Sorting in the Presence of Memory faults (without Redundancy)

-Irene Finocchi

- Giuseppe F. Italiano

Why Resiliency

- Computer platforms with large and inexpensive memories, which are also error-prone
- Consider for instance mergesort: during the merge step, errors may propagate due to corrupted keys (having value larger than the correct one).

Defining Resilience

- An algorithm is resilient to memory faults if, despite the corruption of some memory values before or during its execution, the algorithm is nevertheless able to get a correct output on the set of uncorrupted values.

Assumption Made

- faults may happen *at any time:*
- faults may happen *at any place*
- An algorithms can exploit *O(1) reliable memory words, whose* content gets never corrupted.
- Moving variables around in memory is an atomic operation.
- Total errors that can happen is δ
- Actual number of error is α ($\alpha \leq \delta$)

Trivial Method of Sorting (with redundancy)

- If each value were replicated *k times, by majority techniques* we could easily tolerate up to (k − 1)/2 *faults.* [$\delta = (k-1)/2$]
- *The* algorithm's overhead in terms of both space and running time would also be $\Theta(k)$.
- In order to be resilient to $O(n^{1/2})$ *faults, a sorting algorithm* would require $O(n^{3/2} \log n)$ *time and* $O(n^{3/2})$ *space.*

Defining Resiliency in Sorting

- Algorithm described do not wish to recover corrupted data, but simply wants to correct on uncorrupted data, without incurring much of any time or space overhead.
- Given a set of *n keys that* need to be sorted. The value of at most *δ keys can be* arbitrarily corrupted (either increased or decreased) during the sorting process. A sorting algorithm is *fault-tolerant if it correctly orders the set of uncorrupted* keys.
- if keys get corrupted at the very end of the algorithm execution, we cannot prevent them from occupying wrong positions in the output sequence.

Some Preliminary Definition

- Definition 1. Faithfully ordered
- *A sequence is faithfully ordered if its uncorrupted keys are sorted.*

- Definition 2. K-unordered
- *A sequence is k-unordered if k is the minimum number of keys whose removal makes the remaining subsequence sorted.*

Note: each faithfully ordered sequence is k-unordered for some $k \leq \delta$, where $\delta \leq n$

- Definition 3. *strongly* fault tolerant *merging algorithm*
- *A sorting or merging algorithm is strongly* fault tolerant *if it produces a faithfully ordered sequence, i.e., it correctly sorts all of the uncorrupted keys.*

- Definition 4. *is k-weakly* fault tolerant *merging algorithm*
- *A sorting or merging algorithm is k-weakly* fault tolerant *if it produces a k-unordered sequence, i.e., if it correctly sorts all but k keys.*

Note: a strongly fault tolerant algorithm is δ-weakly fault tolerant.

Naive fault-tolerant sorting

- A fault-tolerant algorithm that sorts all the correct keys in *$O(\delta \cdot n \log n)$ worst-case time can be easily obtained from* merge-sort.
- At each merge step, instead of taking the minimum among two keys, we take the minimum among ($2\delta + 2$) keys, *$\delta+1$ per sequence; since there can be at most δ errors,* at least one correct key per sequence is considered.
- In order to avoid problems in the recursion stack, we use the standard iterative bottom-up implementation of merge-sort, sorting all the sequences of length $2i$ *before any sequence of length $2i+1$, for $i = 1$ up to $\log n$*

Naive-Merge-sort Analysis

- The running time is $O(\delta\ n \log n)$ in the worst case, and it becomes $O(\delta\ n)$ *when* $\delta = \Omega(n^{\varepsilon})$, for some $\varepsilon > 0$.

Purifying k-unordered sequences

- Build Stack & list of Discarded Keys as Follows:
- Top of the stack and the index *i* that scans *X in the O(1)-size reliable memory*
- At the *i-th* step,

  if *X[i]* $\geq$ *top,*

  push it onto stack

  else

  add both the top and *X[i]* to list of discarded keys, pop the stack

  compute the maximum of the topmost $\delta + 1$ *keys*

  move it to the top.

Analysis Purifying k-unordered sequences

- Invariant 1 (Stack Invariant). *Throughout the algorithm, the key on the top is larger than or equal to all the keys that have not been corrupted since they were pushed onto the stack.*

- Proof by induction
- top of the stack is fault-free, because it is stored in reliable memory.
- *The base step, with the* stack containing just one element, holds.
- Assuming: the invariant holds at the beginning of the *i-th step.*
- *If* $X[i] \geq top$, *$X[i]$ is pushed onto stack and invariant* remains satisfied by transitivity.
- If $X[i] < top$, *the stack is* popped and the invariant may be no longer satisfied if the key below the discarded top got corrupted (namely, if its value was decreased).
- In this case, let *m be the maximum* of the topmost $\delta + 1$ *keys: m is new top.*
- Now at least one of the $\delta+1$ *considered keys is correct:*
- let *x be any such correct key. At the time when x was at* the top, the invariant was true by inductive hypothesis, and therefore *x is still larger than or equal to all the uncorrupted* keys below its position.
- Since $m \geq x$, *the new top satisfies* the invariant with respect to the entire stack.

Analysis Purifying k-unordered sequences

- *Lemma 1: (Remember)*
- *Algorithm Purify computes a faithfully ordered subsequence S of a k-unordered sequence X of length n in $O(n+\delta\cdot(k+\alpha))$ worst-case time, where $\alpha \leq \delta$ is the actual number of memory faults introduced during the execution of* Purify*.*

O($\alpha.\delta$)-Weakly Fault Tolerant merge algorithm

- Let *A and B be the sequences to be merged.*
- Let *i and j be the indices to* arrays *A and B, respectively.*
- In addition to comparing *A[i] and B[j] and advancing one of the two* indices, the algorithm updates two additional variables, respectively called *wait-A and wait-B*

Note: Indices, wait variables and counter t are all stored in O(1)-size reliable memory.

O($\alpha.\delta$)-Weakly Fault Tolerant merge algorithm

- If A[i] added to output sequence
  - Wait-A = 0
  - Wait-B ++
- If B[j] added to output sequence
  - Wait-A ++
  - Wait-B = 0
- If(Wait-A = 2δ+1) (wlog for B)
  - Wait-A = 0
  - Wait-B = 0
  - For (k = i+1 till i+2δ+1)
    - If(A[i]<A[k]) t++
- If(t ≥ δ+1) (wlog for B)
  - Output A[i] & i++ (i.e. A[i] is corrupted)
- If(t < δ+1) (wlog for B)
  - Algorithm cannot decide whether A[i] is corrupted or not

O($\alpha.\delta$)-Weakly Fault Tolerant merge algorithm (Analysis)

- *Lemma 2 (remember)*
- *Given two faithfully ordered sequences of total length n, algorithm WFT-Merge merges the sequences in O(n) time and returns an O($\alpha \cdot \delta$)-unordered sequence, where $\alpha \leq \delta$ is the number of corrupted keys at the end of the algorithm execution.*

(δ-weakly) Strongly fault-tolerant merge Algorithm

- Let *A and B be the sequences to be merged of length n1 & n2 respectively.*
- *Without loss of generality: n2 ≤ n1*
- Let *i and j be the indices to* arrays *A and B, respectively.*
- Basic idea: Extract keys from shorter sequence B and place it in correct position w.r.t longer sequence A.

Note: Indices and counter t are all stored in O(1)-size reliable memory.

(δ-weakly) Strongly fault-tolerant merge Algorithm

- Extract Min from B[j:j+δ]
- Let b = B[h] be that minimum s.t. $j \leq h \leq j+\delta$
- Shift right all keys and move b to B[j]
- Now Scan A from left to right starting from i.
  - Add keys to output untill we find A[i] > b

  (since A[i] can be corrupted returning b before A[i] is wrong)

  Let t be count of keys < A[i] in the Window A[i+1:i+2δ+1]
- If ($t \geq \delta+1$) A[i] is corrupted and we continue the scanning
- If ($t < \delta$) divide window in 2 groups
  - Group1: keys < b
  - Group2: keys $\geq$ b

  And arrange s.t. group1 comes before b maintaining relative order

  Output keys of W $\leq$ b, followed by b and start new step.

(δ-weakly) Strongly fault-tolerant merge Algorithm

- Lemma 3 (Remember)
- *Let A and B be two faithfully ordered sequences of length n1 and n2, respectively, with n2 ≤ n1. Algorithm* SFT-Merge *faithfully merges the sequences in $O(n1+(n2+\alpha)\cdot\delta)$ time, where $\alpha \le \delta$ is the number of corrupted keys at the end of the algorithm execution.*

Solving the Jigsaw by placing the Subroutine

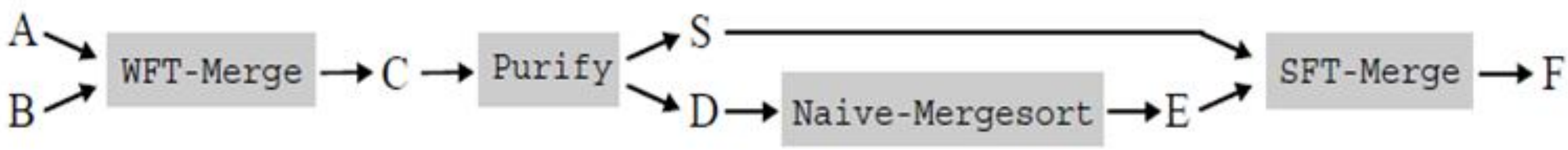

Figure 1: The merging algorithm.

- The input sequences, *A and B, are first merged using the* linear-time subroutine WFT-Merge.
- The output sequence, *C,* may not be faithfully ordered, i.e., some correct elements may be in a wrong position. Such errors are recovered by the combined use of Purify and SFT-Merge.

Note:

- The crux of merging algorithm is to use the slower strongly fault-tolerant subroutine on two unbalanced sequences
- The shorter(S) of which has length proportional to the actual number of corrupted.