A Self-correcting Graph Connected Component Algorithm

Piyush Sao, Oded Green, Chirag Jain, Richard Vuduc

The 6^{th} Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS) 2016

http://hpcgarage.org/ftxs16/

Georgia Tech College of Computing
Computational Science and Engineering

Summary of Contributions

Self-correcting Algorithms

We introduce a new fault tolerant algorithm design principle that we call *self-correction*. A self-correcting algorithm remains in a valid state, despite the faulty execution of an iteration, under the assumption that its previous state was a valid one.

Self-Correcting Connected Components Algorithm

- We apply the ideas of self-correction to Label-propagation algorithm for graph connected component problem.
- Assumes availability of selective reliability mode.
- Requires $\mathcal{O}(V)$ additional storage and computations per iteration compared to $\mathcal{O}(|V| + |E|)$ cost for the baseline algorithm.
- 10-35% increases in execution time for one error for 64 memory operations.

Summary of Contributions

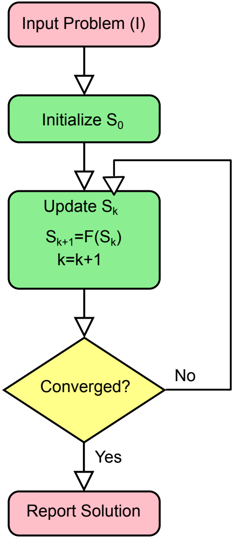
Self-correcting Algorithms

We introduce a new fault tolerant algorithm design principle that we call *self-correction*. A self-correcting algorithm remains in a valid state, despite the faulty execution of an iteration, under the assumption that its previous state was a valid one.

Self-Correcting Connected Components Algorithm

- We apply the ideas of self-correction to Label-propagation algorithm for graph connected component problem.
- Assumes availability of selective reliability mode.
- Requires $\mathcal{O}(V)$ additional storage and computations per iteration compared to $\mathcal{O}(|V| + |E|)$ cost for the baseline algorithm.
- 10-35% increases in execution time for one error for 64 memory operations.

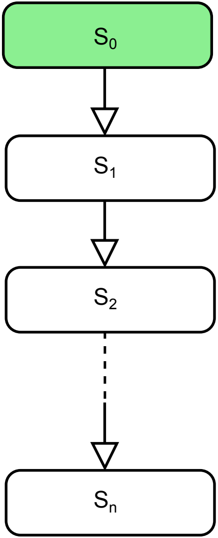
Iterative Algorithms



Iterative Algorithms

- A typical iterative algorithm has following components:
  1. The input problem;
  2. Intermediate variable;
  3. Update relation;
  4. Convergence checking; and
  5. Solution.

Iterative Algorithm as State Machine

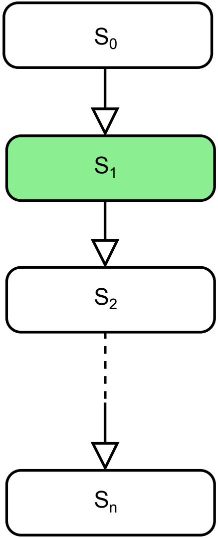


Iterative Algorithms as State Machines

- An iterative algorithm can be viewed as state machine.
- State of the algorithm: subset of intermediate variables required for continued execution of the algorithm.
- Starts with an initial state S_0
- Uses update relation to transition from one state to another

$$S_{k+1} \leftarrow S_k$$

Iterative Algorithm as State Machine

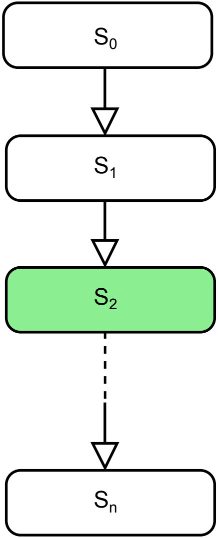


Iterative Algorithms as State Machines

- An iterative algorithm can be viewed as state machine.
- State of the algorithm: subset of intermediate variables required for continued execution of the algorithm.
- Starts with an initial state S_0
- Uses update relation to transition from one state to another

$$S_{k+1} \leftarrow S_k$$

Iterative Algorithm as State Machine

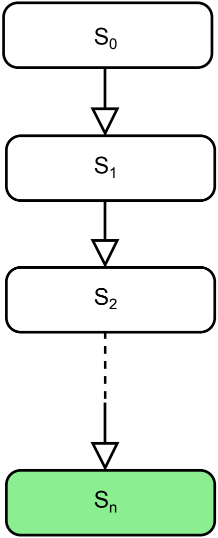


Iterative Algorithms as State Machines

- An iterative algorithm can be viewed as state machine.
- State of the algorithm: subset of intermediate variables required for continued execution of the algorithm.
- Starts with an initial state S_0
- Uses update relation to transition from one state to another

$$S_{k+1} \leftarrow S_k$$

Iterative Algorithm as State Machine

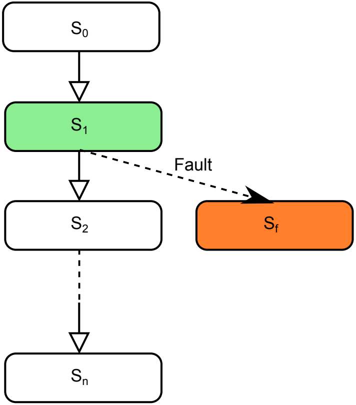


Iterative Algorithms as State Machines

- An iterative algorithm can be viewed as state machine.
- State of the algorithm: subset of intermediate variables required for continued execution of the algorithm.
- Starts with an initial state S_0
- Uses update relation to transition from one state to another

$$S_{k+1} \leftarrow S_k$$

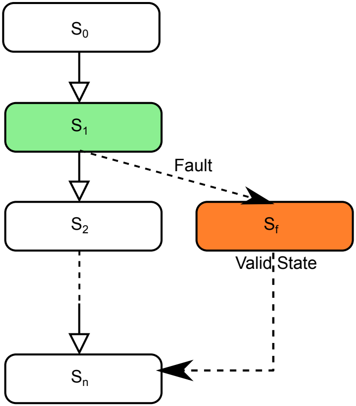
Single Fault in Iterative Algorithm



Valid and Invalid States

- Valid state: under fault-free execution of the algorithm from that state, the algorithm will converge to the correct solution; otherwise invalid.
- In fault free execution, the algorithm always remains in a valid state.
- Any hardware fault can cause the algorithm to reach an invalid state.
- In general determining whether a given state is valid or not, is non-trivial.

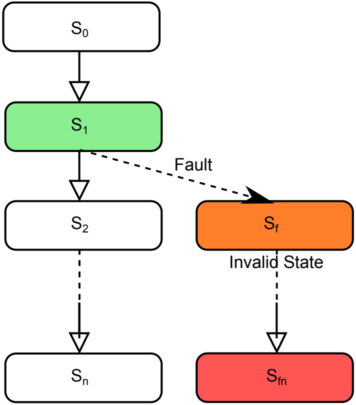
Single Fault in Iterative Algorithm



Valid and Invalid States

- Valid state: under fault-free execution of the algorithm from that state, the algorithm will converge to the correct solution; otherwise invalid.
- In fault free execution, the algorithm always remains in a valid state.
- Any hardware fault can cause the algorithm to reach an invalid state.
- In general determining whether a given state is valid or not, is non-trivial.

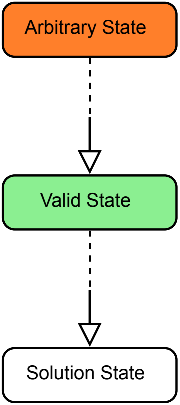
Single Fault in Iterative Algorithm



Valid and Invalid States

- Valid state: under fault-free execution of the algorithm from that state, the algorithm will converge to the correct solution; otherwise invalid.
- In fault free execution, the algorithm always remains in a valid state.
- Any hardware fault can cause the algorithm to reach an invalid state.
- In general determining whether a given state is valid or not, is non-trivial.

Self-stabilizing Algorithms



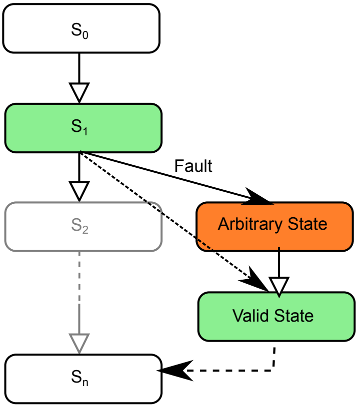
Self-stabilizing Algorithms

- Starting from any arbitrary state, valid or invalid, a self-stabilizing algorithm will reach a valid in finite number of steps.
- Natural fault-tolerance mechanism.
- Examples: Stationary iterations, Newton Iteration.
- Self-stabilization is a *strong* property.

Scala'13

- Self-stabilizing Steepest Descent and Conjugate Gradient.
- Periodic state correction.
- May not be generalized to all iterative algorithms.

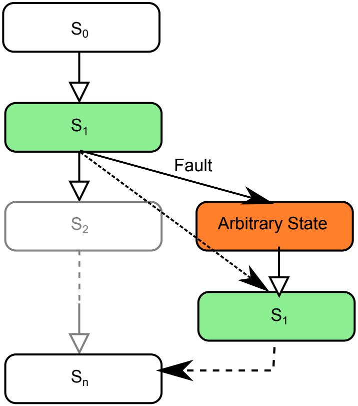
Self-Correcting Algorithms



Self-correcting Algorithms

- A self-correcting algorithm is an iterative algorithm that, starting in some valid state, remains in a valid state or comes to a valid state in finite number of steps even if a fault occurs.
- Requires that algorithm starts from a valid state.
- Uses information from previously known valid state.
- Example: Checkpoint-restart, FT-GMRES.

Checkpoint-restart as a Self-correcting algorithm



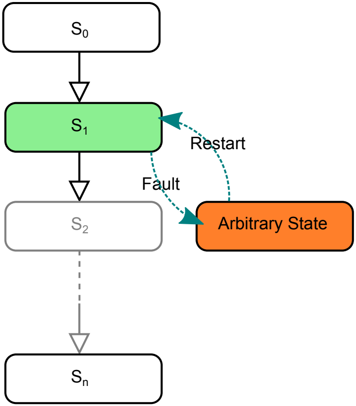
Checkpoint-restart based fault tolerance

- Bring to valid state by restoring a check-pointed valid state.
- At high fault rate, algorithm will not make any progress.

Broader idea of self-correction is to use S_1 to construct an state

$$\tilde{S}_2 \approx S_2$$

Checkpoint-restart as a Self-correcting algorithm



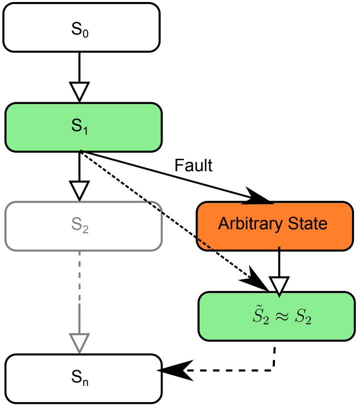
Checkpoint-restart based fault tolerance

- Bring to valid state by restoring a check-pointed valid state.
- At high fault rate, algorithm will not make any progress.

Broader idea of self-correction is to use S_1 to construct an state

$$\tilde{S}_2 \approx S_2$$

Checkpoint-restart as a Self-correcting algorithm



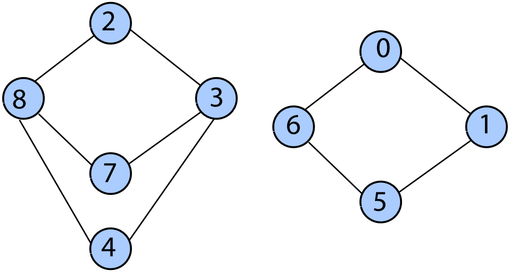
Checkpoint-restart based fault tolerance

- Bring to valid state by restoring a check-pointed valid state.
- At high fault rate, algorithm will not make any progress.

Broader idea of self-correction is to use S_1 to construct an state

$$\tilde{S}_2 \approx S_2$$

Label Propagation Algorithm for Graph Connected Component Algorithm

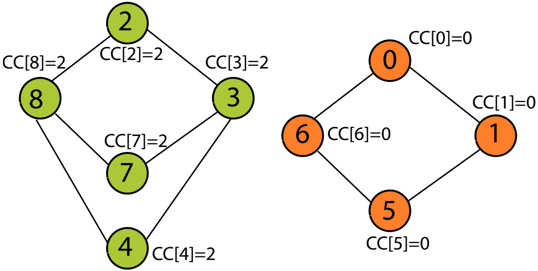


Graph Connected-component Problem

We seek to find number of connected-components in the graph and which connected component each vertex belongs to.

- Used for community detection, centrality analytics and streaming graph analytics.
- Label propagation is highly suited for parallel computing.

Label Propagation Algorithm for Graph Connected Component Algorithm

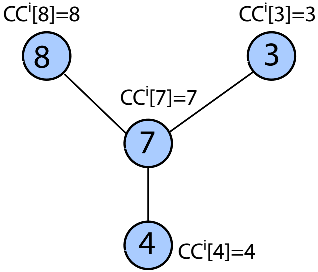


Graph Connected-component Problem

We seek to find number of connected-components in the graph and which connected component each vertex belongs to.

- Used for community detection, centrality analytics and streaming graph analytics.
- Label propagation is highly suited for parallel computing.

Label Propagation Algorithm

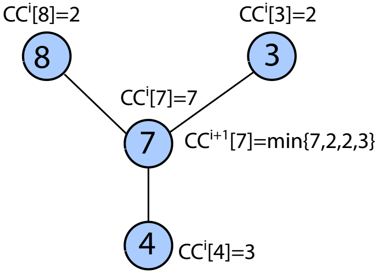


- Propagates the minimum label in the connected component.
- We maintain a *label* array CC for each vertex.
- CC is initialized to vertex id for each vertex.
- In each iteration, each vertex calculates minimum label among all near-neighbours $\mathcal{N}(v) = v \cup adj(v)$

$$CC^i[v] = \min_{u \in \mathcal{N}(v)} CC^{i-1}[u].$$

- Iteration terminates when no change occur in an iteration.

Label Propagation Algorithm

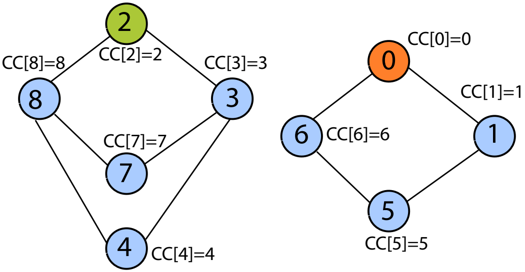


- Propagates the minimum label in the connected component.
- We maintain a *label* array CC for each vertex.
- CC is initialized to vertex id for each vertex.
- In each iteration, each vertex calculates minimum label among all near-neighbours $\mathcal{N}(v) = v \cup adj(v)$

$$CC^i[v] = \min_{u \in \mathcal{N}(v)} CC^{i-1}[u].$$

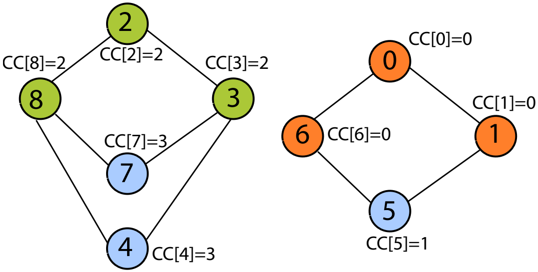
- Iteration terminates when no change occur in an iteration.

Label Propagation Algorithm- Example



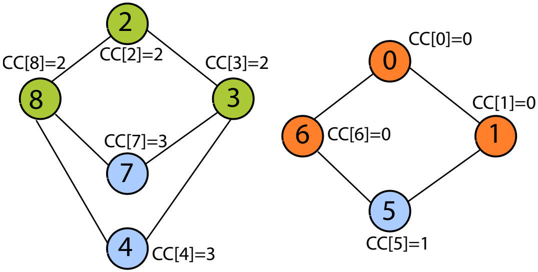
- Minimum label in the connected component (shown in green and orange) propagates to all the vertex in the connected components.

Label Propagation Algorithm- Example



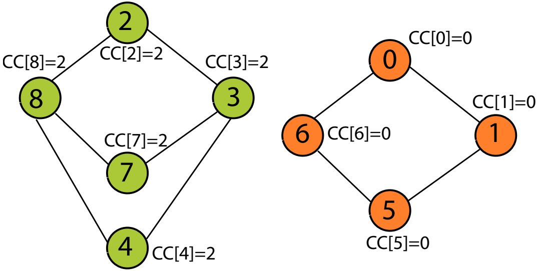
- Minimum label in the connected component (shown in green and orange) propagates to all the vertex in the connected components.

Label Propagation Algorithm- Example



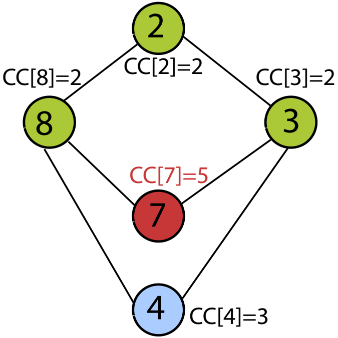
- Minimum label in the connected component (shown in green and orange) propagates to all the vertex in the connected components.

Label Propagation Algorithm- Example



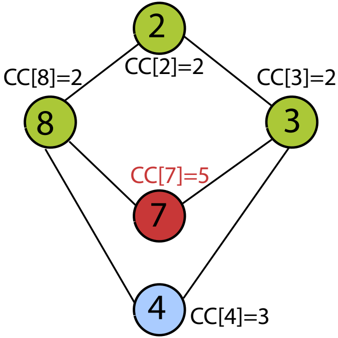
- Minimum label in the connected component (shown in green and orange) propagates to all the vertex in the connected components.

State of Label Propagation Algorithm



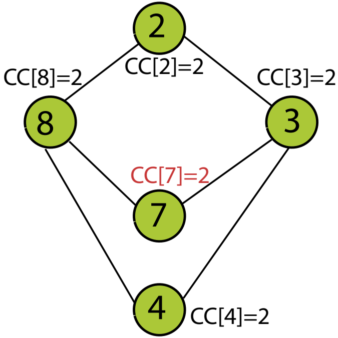
- *CC* vector defines the state of the LP algorithm.

Single Fault In Label Propagation Algorithm- Fault Correction



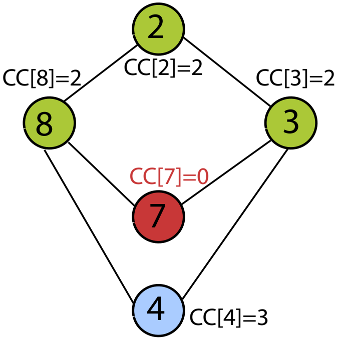
- CC value can be corrupted due to hardware faults.
- Depending on error caused by the fault, some error can be corrected by the algorithm.
- Example: corrupted $CC[v]$ value is arbitrarily high.
- Such faults do not cause the algorithm to transition to an invalid state, but they can still cause delay in convergence.

Single Fault In Label Propagation Algorithm- Fault Correction



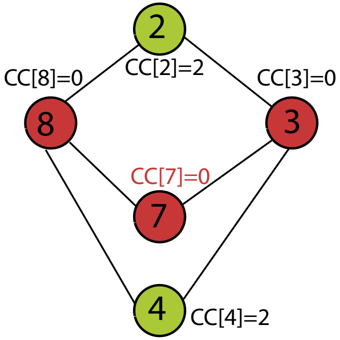
- *CC* value can be corrupted due to hardware faults.
- Depending on error caused by the fault, some error can be corrected by the algorithm.
- Example: corrupted $CC[v]$ value is arbitrarily high.
- Such faults do not cause the algorithm to transition to an invalid state, but they can still cause delay in convergence.

Single Fault In Label Propagation Algorithm- Fault Propagation



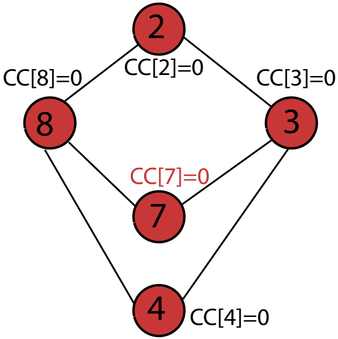
- If the fault causes a corruption such $CC[v]$ is changed to a values lower than the $CC^{\infty}[v]$, the error will propagate to all the other vertex in the connected component.
- Such faults do not cause the algorithm to transition to an invalid state.
- It is not computationally easy to determine whether a given snapshot of CC is a valid state.

Single Fault In Label Propagation Algorithm- Fault Propagation



- If the fault causes a corruption such $CC[v]$ is changed to a values lower than the $CC^{\infty}[v]$, the error will propagate to all the other vertex in the connected component.
- Such faults do not cause the algorithm to transition to an invalid state.
- It is not computationally easy to determine whether a given snapshot of CC is a valid state.

Single Fault In Label Propagation Algorithm- Fault Propagation



- If the fault causes a corruption such $CC[v]$ is changed to a values lower than the $CC^{\infty}[v]$, the error will propagate to all the other vertex in the connected component.
- Such faults do not cause the algorithm to transition to an invalid state.
- It is not computationally easy to determine whether a given snapshot of CC is a valid state.

Valid States

Classification of different corruption

Consider following three cases:

1. $CC[v] > v$: Easy to detect and automatically corrected in most cases.
2. $CC^{\infty}[v] \leq CC[v] \leq v$: ??
3. $CC[v] < CC^{\infty}[v]$: Will definitely cause algorithm to fail.

Theorem

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,

$$CC^{\infty}[v] \leq CC[v] \leq v.$$

The only problem is, $CC^{\infty}[v]$—being the solution, is not known.

Valid States

Classification of different corruption

Consider following three cases:

1. $CC[v] > v$: Easy to detect and automatically corrected in most cases.
2. $CC^{\infty}[v] \leq CC[v] \leq v$: ??
3. $CC[v] < CC^{\infty}[v]$: Will definitely cause algorithm to fail.

Theorem

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,

$$CC^{\infty}[v] \leq CC[v] \leq v.$$

The only problem is, $CC^{\infty}[v]$—being the solution, is not known.

Valid States

Classification of different corruption

Consider following three cases:

1. $CC[v] > v$: Easy to detect and automatically corrected in most cases.
2. $CC^{\infty}[v] \leq CC[v] \leq v$: ??
3. $CC[v] < CC^{\infty}[v]$: Will definitely cause algorithm to fail.

Theorem

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,

$$CC^{\infty}[v] \leq CC[v] \leq v.$$

The only problem is, $CC^{\infty}[v]$—being the solution, is not known.

Valid States

Classification of different corruption

Consider following three cases:

1. $CC[v] > v$: Easy to detect and automatically corrected in most cases.
2. $CC^{\infty}[v] \leq CC[v] \leq v$: ??
3. $CC[v] < CC^{\infty}[v]$: Will definitely cause algorithm to fail.

Theorem

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,

$$CC^{\infty}[v] \leq CC[v] \leq v.$$

The only problem is, $CC^{\infty}[v]$—being the solution, is not known.

Valid States

Classification of different corruption

Consider following three cases:

1. $CC[v] > v$: Easy to detect and automatically corrected in most cases.
2. $CC^{\infty}[v] \leq CC[v] \leq v$: ??
3. $CC[v] < CC^{\infty}[v]$: Will definitely cause algorithm to fail.

Theorem

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,

$$CC^{\infty}[v] \leq CC[v] \leq v.$$

The only problem is, $CC^{\infty}[v]$—being the solution, is not known.

Self-correcting Label Propagation Algorithm- 1

We apply principle of self-correction to resolve the apparent difficulty in verifying state validity.

- We assume the previous state CC^{i-1} is a valid one.
- Checking
$$CC^{i}[v] = \min_{u \in \mathcal{N}(v)} CC^{i-1}[u],$$
where $\mathcal{N}(v) \equiv v \cup adj(v)$ is the immediate *neighborhood* of v, will require re-computing entire iteration.
- We show that $CC^{i}[v]$ is still a valid value even if we can relax the minimization criterion to
$$CC^{i}[v] \in \{CC^{i-1}[u] \mid u \in \mathcal{N}(v)\},$$

Self-correcting Label Propagation Algorithm- 1

We apply principle of self-correction to resolve the apparent difficulty in verifying state validity.

- We assume the previous state CC^{i-1} is a valid one.
- Checking
$$CC^i[v] = \min_{u \in \mathcal{N}(v)} CC^{i-1}[u],$$
where $\mathcal{N}(v) \equiv v \cup adj(v)$ is the immediate *neighborhood* of v, will require re-computing entire iteration.
- We show that $CC^i[v]$ is still a valid value even if we can relax the minimization criterion to
$$CC^i[v] \in \{CC^{i-1}[u] \mid u \in \mathcal{N}(v)\},$$

Self-correcting Label Propagation Algorithm- 1

We apply principle of self-correction to resolve the apparent difficulty in verifying state validity.

- We assume the previous state CC^{i-1} is a valid one.
- Checking
$$CC^{i}[v] = \min_{u \in \mathcal{N}(v)} CC^{i-1}[u],$$
where $\mathcal{N}(v) \equiv v \cup adj(v)$ is the immediate *neighborhood* of v, will require re-computing entire iteration.
- We show that $CC^{i}[v]$ is still a valid value even if we can relax the minimization criterion to
$$CC^{i}[v] \in \{CC^{i-1}[u] \mid u \in \mathcal{N}(v)\},$$

Self-correcting Label Propagation Algorithm- 1

We apply principle of self-correction to resolve the apparent difficulty in verifying state validity.

- We assume the previous state CC^{i-1} is a valid one.
- Checking
$$CC^i[v] = \min_{u \in \mathcal{N}(v)} CC^{i-1}[u],$$
where $\mathcal{N}(v) \equiv v \cup adj(v)$ is the immediate *neighborhood* of v, will require re-computing entire iteration.
- We show that $CC^i[v]$ is still a valid value even if we can relax the minimization criterion to
$$CC^i[v] \in \{CC^{i-1}[u] \mid u \in \mathcal{N}(v)\},$$

Self-correcting Label Propagation Algorithm- 2

Theorem

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. $CC^i[v] \leq v$; *and*
2. *there exists a vertex u such that $CC^i[v] = CC^{i-1}[u]$ and $u \in \mathcal{N}(v)$.*

Cost of Direct Verification

- Verifying $CC^i[v] \leq v$ requires $\mathcal{O}(V)$ operation.
- Verifying second condition requires traversing adjacency list for each vertex v, that will require $\mathcal{O}(V + E)$ operations, as costly as an LP iteration.

Self-correcting Label Propagation Algorithm- 2

Theorem

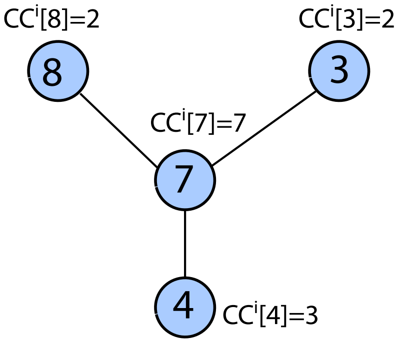
Given a valid state for the previous iteration, CC^{i-1}, the current connected component array CC^i is a valid state if for all vertices v, CC^i satisfies these conditions:

1. $CC^i[v] \leq v$; *and*
2. *there exists a vertex u such that $CC^i[v] = CC^{i-1}[u]$ and $u \in \mathcal{N}(v)$.*

Cost of Direct Verification

- Verifying $CC^i[v] \leq v$ requires $\mathcal{O}(V)$ operation.
- Verifying second condition requires traversing adjacency list for each vertex v, that will require $\mathcal{O}(V + E)$ operations, as costly as an LP iteration.

Validity Checking: Auxiliary Data Structure- 1



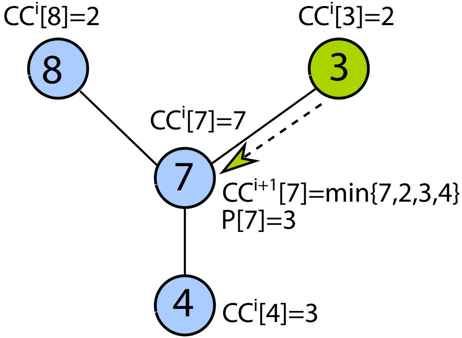
Parent Array

- Parent array P: We may store store information of the vertex u that caused the last change in $CC[v]$.
- If $u = P[v]$ then $CC^i[v] = CC^{i-1}[P[v]]$, can be verified in $\mathcal{O}(V)$ operations for all vertex.
- Storing P requires an memory of $\mathcal{O}(V)$.

Corruption of P

- P also can be corrupt.
- P is valid if $P[v] \in \mathcal{N}(v)$ for all vertex v.
- Checking P is valid requires again $\mathcal{O}(V + E)$ operations.

Validity Checking: Auxiliary Data Structure- 1



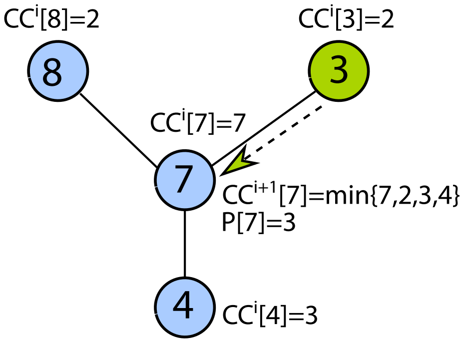
Parent Array

- Parent array P: We may store store information of the vertex u that caused the last change in $CC[v]$.
- If $u = P[v]$ then $CC^i[v] = CC^{i-1}[P[v]]$, can be verified in $\mathcal{O}(V)$ operations for all vertex.
- Storing P requires an memory of $\mathcal{O}(V)$.

Corruption of P

- P also can be corrupt.
- P is valid if $P[v] \in \mathcal{N}(v)$ for all vertex v.
- Checking P is valid requires again $\mathcal{O}(V + E)$ operations.

Validity Checking: Auxiliary Data Structure- 1



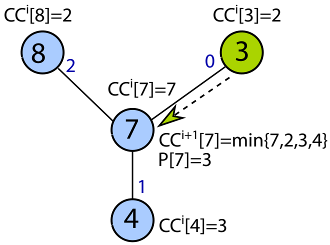
Parent Array

- Parent array P: We may store store information of the vertex u that caused the last change in $CC[v]$.
- If $u = P[v]$ then $CC^i[v] = CC^{i-1}[P[v]]$, can be verified in $\mathcal{O}(V)$ operations for all vertex.
- Storing P requires an memory of $\mathcal{O}(V)$.

Corruption of P

- P also can be corrupt.
- P is valid if $P[v] \in \mathcal{N}(v)$ for all vertex v.
- Checking P is valid requires again $\mathcal{O}(V + E)$ operations.

Validity Checking: Auxiliary Data Structure- 2



Index Based Parent Array

- Instead of storing u, we store index of u in $adj(v)$.

$$
\begin{aligned}
E &\leftarrow adj(v) \\
u &\leftarrow E[k] \\
P^*[v] &\leftarrow k
\end{aligned}
$$

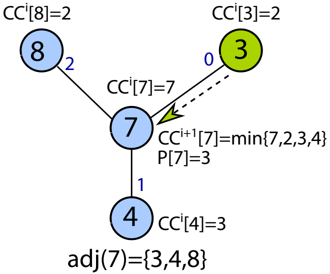
- When $P[v] = v$, then $P^*[v] = -1$
- $CC^i[v] = CC^{i-1}[P[v]]$ reduces to

$$CC^i[v] = CC^{i-1}[adj(v)[P^*[v]]];$$

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

- Validity of P^*:
$-1 \leq P^*[v] < |adj(v)|$; $\mathcal{O}(V)$ operations.

Validity Checking: Auxiliary Data Structure- 2



Index Based Parent Array

- Instead of storing u, we store index of u in $adj(v)$.

$$E \leftarrow adj(v)$$
$$u \leftarrow E[k]$$
$$P^*[v] \leftarrow k$$

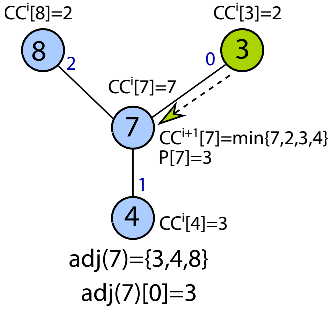
- When $P[v] = v$, then $P^*[v] = -1$
- $CC^i[v] = CC^{i-1}[P[v]]$ reduces to

$$CC^i[v] = CC^{i-1}[adj(v)[P^*[v]]];$$

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

- Validity of P^*:
$-1 \leq P^*[v] < |adj(v)|$; $\mathcal{O}(V)$ operations.

Validity Checking: Auxiliary Data Structure- 2



Index Based Parent Array

- Instead of storing u, we store index of u in $adj(v)$.

$$\begin{aligned} E &\leftarrow adj(v) \\ u &\leftarrow E[k] \\ P^*[v] &\leftarrow k \end{aligned}$$

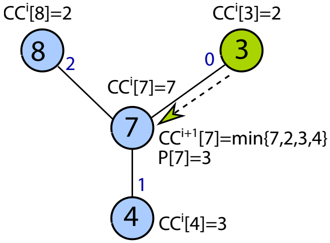
- When $P[v] = v$, then $P^*[v] = -1$
- $CC^i[v] = CC^{i-1}[P[v]]$ reduces to

$$CC^i[v] = CC^{i-1}[adj(v)[P^*[v]]];$$

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

- Validity of P^*:
$-1 \leq P^*[v] < |adj(v)|$; $\mathcal{O}(V)$ operations.

Validity Checking: Auxiliary Data Structure- 2



adj(7)={3,4,8}

adj(7)[0]=3

P*[7]=0

Index Based Parent Array

- Instead of storing u, we store index of u in $adj(v)$.

$$\begin{aligned} E &\leftarrow adj(v) \\ u &\leftarrow E[k] \\ P^*[v] &\leftarrow k \end{aligned}$$

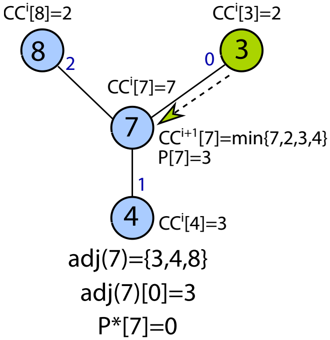
- When $P[v] = v$, then $P^*[v] = -1$
- $CC^i[v] = CC^{i-1}[P[v]]$ reduces to

$$CC^i[v] = CC^{i-1}[adj(v)[P^*[v]]];$$

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

- Validity of P^*:
$-1 \leq P^*[v] < |adj(v)|$; $\mathcal{O}(V)$ operations.

Validity Checking: Auxiliary Data Structure- 2



Index Based Parent Array

- Instead of storing u, we store index of u in $adj(v)$.

$$\begin{aligned} E &\leftarrow adj(v) \\ u &\leftarrow E[k] \\ P^*[v] &\leftarrow k \end{aligned}$$

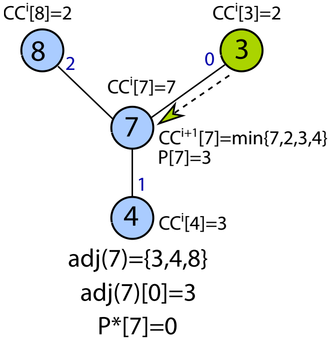
- When $P[v] = v$, then $P^*[v] = -1$
- $CC^i[v] = CC^{i-1}[P[v]]$ reduces to

$$CC^i[v] = CC^{i-1}[adj(v)[P^*[v]]];$$

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

- Validity of P^*: $-1 \leq P^*[v] < |adj(v)|$; $\mathcal{O}(V)$ operations.

Validity Checking: Auxiliary Data Structure- 2



Index Based Parent Array

- Instead of storing u, we store index of u in $adj(v)$.

$$\begin{aligned} E &\leftarrow adj(v) \\ u &\leftarrow E[k] \\ P^*[v] &\leftarrow k \end{aligned}$$

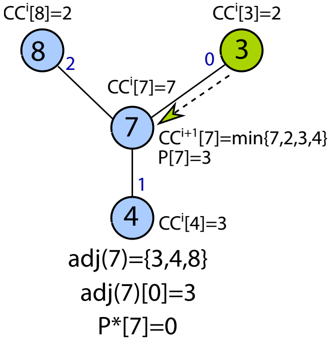
- When $P[v] = v$, then $P^*[v] = -1$
- $CC^i[v] = CC^{i-1}[P[v]]$ reduces to

$$CC^i[v] = CC^{i-1}[adj(v)[P^*[v]]];$$

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

- Validity of P^*: $-1 \leq P^*[v] < |adj(v)|$; $\mathcal{O}(V)$ operations.

Validity Checking: Auxiliary Data Structure- 2



Index Based Parent Array

- Instead of storing u, we store index of u in $adj(v)$.

$$
\begin{aligned}
E &\leftarrow adj(v) \\
u &\leftarrow E[k] \\
P^*[v] &\leftarrow k
\end{aligned}
$$

- When $P[v] = v$, then $P^*[v] = -1$
- $CC^i[v] = CC^{i-1}[P[v]]$ reduces to

$$CC^i[v] = CC^{i-1}[adj(v)[P^*[v]]];$$

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

- Validity of P^*: $-1 \leq P^*[v] < |adj(v)|$; $\mathcal{O}(V)$ operations.

Fault Detection and Correction

Invalid State Detection

In summary, the set of conditions to check for each vertex are:

$$
\begin{aligned}
CC^i[v] &\leq v; \\
-1 \leq P^*[v] &< |adj(v)|; \text{ and} \\
CC^i[v] &= \begin{cases} v & \text{if } P^*[v] = -1 \\ CC^{i-1}[adj(v)[P^*[v]]] & \text{if } P^*[v] \neq -1 \end{cases}.
\end{aligned}
$$

State Correction

For any vertex v, if state validity check fails then, we recompute $CC^i[v]$.

Fault Detection and Correction

Invalid State Detection

In summary, the set of conditions to check for each vertex are:

$$
\begin{aligned}
&CC^i[v] \leq v; \\
&\quad -1 \leq P^*[v] < |adj(v)|; \text{and} \\
&CC^i[v] = \begin{cases} v & \text{if } P^*[v] = -1 \\ CC^{i-1}[adj(v)[P^*[v]]] & \text{if } P^*[v] \neq -1 \end{cases}.
\end{aligned}
$$

State Correction

For any vertex v, if state validity check fails then, we recompute $CC^i[v]$.

Overhead of Self-correcting Label-propagation Algorithm

Overhead	Asymptotic Complexity
Fault detection	$\mathcal{O}(\lvert V\rvert)$
Fault correction	$\mathcal{O}(f\lvert E\rvert/\lvert V\rvert)$
Auxiliary data structure	$\mathcal{O}(\lvert V\rvert)$

- Number of state corrections f, can be significantly less than faults occurred.
- Fault detection and correction needs to be done in a guaranteed reliable mode.

Experimental Setup

Machine Parameter

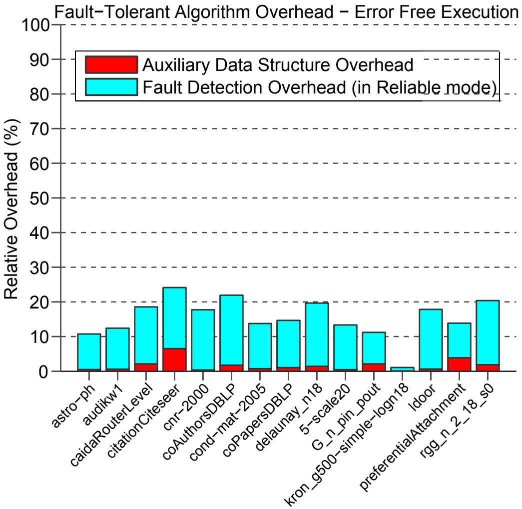
Prop	SNB16c
Micro-architecture	Sandy-Bridge
Sockets×Cores	2×8
Clock Rate	2.4GHz
DRAM capacity	128GB
DRAM Bandwidth	72GB/s
Compiler	Intel "C" compiler 15.0.0

Fault Injection

- Fault injection in reading graph data structure and *CC* array.
- Each fault injection read is independent
- Normalized by number of edges in the network

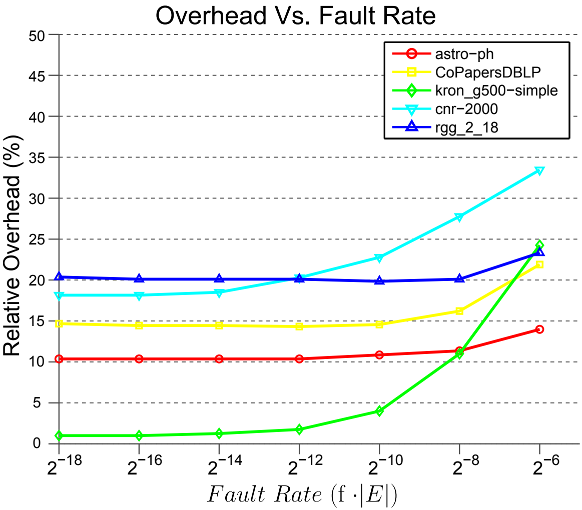
- Test Network: 14^{th} DIMACS graph challenge

Fault Free Execution Overhead

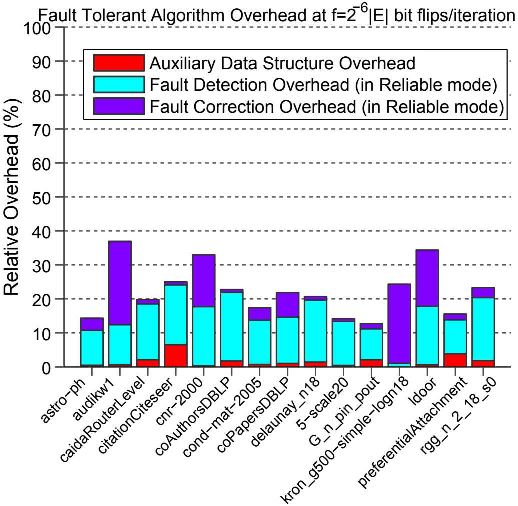


On an average 1.3% overhead for maintaining additional data structure and 14% for fault detection.

Overhead of Fault Tolerant Algorithm in the Presence of Faults



Overhead of Fault Tolerant Algorithm in the Presence of Faults



Fault correction adds additional 9% overhead at 2^{-6} bit flips per every memory access.

Conclusion

Conclusion

- We introduced the ideas of self-correcting algorithm to build fault tolerant algorithms.
- We presented a self-correcting label propagation algorithm for graph connected component problem.
- Key steps involved:
  - Analyze valid and invalid state;
  - Use self-correction hypothesis to simplify invalid state detection;
  - Use previous valid states to recover from invalid state.
- Asymptotically lower overhead for fault detection and correction.
- 10-35% increases in execution time for one error for 64 memory operations.