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

The exceeding growth of big datasets has created a need for using distributed systems and accelerators for real-time analytics, including for graph and social network analytics. Social networks such Facebook and Twitter are constantly growing at time of writing have over members with relationships between these members. The decrease of transistor sizes and the improved power consumption, as part of Moore's laws, has brought a new challenge with it - an increased number of faults due to random bit flipping. This problem is very likely to become even more challenging as transistor sizes continue to decrease and the power envelope restrictions become even tighter. As larger systems, with a higher thread count, become ubiquitous so will the problem of power related faults.

In this work, we tackle the problem of faults in a connected component algorithm that is propagation based. We modify the parallel algorithm of Shiloach-Vishkin [6] and make it fault-tolerant by augmenting the data structures used by the algorithm and by adding a detection and correction scheme at the end of every iteration. By detecting and correcting the error at the end of each iteration we are able to limit the propagation of the error to the remainder of the graph in the following iterations. This can prove to be especially important for distributed implementations that are communication bound.

In this paper we show that the overhead of our fault tolerant algorithm is not significant. Specifically, when comparing the fault tolerant algorithm with the its fault-free counter part for a fault-free execution (i.e. there are no faults), the use overhead of our algorithm is on average 15%. This overhead includes updating the new auxiliary data structure and running the detection scheme [1]. When faults do occur, the execution time of our fault tolerant algorithm is near constant for small to moderate fault rates. For extremely high fault rates, where there is one error in each 64-1024 memory operations, we see the overhead increasing by anywhere $2X - 4X$ that the no-fault overhead. It is well worth noting that systems working at such a high fault rate would be considered highly unreliable. Lastly, we show that so long as the fault rate is reasonable (i.e. the system is reliable), then our algorithm typically does not increase significantly the number of iterations needed to converge to the correct answer. Extremely high error rates can prevent propagation of a label in a specific iteration, thus adding an iteration to the end.

2. FAULT MODEL

In this paper, a fault includes any instance in which the underlying hardware deviates from its expected behavior. We will use the taxonomy of faults outlined by Hoemmen and Heroux [?], among others, as summarized below.

We distinguish *hard faults* and *soft faults*. A hard fault interrupts the program, causing it to immediately crash or terminate. This occurs when, for instance, a node or network link fails. By contrast, a soft fault does not cause immediate interruption of the program. L1 cache bit flips are a common type of soft fault.

A particularly insidious manifestation of soft faults is *silent data corruption* (SDC), where an soft fault corrupts the intermediate variables of the algorithm without notifying the application. Such data corruption may propagates and eventually result in incorrect output masquerading as correct.

[1]The correction scheme is not executed as no faults are detected.

© 2016 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:     $CC[v] = v; CCp[v] = v;$
 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
```

Recently, a number of techniques have been developed for dealing with soft faults in numerical linear algebra. There is a large and growing literature on so called *algorithm based fault tolerance* (ABFT), that primarily rely on testing checksum invariants [**? ? ? ?**]. For iterative numerical algorithms, techniques have been developed where algorithm can recover from fault by itself, thus obviating need for fault detection and correction[**? ? ?**].

Insert Citations In contrast to numerical linear algebra— which is characterized by large data parallel floating point computation, the graph computation is characterized by irregular and indirect memory accesses. While for slow storage such as DRAM and disk, error correcting codes are deployed and efficient, fast memory such as cache and registers still remains unprotected[cite, cite]. Therefore, large scale graph computation is as susceptible to soft faults as any other computation.

As far as we know, the issue of fault tolerance in graph computation is, so far, addressed only in context of hard fault and they are based on check-pointing and restarting method[cite cite]. Checkpoint and restart based method are usually not scalable with high fault rates, and are still prone to silent data corruption.

Redo the summary

This paper concerns only transient soft faults. Thus, in the consequent, a "fault" is a transient soft fault unless otherwise noted. When faults cause the output to fall outside acceptable limits, we say the algorithm has failed. For an algorithm to terminate successfully, at least some amount of computation must be done reliably. However, in general we do not have control over which operations are done correctly and which operations are done incorrectly. In this paper, we will attempt to distinguish algorithmic operations that must be performed reliably from those that may be performed unreliably. We refer to these different modes of computation as reliable mode (or reliable computation) from unreliable mode (or computation), without saying precisely how to implement these modes. The prior work of others similarly assumes "selective reliability" [**?**].

3. CONNECTED COMPONENTS

Connected components are widely used in graph analytics as these represents subgraphs in the entire graph that all vertices are connected. While connected components can not imply the level of significance of a vertex, having connected components can be useful for community detection, centrality analytics, and for streaming graph analytics. Several recent studies show that many real-world social networks may have one connected component that includes rougly 90% of the vertices [5, 8]. While many vertices are part of the large connected component, they are not tighlty bound to this component and are typically on the "outskirts" of this component.

To deal with the increased sizes of graphs numerous parallel algorithms have been designed. One of the mostly widely used algorithms is that of Shiloach-Vishkin [6]. Variations of Shiloach-Viskin can be found in LIGRA [7] for the CPU, Gunrock for the GPU [9], and a scalable distributed implementation [3]. While this algorithm offers a parallel formulation for connected components, it can also be implemented sequentially or used with a single thread as was done in [4] for figuring out the cost of hardware branch prediction accuracy for graph algorithms.

Shiloach-Vishkin (SV) Algorithm for Connected components

In our current work we augment the SValgorithm to support fault tolerant execution. The SValgorithm is a label-propagation algorithm where a label of vertex inside the connected component is propagated to the remaining vertice of that component. Many formulations use the label of the vertex with either the minimal or maximal id within a component as the value that is propagate. We to use this formulation and propagate the smallest id within the conected component. Pseudo code for this algorithm can be found in Algorithm 1. The symbols used in this algorithm and the fault tolerant algorithm introduced in the next section can be found in Table 1.

In the initialization phase, each vertex is placed in its own connected component. From this point on, in each iteration of the algorithm vertices inspect the connected component value of their neighbors, CC. If one of the neighbors has a smaller id then the vertex will switch its connected component to the one that its neighbor has chosen. Figure 1 depicts how the label propagates from its initial state, Figure 1 (a), to its final state Figure 1 (e), where the label of the vertex with the smallest id takes over all the vertices in the component. The algorithm terminates when an iteration as completed and no vertex has swapped its CClabel. One of the key findings of [4] is that the label swapping occurs at a higher rate in the initial iterations. In the advanced iteration, the only vertices that have yet to swap their labels are those that are far away from the vertex with the smallest id.

The time complexity of the SV algorithm is as follows. Each iteration requires $O(|V| + |E|)$ computations for accessing the connected component array for all vertices and their adadjacencies. The number of iterations of the algorithm is bound by $O(log(|V|))$ iterations [6] using the shortcutting stage of the algorithm. For many real world networks, especially small world networks, the number of iterations is bounded by the graph diameter d which is the upperbound on how long it would take for a value to propagate from one end of the graph to the other. As such, the total time complexity of the algorithm can be written as $O(log(|V|) \cdot (|V| + |E|))$ or $O(d \cdot (|V| + |E|))$. We chose the later of these for our analysis.

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
```

Observartions

1) The pseudo code in Algorithm 1 uses CC^i to store the connected components after iteration i. While it may seem that these arrays are stored for all the iterations, in practice it is only necessary to store these for two consecutive iterations as the previous ones are not necessary.

2) Upon completion of the algorithm, each vertex is assigned to exactly one connected component and this can be found in CC^∞. Given that our formulation of SVuses the minimal id formulation, we assume that each vertex points to the vertex with the smallest id in the connected component. We refer to this as the **final and correct state**.

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.

4.2 Fault tolerant SVAlgorithm

In principle thm. **??** 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 [1]. 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. **??**, 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. **??**. Thus, fault detection and correction becomes computationally more tractable.

4.3 Fault Detection

To detect a fault, after each faulty SViteration, we check condition 1, 2 and 3 of thm. **??**. 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

Table 1: Symbols used in the fault free SValgorithm 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)

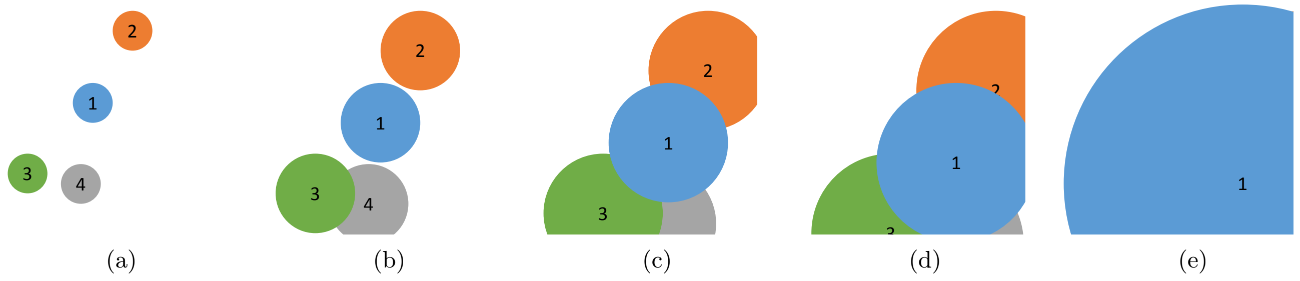

Figure 1: These sub-figures conceptually show how the connected component id propagates through the graph as time evolves. Subfigures represent snapshots of the algorithm at different times. For simplicity, this example assumes that all the shown vertices are connected. Initially the number of connected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

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 $\mathcal{O}(|V|)$.

4.4 Fault Recovery

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 $\mathcal{O}(f|E|/|V|)$.

4.5 Convergence Detection

Recall that, in a fault free execution of SValgorithm, algorithm terminates when there are no more label swaps in an iteration. In presence of faults, however, there can be label swaps due to faults even when algorithm has already converged. Thus, our previous convergence detection scheme will not be efficient in presence of faults.

To detect the convergence in presence of faults, we make use of following observation. In any iteration of alg. 2, number of label swaps in the main loop, is greater than or equal to number of corrections in correction phase. If the algorithm has converged to correct solution, then any following iteration, any label swap can happen only due to faults. In the following correction phase, some of those faults might get detected and corrected, thus number of corrections will be less than or equal to number of changes. On the other, if all incorrect values are detected and corrected, then algorithm has converged to correct solution.

4.6 Selective reliability

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 2: Overhead of FT-SV

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

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

Table 4: Testbeds used for performance evaluation.

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

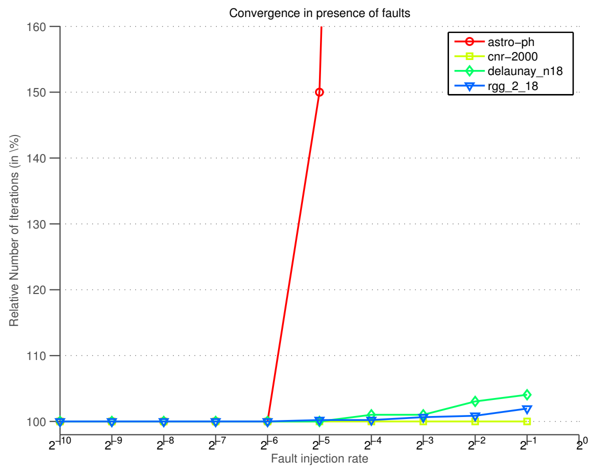

Figure 2: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

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 3. These graphs are taken from the 10th Dimacs Implementation Challenge [2], 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 4

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. 2, we show the convergence of SV algorithm in 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. 2, 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

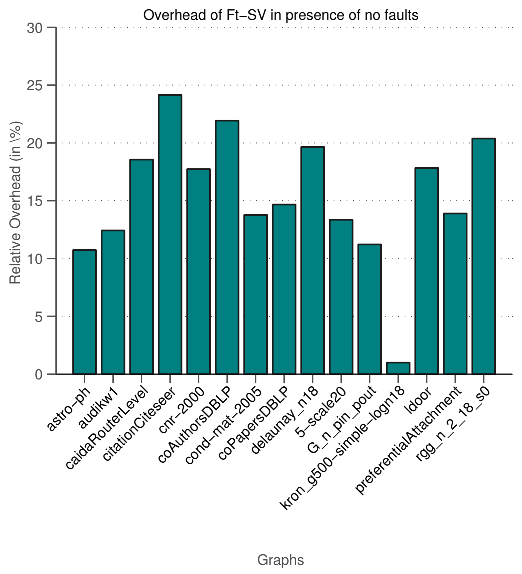

Figure 3: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

Overhead with increasing Faults.

In addition to extra iteration to converge, extra computation that goes in fault detection and correction can be also a source of significant overhead. We analyze overhead of fault detection and correction in two parts. First, we evaluate the overhead of algorithm FT-SVwhen no faults are injected. Then, we evaluate how overhead of proposed algo-

Table 3: List of matrices used for experimentation

Matrix Name	Source	#Vertices	#Edges	$\|E\|/\|V\|$
astro-ph	collaboration network	16706	242502	14.5
audikw1	UF Sparse Matrix Collection	943695	77651847	82.3
caidaRouterLevel	Internet router-level graph	192244	1218132	6.3
cnr-2000	Web crawl	325557	2738969	8.4
citationCiteseer	Citation network	268495	1156647	4.3
coAuthorsDBLP	Citation network	299067	977676	3.3
coPapersDBLP	Citation network	540486	15245729	28.2
cond-mat-2005	Condensed matter collaborations	40421	175691	4.32
delaunay_n18	Delaunay triangulations	262144	786396	3.0
er-fact1.5-scale20	Erdos-Renyi Graphs	1048576	10904496	10.4
G_n_pin_pout	Gnp random-graph	100000	501198	5.01
kron_g500-simple-logn18	synthetic graphs	262144	10582686	40.4
ldoor	Sparse Matrix	952203	22785136	24
preferentialAttachment	Clustering Instances	100000	499985	5
rgg_n_2_18_s0	random geometric graph	262144	1547283	5.9

rithm behaves at different fault rates.

Zero-Overhead.

Zero-overhead denotes the overhead of the algorithm in absence of any injected faults. In this case, all the additional computation goes into fault detection. We compare the zero-overhead of alg. 2, with fault free execution of alg. 1.

In fig. 3, we show the *relative* zero-overhead of alg. 2 different test graphs. In all the test cases, zero-overhead remains smaller than 25%. In the graphs with smaller $|E|/|V|$ ratio, for instance *citationCiteseer* ($|E|/|V| = 4.3$), *coAuthorsDBLP* (3.3), *rgg_n_2_18_s0* (5.9), have large zero-overhead. On the other hand graphs with large $|E|/|V|$ ratio such as *kron_g500-simple-logn18* ($|E|/|V| = 40$), we observe small zero-overhead. This variation of zero-overhead with $|E|/|V|$ ratio, us expected as asymptomatic complexity of fault detection in an FT-SViteration is $\mathcal{O}(V)$, while cost of an SViteration is $\mathcal{O}(|V| + |E|)$.

Overhead of FT-SV*in presence of faults.*

We evaluated overhead of FT-SVin presence of faults to understand scalability of the proposed algorithm with increasing fault rates. To do so, we vary fault rates from $2^{-}18|E|$ to $2^{-}6|E|$ distinct bit flips in every iteration for all test graphs. In fig. 4, we show the the relative overhead of FT-SValgorithm with respect to increasing fault rates for four matrices. We expect that overhead will increase linearly with increasing fault rates. For all the four matrices, overhead of fault detection is small for fault rates upto $2^{-10}|E|$ bit flips in every iteration. Beyond $2^{-10}|E|$, we see overhead increasing slowly, however increase in all the cases is sub-linear. This is due to many graphs considered in our study follows power law distribution of for degree of vertices, thus these graphs have very high number of vertices with very small degrees, thus correction for large number of vertices are $\mathcal{O}(1)$ operations. Among all the graphs tested, the graph *kron_g500-simple-logn18* showed steepest increase in

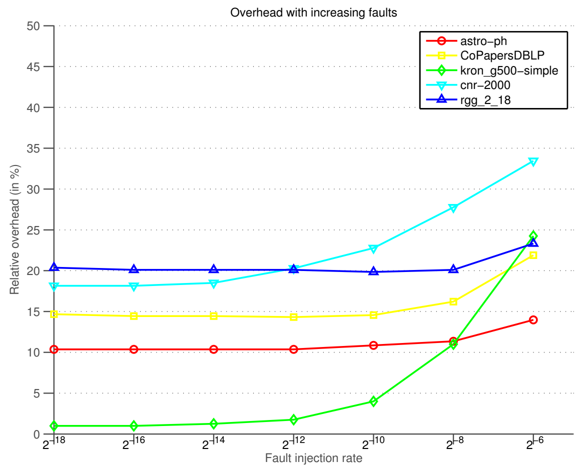

Figure 4: Comparison of model-driven work partitioning scheme to two static work partitioning scheme

overhead with increasing fault rates. The graph *kron_g500-simple-logn18* is a random Kronecker geometric graph and has very uniform distribution of degree among vertices, and thus with fault rate, correction cost shows linear increase. In all cases even in the highest fault rates considered in the paper, net overhead of FT-SVwere smaller than 35%. In contrast to that, naive redundency based fault tolerance algorithm will have overhead of more than 100%. Thus, we see that FT-SVcan tolerate efficiently withstand high fault rates.

6. RELATED WORK

7. CONCLUSION AND FUTURE WORK

8. ACKNOWLEDGMENT

References

[1] R. P. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

[2] H. M. P. S. C. S. David A. Bader, Andrea Kappes and D. Wagner. Benchmarking for graph clustering and partitioning. *Encyclopedia of Social Network Analysis and Mining*, pages 73–82.

[3] P. Flick, C. Jain, T. Pan, and S. Aluru. A parallel connectivity algorithm for de bruijn graphs in metagenomic applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 15. ACM, 2015.

[4] O. Green, M. Dukhan, and R. Vuduc. Branch-Avoiding Graph Algorithms. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 212–223, 2015.

[5] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[6] Y. Shiloach and U. Vishkin. An O(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[7] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 135–146, 2013.

[8] S. H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, 2001.

[9] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 50, pages 265–266. ACM, 2015.