mathcalO(1)$ computations.

To check for any vertex v whether $H(v) \in \{v, E(v)\}$, in a naive algorithm, we may traverse through adjacency list of vertex v, and check if $H(v)$ is indeed a neighbor of v. However, doing so for each vertex will require $\mathcal{O}(|V| + |E|)$ computations. It may be speeded-up by using binary search, which may reduce complexity to $\mathcal{O}(|V| log(|V|))$, which may be still high. We overcome this problem by storing relative address of $H(v)$ in the adjacency list instead of storing absolute vertex-id of $H(v)$. In other words, if $H(v)$ is *ind*-th entry in the adjacency list of v, .i.e $H[v] = E[v][ind]$, then we store *ind*. This reduces the checking if $H(v) \in v, E(v)$ to checking if $ind < |E(v)|$. Thus, checking first condition becomes $\mathcal{O}(1)$ operations. Therefore total overhead of fault detection is $\mathcal{O}(|V|)$.

If a fault is detected for any vertex v, we correct it by calculating $CC[v]$ again. If we assume fault rate is f, and average degree of any vertex is $\mathcal{O}(|E|/|V|)$, then cost of correction will be $\mathcal{O}(f|E|/|V|)$.

Since, weak condition require that CC from previous iteration should be correct, for algorithm to converge to correct value, in this work, we assume that fault detection and corrections steps are done in a reliable mode. Algorithm 2 shows the complete algorithm. We list overhead of alg. 2 in

Table 1: Overhead of FT-SV

	Asymptotic Overhead
Memory	$\mathcal{O}(\lvert V\rvert)$
Fault Detection	$\mathcal{O}(\lvert V\rvert)$
Fault Correction	$\mathcal{O}(f\lvert E\rvert/\lvert V\rvert)$

Table 3: Testbeds used for performance evaluation.

Prop	SNB20c
Sockets×Cores	2×8
Clock Rate	2.4GHz
DRAM capacity	128GB
DRAM Bandwidth	72GB/s

5. RESULTS

We performed a series of experiments to test robustness and overhead of our algorithm described in § 3.

5.1 Fault Injection Methodology

Since memory accesses are most performance critical part in SVcomputation, we inject faults in memory access pattern. There are two main memory accesses in SViteration: traversing adjacency list for each vertex; and accessing CC array for each vertex in adjacency list.

In the first case, before each SVsweep, we randomly select $f|E|$ edges, where f is fault-rate. Let $e = (v, u)$ be one of the selected edges. When we encounter e while traversing adjacency list of v, we flip one of the bits of u—the vertex which e is pointing—randomly. Therefore, due to the fault, v will visit flipped vertex $\hat{u}$, instead of accessing u.

In the second case, we will again choose another set of $f|E|$ edges. Let $e = (v, u)$ be one of the selected edges. When we encounter e while traversing adjacency list of v, we flip one of the bits of $CC[u]$. Therefore, due to the fault, v will visit the correct vertex, however, it see incorrect value of $CC[u]$. It should be noted that $CC[u]$ will be accessed multiple times in a SViteration, and we assume that all accesses to $CC[u]$ in an iteration are independent. In other words, if $CC[u]$ is accesses while visiting vertex v_1, v_2, and if $CC[u]$ is corrupted while visiting v_1, then $CC[u]$ may or may not be corrupted when it is accessed while visiting v_2.

5.2 Experimental Setup

Test Graphs.

The graphs used in our tests are listed in table **??**. These graphs are taken from the 10th Dimacs Implementation Challenge [**?**], come from various real and synthetic applications.

Testbed.

We prototyped baseline and fault tolerant implementation using *C* language. We used the Intel C Compiler (ICC 15.0.0), with highest level of optimization $-O3$ to compile our benchmarks. We ran all our experiments on SNB16c, key properties of the systems are listed in table 3

5.3 Convergence in presence of faults

First, we try to understand the convergence property of SValgorithm in presence of faults. Recall that in the worst case, an *acceptable* fault that is not detected by alg. 2, may delay the convergence by one iteration. On the other hand, one additional iteration means additional faults, which may necessitate an additional iteration for convergence. Thus, it is expected that with increasing fault rate, convergence might slow down. To understand practicality of our algorithm, it becomes essential to understand such slow down in convergence.

In fig. 3, we show the convergence of SV algorithm in

Table 2: List of matrices used for experimentation

Matrix Name	Source	#Vertices	#Edges
astro-ph	collaboration network	16706	242502
audikw1	UF Sparse Matrix Collection	943695	77651847
caidaRouterLevel	Clustering	192244	1218132
cnr-2000		325557	2738969
coAuthorsDBLP		299067	977676
coPapersDBLP		540486	15245729
cond-mat-2005		40421	175691
delaunay_n18		262144	786396
er-fact1.5-scale20		1048576	10904496
G_n_pin_pout		100000	501198
kron_g500-simple-logn18		262144	10582686
ldoor		952203	22785136
preferentialAttachment		100000	499985
rgg_n_2_18_s0		262144	1547283

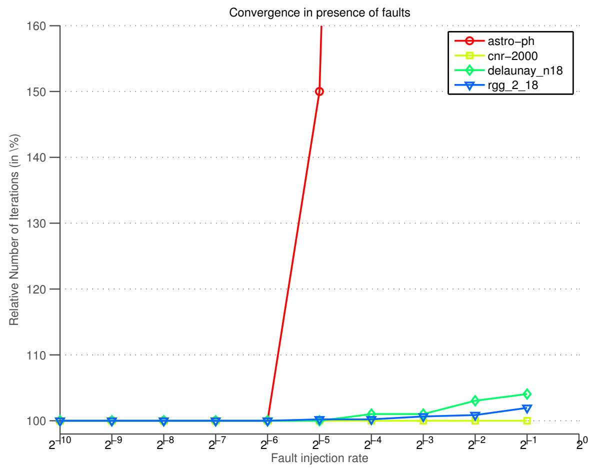

Figure 2: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

presence of faults for four graphs. We vary the fault rate from $2^-10 \times |E|$ bit flips in every SV iteration to $2^-1 \times |E|$ bit flips. Note that this fault injection rate is extremely high and, we do so to stress test the proposed algorithm.

In fig. 3, we observe that for all practical fault rates($< 2^{-6}|E|$) , algorithm converges without any additional iteration. Additionally, except *astro-ph*, all other graphs converge to correct solution within 5% additional iteration at the highest fault rate. As such, we conclude that our proposed algorithm can withstand high fault rates, with minimal additional iteration overhead.

The difference in behavior between graph *astro-ph*, and other graphs is due to two reasons. First, the graph *astro-ph* has the smallest diameter all four test cases, and it only takes 10 iterations in fault free case to converge. If a fault that might cause an additional iteration occurs, it provides fewer opportunity to be corrected in later iterations. Secondly, ...

5.4 Overhead of fault detection and correction

Zero-Overhead.

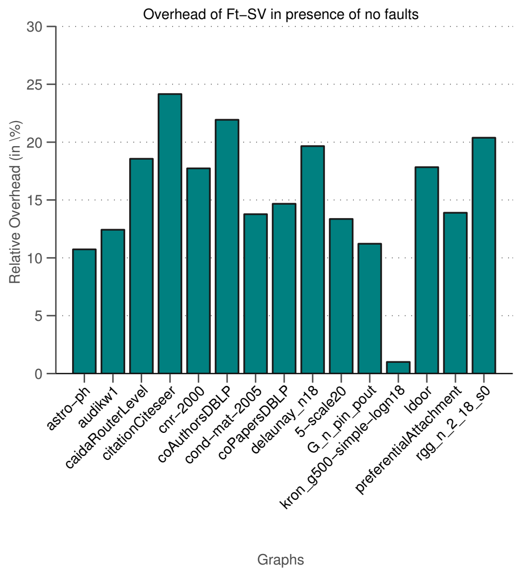

Figure 3: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

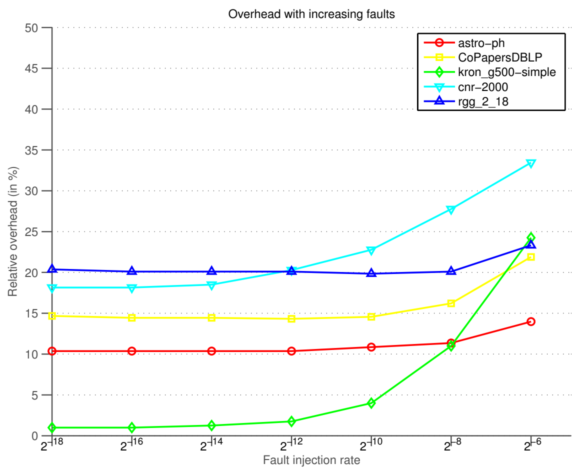

Figure 4: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

Overhead with increasing Faults.

6. RELATED WORK

7. CONCLUSION AND FUTURE WORK

8. ACKNOWLEDGMENT