FT-iSort: Efficient Fault Tolerance for Introsort

Sihuan Li
University of California, Riverside
Riverside, CA 92521
sli049@ucr.edu

Hongbo Li
University of California, Riverside
Riverside, CA 92521
hli035@ucr.edu

Xin Liang
University of California, Riverside
Riverside, CA 92521
xli007@ucr.edu

Jieyang Chen
Oak Ridge National Laboratory
Oak Ridge, TN 37831
chenj3@ornl.gov

Elisabeth Giem
University of California, Riverside
Riverside, CA 92521
gieme01@ucr.edu

Kaiming Ouyang
University of California, Riverside
Riverside, CA 92521
kouya001@ucr.edu

Kai Zhao
University of California, Riverside
Riverside, CA 92521
kzhao016@ucr.edu

Sheng Di
Argonne National Laboratory
Lemont, IL 60439
sdi1@anl.gov

Franck Cappello
Argonne National Laboratory
Lemont, IL 60439
cappello@mcs.anl.gov

Zizhong Chen
University of California, Riverside
Riverside, CA 92521
chen@cs.ucr.edu

ABSTRACT

Introspective sorting is a ubiquitous sorting algorithm which underlies many large scale distributed systems. Hardware-mediated soft errors can result in comparison and memory errors, and thus cause introsort to generate incorrect output, which in turn disrupts systems built upon introsort; hence, it is critical to incorporate fault tolerance capability within introsort. This paper proposes the first theoretically-sound, practical fault tolerant introsort with negligible overhead: FT-iSort. To tolerate comparison errors, we use minimal TMR protection via exploiting the properties of the effects of soft errors on introsort. This algorithm-based selective protection incurs far less overhead than naïve TMR protection designed to protect an entire application. To tolerate memory errors that escape DRAM error correcting code, we propose XOR-based re-execution. We incorporate our fault tolerance method into the well-known parallel sorting implementation HykSort, and we find that fault tolerant HykSort incurs negligible overhead and obtains nearly the same scalability as unprotected HykSort.

CCS CONCEPTS

• **Software and its engineering** → **Software fault tolerance**; *Software reliability*;

KEYWORDS

Algorithm Based Fault Tolerance, soft errors, comparison errors, fault tolerant sorting, introsort

ACM Reference Format:

Sihuan Li, Hongbo Li, Xin Liang, Jieyang Chen, Elisabeth Giem, Kaiming Ouyang, Kai Zhao, Sheng Di, Franck Cappello, and Zizhong Chen. 2019. FT-iSort: Efficient Fault Tolerance for Introsort. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19)*, November 17–22, 2019, Denver, CO, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3295500.3356195>

1 INTRODUCTION

Faults can happen since no hardware is perfectly reliable. Though they are rare in a single machine, they have become very common in large scale high performance computing (HPC) systems because of the extremely large amount of components [20]. For example, the supercomputer Jaguar has been shown to experience a double-bit-flip fault once a day in memory, cache, or even registers, and the frequency of faults is projected to be once every 4 minutes in exascale [21]. The decreasing chip size [42] and progressive energy-saving techniques [36] by overclocking or undervolting also disrupt stability of CPUs which increases the risk of faults even further. Many studies [10, 22] have shown the importance of fault tolerance to use exascale HPC systems effectively.

Faults have different impact on a process. If a fault crashes a process, it is called a fail-stop fault which can be covered by checkpoint/restart techniques [9, 31]. If a fault does not interrupt the process, it is a fail-continue fault. Though finished, the process may yield corrupted results which is not noticable by OS. Thus, such fault is also called silent data corruption or soft error which is the scope of this paper.

Soft errors that strike storage system such as memory or cache are effectively protected by error-correcting-code (ECC) though

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC '19, November 17–22, 2019, Denver, CO, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6229-0/19/11...\$15.00

<https://doi.org/10.1145/3295500.3356195>

multiple-bit-flips are still not detected. Soft errors that strike computing components such as arithmetic logic units are more difficult to handle at low cost. A general solution to these errors is triple modular redundancy [4] (TMR). In spite of its good generality and high error coverage, it doubles the hardware usage or time complexity to detect an error which will be tripled to correct an error.

Algorithm based fault tolerance [23] (ABFT) becomes an attractive option because of its much lower overheads than TMR. The idea of ABFT is to detect and correct soft errors based on a known invariant an algorithm holds. Thus, different algorithms have different ABFT solutions. For example, ABFT has been extended to matrix operations [11–14, 39–41] (Cholesky, QR, LU, SVD); iterative methods [15, 37] and data transformation kernel such as fast Fourier transform [28], based on their widely-used well-performed implementations to benefit larger user group.

Sorting is also one of the top 10 algorithms with greatest influence in computer science [3]. However, ABFT studies on sorting are limited. These works [5, 6, 18, 19, 26, 30, 32] are valuable in terms of theoretical analyses and pure algorithm designs. However, none of them can be efficiently applied to real world HPC environment due to non-negligible constants in their complexity bounds and sophisticated mechanism. For example, the experimental study [17] shows that fault tolerant sorting algorithm proposed by Finocchi et al. introduces 1.5X to 2X more running time than the original unprotected sorting.

To remedy this lack, this paper introduces the first efficient and practical fault tolerant sorting algorithm, FT-iSort, based on introspective sorting (introsort) since introsort delivers the best average performance of all sorting algorithms, and has been adopted by many standard libraries including the GNU Standard C++ Library, the SGI C++ Standard Template Library and the Microsoft .NET Framework Class Library. Introsort is widely used in various data-intensive scientific simulations and data analytics applications [35]. The stability of these computations depends heavily on introsort functioning correctly. However, soft errors can disrupt the sorting and cause introsort to yield unsorted sequences (because of comparison errors), or a sequence containing corrupted data (because of memory errors), which can in turn lead to unexpected results in the applications relying on introsort. The ABFT work on introsort can benefit a wider range of applications. Also, the error detection ability in FT-iSort will contribute to gathering the statistics of comparison error rates which is not existing in the literature to the best of our knowledge.

To tolerate comparison errors, we perform a rigorous study of the effect of soft errors on introsort, which reveals that only surprisingly light-weight protection is needed and allows us to use minimal TMR protection to comparison operations. This algorithm-based selective protection incurs far less overhead than naïve TMR protection designed to protect the entire application [27].

The only available method of tolerating memory errors that escape the DRAM error correcting code (ECC) is re-execution if there is no 100% more extra space to back up a copy of the memory. Pure re-execution is inefficient: we propose XOR-based re-execution. XOR is the result of exclusive or operation on entire array and should remain unchanged during sorting; by performing XOR in the beginning, middle, and end of sorting, we detect memory errors more quickly and initiate re-execution when the XOR varies.

The principal contributions of this paper are:

- We are the first to quantitatively prove the impact of comparison errors on the sortedness of quicksort output: Erroneous quicksort yields a nearly-sorted output and the unsorted data is bounded by the number of comparison errors e .
- We are the first to quantitatively prove the impact of comparison errors on the sortedness of heapsort output: Erroneous heapsort result output consists of a few sorted segments, and the number of these segments is bounded by $2e+1$.
- We propose a fault tolerant algorithm for introsort that protects against both comparison and memory errors, and implement it based on the widely-used GNU Standard C++ Library introsort. For comparison errors, naïve redundancy incurs 13-30% overhead in the presence of comparison errors and 0.9-13% overhead in the absence of errors; our strategy incurs 1.1-1.8% overhead in both cases. For memory errors our strategy reduces the overhead of naïve methods by 97% in the absence of errors, and by 61% in the presence of errors without the impact of round-off error.
- We apply our fault tolerance strategies to HykSort, the best known parallel sorting implementation based on the default sort in the GNU Standard C++ Library. We protect the sort with negligible overhead and achieve good scalability.

The paper is organized as follows. We introduce the basics related to introsort in Section 2, and we then theoretically prove the impact of comparison errors on quicksort and heapsort in Sections 3 and 4. Section 5 presents our practical error correction algorithms and their complexity analysis. The proposed algorithms are extended to parallel environment in Section 6. Our proposed FT-iSort is evaluated in Section 7, and we then conclude our paper.

2 SORTING ALGORITHMS AND TERMINOLOGY

2.1 Introsort

Introsort is a hybrid sorting algorithm built upon quicksort and heapsort; introsort achieves the best average performance of quicksort ($O(n \log n)$) but avoids the poor performance of quicksort in the worst case ($O(n^2)$) by switching to heapsort (a guaranteed $O(n \log n)$). We review these two building blocks before presenting the introsort algorithm.

Quicksort. We use the basic quicksort algorithm presented by Lomuto [7] in our discussion for simplicity. It should be noted that our analysis is valid for quicksort variations [5, 8] as well; improvements to aspects such as pivot selection do not impact our analysis. Quicksort is a divide and conquer algorithm that sorts a given n -element array, $a[]$. Quicksort chooses the first element of the given array as the *pivot*, around which the array is partitioned.

Quicksort chooses $a[0]$ as the pivot to partition the original array into two sub-arrays, $[a[1], a[2], a[3], \dots, a[j]]$, with each element strictly less than $a[0]$, and $[a[j+1], a[j+2], a[j+3], \dots, a[n-1]]$, with each element no less than $a[0]$. The pivot $a[0]$ is then swapped to a medial position between the two sub-arrays. These two steps are then recursively applied to the two sub-arrays, until each sub-array contains at most one element. After each partitioning, the

pivot rests in what will be its final position in the sorted array, and thus it remains untouched in later recursive sorting.

Heapsort. Heapsort is built on the idea of a *max-heap*: a complete binary tree where the value of each non-leaf node is greater than or equal to the value of any first-generation child. Heapsort represents a max-heap as an array, with the root as the first element of the array; the first element is therefore maximal.

Heapsort consists of two stages: max-heap creation and sorting using the max-heap. In the first stage, the given n -element array $a[]$ is adjusted to form a max-heap. The second stage is an iterative process of excluding the largest element and adjusting the max-heap for the remaining elements. In the first iteration, the largest value, $a[0]$, is excluded by swapping it with $a[n-1]$, and then adjusting the first $n-1$ elements to form a new max-heap; the largest element is then found at the end of the array. The second iteration repeats the process on the $(n-1)$ -element max-heap formed from the remaining $(n-1)$ elements of the array, swapping out the second largest value $a[0]$ with $a[n-2]$ and adjusting a new $(n-2)$ -element max-heap, and so on for the remaining iterations. The array is sorted in ascending order at the end of this process.

Algorithm 1 *std::sort(a[], n)*

```

1: if  $n \leq 1$  then return
2: end if
3: introsort_loop(a[],  $2 \log n$ )
4: insertion_sort(a[], n)
5: return

Function: introsort_loop(a[], depth_limit):
while  $a[].current\_partition.size > 16$  do
  if  $depth\_limit = 0$  then
    heap_sort(a[].current_partition)
    return
  end if
   $depth\_limit \leftarrow depth\_limit - 1$ 
   $P \leftarrow pivot(a[].current\_partition)$ 
   $[L[], R[]] \leftarrow partition(a[].current\_partition, P)$ 
  introsort_loop(R[],  $depth\_limit$ )
   $a[].current\_partition \leftarrow L[]$ 
end while

```

Introsort. The GNU Standard C++ Library is internationally recognized, and introsort is the GNU C++ default sorting routine. We base our discussion and implementation of introsort on the version found in this library. Our fault tolerant introsort is thus readily available to a large user base.

GNU C++ users can invoke introsort by calling `std::sort`. Algorithm 1 shows the code skeleton of `std::sort` and the first major phase, `introsort_loop()`. While in `introsort_loop()`, quicksort is used to recursively partition the given array either until all partitions are 16 elements or less, or until the recursion depth is deeper than the pre-defined depth limit of $2 \log n$, where n is the size of the array. When the recursive partitioning stops due to hitting the depth limit, `introsort_loop()` uses heapsort to sort the remaining partitions with more than 16 elements, avoiding the $O(n^2)$ worst-case complexity of quicksort.

Figure 1: Example introsort execution; phase 1 comprises quicksort and heapsort; phase 2 executes insertion sort.

Once the first phase completes, all partitions with 16 elements or less remain unsorted. These small, unsorted partitions are called *leaf partitions*. After `introsort_loop()` completes, the second major phase, `insertion_sort()`, uses an insertion sort on the entire array; the leaf partitions are of an optimal size for insertion sort to perform well, and a completely-sorted array is returned at the end.

Figure 1 illustrates both phases of the introsort sorting process with a recursion depth limit of 4.

2.2 Target Errors

We assume soft errors can lead to two kinds of errors: comparison errors and memory errors.

2.2.1 Comparison error. Paired comparison is the operation of comparing two values, resulting in a true or false bool value output; it is the most common basic computation operation of introsort. A *comparison error* occurs when soft errors corrupt the result of a comparison instruction either by altering the instruction operation code (opcode) or the register operands [2]. For example, a comparison instruction $a < b$ may return an incorrect result if the operation is changed to another operation, such as $a + b$; it may also return an incorrect result if the register file storing the real result of $a < b$ is corrupted, e.g., true (1) to false (0), or false (0) to true (1).

Most algorithm-based fault tolerance (ABFT) studies target computation errors only related to the *key data structures* of the algorithm under consideration [39, 41], ignoring the computation errors involving control-flow data. For example, the ABFT strategy for matrix multiplication algorithms focuses on the computations errors in the matrix multiplication only, without considering computation errors in the calculation of iteration counts. In this paper, we likewise focus on computation errors (comparison errors in the context of sorting) arising from the comparisons of elements in the given array.

We introduce the definition of *adjacent inversion pairs*. An adjacent inversion pair is a pair of adjacent array elements ($a[i]$, $a[i+1]$) such that $a[i] > a[i+1]$; this term is frequently used in our analysis of comparison error impact on introsort output, as well as error-correcting overhead.

For simplicity of presentation, we denote the size of the given array as n , and the total number of comparison errors as e . In our paper, we follow the conventions of previous research [30, 32] and set $e \leq \log n$; this reflects the rarity of real world comparison errors.

2.2.2 Memory error. A *memory error* occurs when a soft error corrupts a value in memory with a bit flip. This paper considers

memory errors inside the given array, that is, the key data structure of the sorting algorithm. We focus on memory errors which cannot be protected by error correction code (ECC).

3 COMPARISON ERRORS IN QUICKSORT

We introduce terms necessary for our analysis in the sequel. *Unsortedness* is defined as the minimum number of removals required to produce a smaller but fully sorted remainder sequence [16]. The smaller the unsortedness of a sequence, the better the sequence is sorted; naturally, the unsortedness of a fully sorted sequence is 0. For example, we have a sequence: [1, 2, 5, 8, 7, 6]. If we remove {8, 7, 6}, the remaining sequence [1, 2, 5] is sorted and the number of removals is 3. However, 3 is not the unsortedness of the sequence, because it is not the minimum number of removals required for a sorted remainder sequence. If we remove {8, 6}, the remaining sequence [1, 2, 5, 7] is sorted; thus, the unsortedness is 2.

Notice all comparisons in a partitioning involve a pivot. If a comparison error occurs when comparing a given pivot O and an element X , we say O is an *error-related pivot* and X is an *erroneous element*. If comparison errors occur in some level, we say the recursion of this level is an *erroneous recursion*.

Figure 2: Illustration of the proof of Theorem 1 with two erroneous recursion levels: the removal procedure is shown from bottom to top, with e elements removed from $a_F[]$ to obtain a sorted remainder array.

With these definitions in mind, we present the following theorem and proof:

Theorem 1: If there are e comparison errors during execution of Lomuto's quicksort algorithm, then the unsortedness of the final output is at most e .

We provide illustrations in Figure 2 to foster easier understanding of our proof. Figure 2 illustrates the procedure to remove all erroneous elements in the erroneous output to obtain a sorted remainder array, which Theorem 1 measures as unsortedness. Keeping this intuitive description in mind, we present the proof.

Proof: Denote erroneous recursions by $i_1, i_2, i_3, \dots, i_m$, where i_m is the deepest-level erroneous recursion. Suppose recursion i_k has e_k comparison errors, forming the erroneous element sequence $X_k = [x_k^{t_1}, x_k^{t_2}, \dots, x_k^{t_{e_k}}]$. Here k ranges from 1 to m , and $\sum_{k=1}^m e_k = e$. Since i_m is the deepest-level erroneous recursion, there are no further comparison errors after recursion level i_m , and each partition newly generated at i_m will be sorted intra-partition after the quicksort completes.

Denote the erroneous quicksort final output array by $a_F[]$. To show the unsortedness of $a_F[]$ is at most e , we need to demonstrate it is possible to remove e elements and obtain a fully sorted remainder array from $a_F[]$. Denote all error-related pivots in i_m by $O_1, O_2, \dots, O_f$, in order of occurrence from left to right. Denote the pairs of partitions in the left and right of each error-related pivot by $a_1^l, a_1^r, a_2^l, a_2^r, \dots, a_f^l, a_f^r$. Each partition a_i^l, a_i^r is sorted in $a_F[]$, since i_m is the deepest-level erroneous recursion, but each of the upper level partitions a_i^l, a_i^r may be unsorted. A comparison error may place an element larger than or equal to the pivot O_i into a_i^l when it should be placed in a_i^r , or it may place an element smaller than or equal to pivot O_i into a_i^r when it should be placed in a_i^l . Either case will result in an unsorted partition a_i^l, a_i^r . Now notice that all such erroneous elements in i_m are in, and exactly form, the sequence $X_m = [x_m^{t_1}, x_m^{t_2}, \dots, x_m^{t_{e_m}}]$. We remove these elements and denote the updated partitions by $a_i^{l'}$ and $a_i^{r'}$. Observe that each of the upper-level partitions $a_i^{l'}, a_i^{r'}$ is sorted: because we have removed all erroneous elements at this level, $a_i^{l'}$ and $a_i^{r'}$ are sorted and all elements in $a_i^{l'}$ are smaller than O_i , with all elements in $a_i^{r'}$ greater than or equal to O_i .

Once all erroneous elements in X_m have been removed, each partition that was generated at recursion i_{m-1} will be sorted in the array sequence $a_F[] \setminus \{X_m\}$. Continuing analysis as was done for i_m , we remove erroneous elements in $X_{m-1}, X_{m-2}, \dots, X_1$. At the end, we achieve a sorted remainder array sequence $a_F[] \setminus \cup_{i=1}^m \{X_i\}$. We see that $\sum_{k=1}^m e_k = e$, and we therefore removed e elements in total to obtain the sorted remainder array; that is, the unsortedness of $a_F[]$ is at most e . $\square$

The bound on unsortedness from this theorem allows us to create an efficient algorithm to correct the output array of an erroneous quicksort, which is detailed in Algorithm 2 in Section 5.1.3 below.

4 COMPARISON ERRORS IN HEAPSORT

We begin our analysis of heapsort with the following definitions. A *heap edge* is a virtual edge connecting a non-leaf node and one of its children, denoted by (v_1, v_2) : v_1 is the non-leaf node in the heap and v_2 is the child of v_1 . We say this heap edge is *sufficient* if the value of v_1 is no less than the value of v_2 ; we say it is *insufficient* otherwise. We use the number of insufficient heap edges in the heap to measure how close it is to a max-heap. The smaller the total number of insufficient heap edges, the closer the heap is to a max-heap, i.e., the heap is a max-heap if and only if there are no insufficient heap edges.

We define the *runs* of an array, $a[]$, as the minimum number of sorted segments into which $a[]$ can be divided [32]. The smaller the number, the more well-sorted the array is; $a[]$ is sorted if and only if the runs is 1. We particularly note that the runs of an array is the number of occurrences of $a[i] > a[i+1]$ where we partition $a[]$ into sorted sub-arrays.

In heapsort, both the two stages, max-heap creation and sorting using the max-heap, iteratively call `adjust_heap()`, which adjusts the new value in the root node of the heap to preserve its structure as a max-heap. It does this by sifting values up or down the heap as needed. We present a simple example here, illustrated in Figure 3. Suppose the value 3 on node V_0 needs to be adjusted to preserve the

Figure 3: Visual demonstration of `adjust_heap()`. The dashed lines represent sifting paths. The bolded, red 3 is sifted down and then up.

max-heap property. In the sifting down phase, 3 swaps down the heap, exchanging places with the value of the greatest child node until it reaches a leaf node—in this case V_7 . This gives us the swap sequence (3, 5) (3, 4) (3, 2) along the sift-down path V_0, V_1, V_3, V_7 . In the sifting up phase, the value 3 begins on a leaf node; however, it may have a greater value than the parent node due to the sift-down phase. The value 3 therefore keeps swapping with the value of higher parent nodes, until it reaches a node where the next higher node has equal or greater value. In this example, the limiting node V_3 is found after only one swap; the swap sequence is (3, 2) along the sift-up path V_7, V_3 .

4.1 From Errors to Insufficient Heap Edges

We present two lemmas needed for the derivation of our bound on the runs of erroneous heapsort output. First, we discuss the behavior of heap edges during heapsort.

Lemma 1: An error free `adjust_heap()` will not generate more insufficient heap edges upon completion.

We first illustrate the main idea behind the proof in Figure 4. Intuitively, only the edges connected to nodes that experience a change in value can possibly change sufficiency. Thus we do not have to consider edges connected only to nodes without updated values. In Figure 4, the left side shows all heap edges that could undergo a change in sufficiency during the execution of `adjust_heap()`; dashed edges are the sift path. We exclude all other nodes in the heap because they remain unchanged during execution. In the right tree, only the red nodes have updated values; the red edges are corresponding heap edges that may experience a change in sufficiency.

Figure 4: Illustration of key ideas behind Lemmas 1 and 2.

Proof: Consider the procedure of one complete `adjust_heap()`, starting from time t_0 . The previous `adjust_heap()` has just finished and a new value a is on the root node and needs to be sifted down. At time t_1 , the sift-up process following sift-down has completed. In an error-free `adjust_heap()`, the following steps take place: (1) The sift-down phase finds a path p from root node to a leaf node; (2) The sift-up phase travels along the inverse of p , from leaf to root, to find the first node v_m which contains a value no less than a ; (3) All nodes that are above v_m along path p , including v_m ,

shift their values to the next higher node and the value a is placed in v_m . Denote p by $\{(v_0, v_1, v_2, \dots, v_l)\}$, where each v_i is the parent node of v_{i+1} and v_l is the leaf node; we represent the other child of v_i by v'_{i+1} . We define $V(v_i)$ to return the value of node v_i , and introduce the subscripts $V_0(v_i)$ and $V_1(v_i)$ to indicate the value of node v_i at time t_0 and time t_1 , respectively. In step (3), only the values on nodes $v_0, v_1, v_2, \dots, v_m$ update: the equation is given by $V_1(v_i) = V_0(v_{i+1})$ for all $i \in [0, m-1]$. As part of the same process, $V_1(v_m) = a$.

We now analyze the heap edges formed by nodes with values that have been updated. If $i \in [0, m-1]$, the heap edges (v_i, v'_{i+1}) are sufficient at t_1 : since the sift-down path always finds the greater child, we have $V_1(v_i) = V_0(v_{i+1}) \geq V_0(v'_{i+1}) = V_1(v'_{i+1})$. If $i \in [0, m-2]$, the heap edge (v_i, v_{i+1}) is insufficient at t_1 if and only if (v_{i+1}, v_{i+2}) was insufficient at t_0 . If $i = m-1$, (v_{m-1}, v_m) is always sufficient as sift-up ensures $V_1(v_{m-1}) = V_0(v_m) \geq a = V_1(v_m)$. If $i = m$, heap edges (v_m, v_{m+1}) and (v_m, v'_{m+1}) are sufficient: sift-up ensures $V_1(v_m) = a > V_0(v_{m+1}) = V_1(v_{m+1})$ and sift down ensures $V_1(v_{m+1}) = V_0(v_{m+1}) > V_0(v'_{m+1}) = V_1(v'_{m+1})$. For $i \in [m+1, l-1]$, (v_i, v_{i+1}) and (v_i, v'_{i+1}) are insufficient at t_1 if and only if they were insufficient at t_0 since these values did not update.

This analysis accounts for all heap edges involved with changed values in the `adjust_heap()` update. All other heap edges in the max-heap remain unchanged. We find each insufficient heap edge at t_1 must result from an insufficient heap edge at t_0 . $\square$

We have established that existing insufficient heap edges will not propagate in the absence of fresh errors; we now address how new errors will generate insufficient heap edges. We present the following lemma:

Lemma 2: Each comparison error that occurs during one execution of `adjust_heap()` will generate at most two additional insufficient heap edges at the completion of the erroneous `adjust_heap()`.

Proof: Using the previous notation, we consider an erroneous `adjust_heap()`, classifying comparison errors of three types: (1) Errors in sift-down; (2) Errors in sift up that judge $a \leq V_0(v_i)$ as true when in fact $a > V_0(v_i)$; (3) Errors in sift-up that judge $a \leq V_0(v_i)$ as false when in fact $a \leq V_0(v_i)$. If $i \in [0, m-1]$, the heap edge (v_i, v'_{i+1}) is insufficient at time t_1 only if a Type 1 error occurred between $V_0(v_{i+1})$ and $V_0(v'_{i+1})$; in this case $V_1(v_i) = V_0(v_{i+1}) \leq V_0(v'_{i+1}) = V_1(v'_{i+1})$, and the edge is insufficient. If $i \in [0, m-2]$, the heap edge (v_i, v_{i+1}) is insufficient at t_1 if and only if (v_{i+1}, v_{i+2}) was insufficient at t_0 . If $i \in [m+1, l-1]$, (v_i, v_{i+1}) and (v_i, v'_{i+1}) are insufficient at t_1 if and only if they were insufficient at t_0 since none of these values updated.

Heap edges involving v_m are more complicated. The heap edges we need to consider are (v_{m-1}, v_m) , (v_m, v_{m+1}) and (v_m, v'_{m+1}) . We notice $a \leq V_0(v_m)$ is the last comparison performed in sifting up that returned true. (v_{m-1}, v_m) is therefore insufficient only if the last comparison during sift-up was a Type 2 error, since it will only be insufficient if $V_1(v_{m-1}) = V_0(v_m) \geq a = V_1(v_m)$ does not hold. (v_m, v_{m+1}) is insufficient only if a Type 3 error occurred, since otherwise $V_1(v_m) = a > V_1(v_{m+1})$. A Type 3 error may also cause insufficiency in edge (v_m, v'_{m+1}) since the relationship between the values on v_m and v'_{m+1} cannot be determined at time t_1 .

We see that Type 3 errors can only happen prior to Type 2 errors due to the order of the sift-up procedure. We have enumerated all possible errors. We see each Type 1 or Type 2 error generates at

most one additional insufficient heap edge; each Type 3 error generates at most two additional insufficient heap edges. We conclude each comparison error during one execution of `adjust_heap()` generates at most two additional insufficient heap edges. $\square$

We derive the following corollary from Lemmas 1 and 2.

Corollary 1: At most $2e$ insufficient heap edges are generated during the execution of a heapsort where e comparison errors occur.

4.2 From Insufficient Heap Edges to Runs

Consider the `sort_heap()` procedure on two adjacent popped elements a and b ; that is, a and b are adjacent when the procedure has finished and popped them out of the heap into their final positions in the array, resulting in the partial output $\dots ba \dots$. An element is popped only when it occupies the root node v_0 after `adjust_heap()` has completed. Set the time when element a occupies the root node as time t_0 , and the time when b occupies the root node as time t_1 . After a is popped, the next popped element will be one of the following two cases: (Case 1) the value in one of the children of v_0 at t_0 , or (Case 2) the value in the last node v_x at t_0 , since it will be moved to v_0 after time t_0 . The comparisons in the interval $(t_0, t_1]$ can be either error-free (Case f), or erroneous (Case e). There are 4 cases in total, which we will denote by $case_{xy}$; for example, $case_{1f}$ is Case 1 occurring with Case f, that is, b was in a child node of v_0 at t_0 and comparisons in $(t_0, t_1]$ were error free.

Theorem 2: Given at most e comparison errors, the runs of the output of an erroneous heapsort will be at most $2e + 1$ if $case_{2f}$ does not occur.

Proof: To partition an array into sorted segments, adjacent inversion pairs must be separated in the middle. Let ba be an adjacent inversion pair in the output of an erroneous heapsort, thus, $b > a$. Using the same notation as above, we define the child node of v_0 at t_0 by v_0^l (and symmetrically, v_0^r). Below we prove a popped inversion pair ba must remove at least one insufficient heap unit or remove the negative impact of at least one comparison error (claim 1).

In $case_{1f}$, using the same analysis as in the proof of Lemma 1, we see the insufficient heap edge (v_0, v_0^l) for $b > a$ at t_0 was removed and a new sufficient heap edge (v_{m-1}, v_m) was generated at t_1 . Other heap edges either change their location or stay still. Thus, the insufficient heap edges decrease by 1 and claim 1 holds.

In $case_{1e}$, the insufficient heap edge (v_0, v_0^l) for $b > a$ at t_0 was removed and at most $2e'$ more insufficient ones were generated if there are at most e' errors (Lemma 2); claim 1 still holds true, as one insufficient heap edge was removed.

In $case_{2e}$, since the value on v_x at t_0 was placed on v_0 after t_0 and was sifted down and then sifted up to v_0 again, all values on all nodes excepting v_0 did not change. $b > a$ means edges (v_0, v_0^l) and (v_0, v_0^r) cannot change from sufficient to insufficient. Though the number of insufficient heap edges may not decrease, at least one error happened and ba removes the negative impact of that error.

In conclusion, a popped adjacent inversion pair must either remove at least one existing insufficient heap edge generated by previous errors, or it must remove the negative impact of at least one error that occurred within $[t_0, t_1]$. According to Corollary 1, the number of popped adjacent inversion pairs will be at most $2e$. Partitioning at the middle of each adjacent inversion pair will give us at most $2e + 1$ sorted segments, which leads to Theorem 2. $\square$

Table 1: Notation for overhead analysis

symbol	meaning
e	Total comparison errors in <i>introsort</i>
n	Input array length
$C_1 n \log n$	Total comparisons in phase 1 of <i>introsort</i>
$C_2 n$	Total comparisons in phase 2 of <i>introsort</i>

$Case_{2f}$ is experimentally shown to not impact the conclusion in Section 7.2, and a faster error-correcting algorithm for heapsort output results from Theorem 2.

E-sorted-merge. Since the result of an erroneous heapsort has at most $2e + 1$ sorted segments, we can apply an efficient error-correcting algorithm using partitioning and merge. We name this algorithm as *e-sorted-merge*. The procedure is:

- (1) Scan the output and record inversion pair locations;
- (2) If no such location is found, return; else go to step (3);
- (3) Merge the sorted segments partitioned by the above locations recursively until a single sorted array is generated.

This algorithm has $\log(2e + 1)$ recursion levels. Its complexity is $n \log(2e + 1)$, which is much more efficient than re-executing heapsort at a cost of $n \log n$.

5 PRACTICABLE STRATEGIES FOR INTROSORT PROTECTION

5.1 Protection against Comparison Errors

The only protection strategy for comparison errors currently in practice is modular redundancy, a general fault tolerance strategy. Direct application of this strategy incurs significantly high overhead for *introsort*. To reduce protection overhead, we propose a strategy based on our prior theoretical proofs.

We introduce the notation shown in Table 1 for clarity in our overhead analysis. It is known [29] that the time complexity of phase 1 of *introsort* is $O(n \log n)$, with comparison as the basic operation. The total number of comparisons in this phase is $C_1 n \log n$, where C_1 is a constant. The total number of comparisons in phase 2 is $C_2 n$, where C_2 is a constant; we show the derivation. Recall the goal of phase 2 is to use insertion sort to sort all leaf partitions, the sizes of which are bounded by 16. For one leaf partition having d elements, $d \leq 16$, the time complexity for insertion sort on the partition is $d^2/2$. The overall complexity to sort all leaf partitions is bounded by $d^2/2 * n/d = dn/2$. We simplify the overall complexity as $C_2 n$ with C_2 bounded by $d/2$; as d is itself bounded by 16, $C_2 \leq 8$.

5.1.1 Naïve Protection. There is currently no protection strategy for the tolerance of comparison errors specifically designed for *introsort*. The most straightforward naïve strategy is to *protect all comparison errors* with triple modular redundancy. Triple modular redundancy (TMR) is commonly used to ensure correct instruction execution and the result generated by an execution with TMR protection is treated as reliable [33]. The basic idea of TMR consists of two parts: *instruction duplication* and *majority voting*. Instruction duplication safeguards *introsort* by ensuring each protected instruction is executed up to three times; majority voting then returns the result agreed by at least two executions as the correct result. To protect the comparison instruction, TMR works as outlined in the following pseudo-code:

```

1  bool ret1 = compare();
2  bool ret2 = compare();
3  if (ret1 == ret2) return ret1;
4  else return compare();

```

Notice that the third execution is needed only when the results of the first two executions do not match; the result of the third execution is assumed to be the correct result based on voting. In the results reported in our paper, TMR overhead was simulated by inserting tripled assembly codes for each focused comparison instruction in the given program.

Overhead. TMR triples comparison instructions and even quadruples when comparison error happens (line 3 also contains a comparison instruction). Thus, the comparison overhead of TMR is at least 200%. Precisely, the comparison overhead (O_c) and data movement overhead (O_d) are respectively:

$$O_c = 2(C_1 n \log n + C_2 n), \text{ and } O_d = 0. \quad (1)$$

5.1.2 Aggressive Protection. The 200% overhead of naïve protection is both unnecessary and far too costly. Only $e < \log n$ erroneous elements will be unsorted even if quicksort is not protected (Theorem 1). These e unsorted elements can be efficiently corrected by the insertion sort following the quicksort, significantly reducing the overhead. We introduce *aggressive protection*, which is outlined as follows:

- Apply TMR-protected *e-sorted-merge* to the output of heap-sort in phase 1.
- Apply TMR to insertion sort in phase 2.

All errors occurring within quicksort in phase 1 are left unattended and are corrected in phase 2 by a TMR-protected insertion sort. This strategy avoids the unnecessary overhead of TMR-protection for quicksort and heapsort.

Overhead. Our analysis reveals that, in the worst case scenario, the overhead of our aggressive protection strategy is superior to the average overhead of naïve protection. There are two cases that compete for worst case scenario: either (1) all e errors occur within quicksort, or (2) all e errors occur within heapsort. In case 1, TMR for insertion sort incurs $2C_2 n$ additional comparisons for the sorting of $n - e$ error-free elements; it incurs en additional data movements as well as $3en$ extra comparisons to correct the e erroneous elements by reordering them to their proper places. The total comparison overhead in this scenario is $O_{c1} = 2C_2 n + 3en$, and the total data movement overhead is $O_{d1} = en$. In case 2, the overhead is generated by *e-sorted-merge* and the TMR protection for insertion sort. The overhead in *e-sorted-merge* originates from scanning the output of heapsort and a merging process; both are protected by TMR. The scanning costs $3n$ comparisons, and the merging of $2e + 1$ sorted segments costs $3n \log(2e + 1)$ comparisons and $n \log(2e + 1)$ data movements. The TMR-protected insertion sort cost is $3n$ comparisons and no data movements, because the array is already sorted by heapsort and *e-sorted-merge*. The total comparison overhead is $O_{c2} = 3n[\log(2e + 1) + 2]$, and the data movement overhead is $O_{d2} = n \log(2e + 1)$. In summary, the worst case scenario has the following comparison overhead (O_c) and data movement overhead (O_d):

$$O_c = \max(O_{c1}, O_{c2}) = \max\{2C_2 n + 3en, 3n[\log(2e + 1) + 2]\}, \quad (2)$$

$$O_d = \max(O_{d1}, O_{d2}) = \max\{en, n \log(2e + 1)\}. \quad (3)$$

The total overhead comprising both comparison and data movement is less than the overhead of naïve protection since $e < \log n$.

Algorithm 2 Improved_nearly_sorting($a[]$)

```

1: find all  $a[i] > a[i+1]$  in  $a[]$ 
2: for all pairs of  $a[i], a[i+1]$  do
3:   erroneous_ele[].add( $a[i], a[i+1]$ )
4:   continue add ( $a[i-1], a[i+1+1]$ ) until non-inversion pair
5:   gap_length[].add(total number of removals)
6: end for
7: for remaining element blocks in  $a[]$  do
8:   move block  $(i+1)$  left by gap_length[i] cells
9: end for
10: erroneous_ele[].sort()
11: for element in erroneous_ele[] do
12:   binary search to find inserting locations
13: end for
14: for blocks divided by inserting locations in  $a[]$  do
15:   move block  $i$  right by erroneous_ele[].size -  $i$ 
16:   insert erroneous_ele[i]
17: end for

```

5.1.3 Conservative Protection. In the worst case scenario, the distance between the incorrect position and correct position of an erroneous element is almost n ; in this case aggressive protection requires too many comparisons ($3en$) and data movements (en). This is due to the insertion sort, and is aggravated by the TMR protection. We propose *conservative protection* to significantly reduce this overhead, applying the following changes to introsort:

- Apply TMR-protected *e-sorted-merge* to the output of heap-sort in phase 1.
- Advance the insertion sort from phase 2 to phase 1; that is, remove the insertion sort in phase 2 and apply insertion sort to each leaf partition in phase 1.
- Apply TMR-protected Algorithm 2 in phase 2, which efficiently corrects any comparison errors in phase 1.

The significant overhead reduction of conservative protection as compared to aggressive protection is a result of two key strategies. First, the advancement of insertion sort avoids non-negligible overhead in the worst case scenario by reducing the maximal distance between the current position and the correct position of an erroneous element from n to 16, and second, Algorithm 2 reduces the overhead of correcting the output array of erroneous quicksort by making use of the fact that the output array is nearly sorted. Our algorithm for correcting nearly-sorted output improves on the algorithm proposed by Cook et al. [16].

Improved sorting algorithm specifically for a nearly-sorted array. Cook et al. [16] propose a sorting algorithm for a nearly sorted array. The algorithm, which can be used in conjunction with TMR protection, functions as follows: remove all adjacent inversion pairs and gather them together, sort these removed elements, and then merge the sorted elements back into the remaining array. Recall that a traditional merge will take n comparisons to merge two sorted arrays with combined size n ; this is then multiplied by 3 to account for TMR protection, resulting in an overhead of $3n$ comparisons for

Figure 5: Illustration of Algorithm 2

the merge alone. We find that such a high overhead is unnecessary since $e < \log n$, that is, the number of erroneous elements is far less than the size of the array. We therefore propose a more efficient last step to correct the nearly-sorted array. We use the following two techniques: first, we use binary search to find the locations where the removed elements should be inserted, which reduces the $3n$ comparison overhead to $3e \log n$, and second, we insert the removed elements in decreasing order into the remaining array, which incurs at most $2n$ data movements.

The proposed algorithm is detailed in Algorithm 2. It consists of the following five small steps: (1) Remove adjacent inversion pairs so that the remaining array is sorted (lines 1-6); (2) Shift the remaining elements to the left so that all empty spaces created by the removed elements are filled (lines 7-9); (3) Sort the small array containing all removed elements (line 10); (4) Find the inserting locations using binary search (lines 11-13); (5) Insert elements back into the array in their correct locations in decreasing order (line 14-17), that is, larger elements should be inserted before smaller elements, ensuring each element moves only once.

Figure 5 illustrates how Algorithm 2 operates, using the example nearly-sorted array $[0, 1, 3, 4, 5, 2, 6, 7, 8, 9]$. The inversion pair $[5, 2]$ is first removed from the array, and the array is adjusted by shifting elements on the right of the removed elements towards the left to fill the empty places created by the removed elements. Note the remaining array now becomes sorted from index 0 to 7. The removed element array $[5, 2]$ is sorted, giving $[2, 5]$. Finally, the removed elements are inserted back into the remaining array in decreasing order.

Overhead. Nearly analogous to the worst case for aggressive protection, the worst case scenario for conservative protection is one of two cases: (1) all e errors occur within quicksort and insertion sort, or (2) all e errors occur within heapsort. In case 1, we know that e errors lead to at most e unsortedness in the quicksort output. Each time an adjacent inversion pair is removed in Algorithm 2, the unsortedness of the remainder array is decreased by at least 1. Thus Algorithm 2 will remove at most e adjacent inversion pairs (i.e., $2e$ elements). For insertion sort, Algorithm 2 is applied to each leaf partition, and each partition has a maximum bounded length of 16; each error will cause up to all 16 elements to be removed. The maximum number of elements to be removed by Algorithm 2 is therefore $16e$. In Algorithm 2, factoring in TMR protection, step 1 takes $3n$ comparisons, step 2 takes n data movements, step 3 takes $3 \cdot 16e \cdot \log(16e)$ comparisons, step 4 takes $3 \cdot 16e \cdot \log n$ comparisons, and step 5 takes $n + 16e$ data movements. The total comparison overhead is $O_{c1} = 3(n + 16e \log n + 48e \log e + 192e)$; the data movement overhead is $O_{d1} = 2(n + 16e)$. In case 2, the overhead comes

from TMR-protected *e*-sorted-merge and Algorithm 2 with TMR protection. *e*-sorted-merge takes $3n \log(2e + 1)$ comparisons and $n \log(2e + 1)$ data movements. Algorithm 2 takes $3n$ comparisons and no data movements since the array is already sorted by heap-sort and *e*-sorted-merge. The total comparison overhead and data movement overhead are, respectively, $O_{c2} = 3n[\log(2e + 1) + 2]$ and $O_{d2} = n \log(2e + 1)$. Therefore, the comparison overhead (O_c) and data movement overhead (O_d) in the worst case scenario are respectively (after eliminating negligible terms):

$$O_c = \max(O_{c1}, O_{c2}) = \max\{3(n + 16e \log n), 3n[\log(2e + 1) + 2]\}, \quad (4)$$

$$O_d = \max(O_{d1}, O_{d2}) = \max\{2(n + 16e), n \log(2e + 1)\}. \quad (5)$$

The total overhead, $O_c + O_d$, is still less than the overhead of naïve protection considering $e < \log n$.

5.1.4 Summary of Comparison Error Mitigation Strategies. In Section 5.1, we present aggressive and conservative protection strategies for introsort based on Theorem 1 and 2. In the worst case scenario, under the error bound $e \leq \log n$, both strategies incur less overhead than baseline naïve protection. The dominant overhead of aggressive protection is $(2C_2n + 3en)$ for comparisons and (en) for data movements; its performance degrades significantly with an increasing number of errors. Thus we propose conservative protection to reduce the overhead to $(3n + 48e \log n)$ for comparisons and $(2n)$ for data movements.

5.2 Protection against Memory Errors

5.2.1 Pure Re-Execution (PR). The most straightforward way to tolerate memory errors in introsort is re-execution with reloaded input, i.e., TMR at the application level. In this method, double executions are performed and a memory error is found to have occurred if the results differ; otherwise, no memory error is found. If a memory error occurs, another execution is performed and the result agreed upon by majority voting is returned. When no memory error occurs, it returns either result since the two executions generate the same output.

5.2.2 XOR-based Re-Execution (XR). The overhead of naïve re-execution protection is significantly high. To reduce overhead, we introduce *exclusive or based re-computation*. XR detects the occurrence of memory errors based on XOR values and re-executes if memory errors are detected. Sanders et al. proposed a modular operation-based memory error detection for integers in big data applications [24]. For the problem of sorting, XOR will work to detect memory errors and it is more efficient and simpler than modular operations. As the summation of the entire array does not change without memory errors, checksum can also be used to detect memory errors. However, checksum has round-off error with floating point numbers. In contrast, XOR operation has no round-off error plus XOR operation is faster than addition. To let XOR operation work for floating point numbers, we just need to treat the bits of a floating point number as an unsigned integer.

Memory error detection using XOR. During the sorting process, the XOR results ($xor = a[0] \oplus a[1] \oplus \dots \oplus a[n - 1]$) over the entire array should remain unchanged since the array contains the same elements; this computed XOR result is known as a XOR. In XR, three XORs are calculated over the array to detect memory errors,

Figure 6: Illustration of XOR calculation schedule.

roughly evenly distributed during the sorting process. Memory errors are detected if the XORs vary; otherwise, we assume no memory error occurs. To ensure the correctness of XOR calculation and XOR memory, we can use TMR for XOR calculation and maintain another copy of XOR results so that we can verify the copies are the same before detect memory error in input data. We notice that if the partitions at recursion level 1 have similar lengths, quicksort and/or heapsort will spend a similar length of time on each partition. This is the inspiration for our scheduling of the three XORs; the calculations are approximately evenly scheduled as follows which is illustrated in Figure 6.

- The first XOR is calculated at the onset of sorting.
- The second XOR is calculated immediately after quicksort and/or heapsort finish sorting the first partition at level 1.
- Once the entire sorting process completes, a third and final XOR is calculated and validated.

Overhead. Suppose the following: The first execution is erroneous, the re-execution is error-free, the second XOR takes place at the temporal midpoint of the introsort procedure, the first memory error occurs in either half of the execution with equal probability, and the ratio of the time cost of one XOR calculation to the time cost of the original introsort execution is c_o . Clearly, the probability of the first error occurring before the second calculation is $\frac{1}{2}$, and the same for occurring after. When the first error arises before the second XOR calculation, XOR calculations in this erroneous execution cost $2c_o$, the erroneous execution (which does not continue past the second XOR) costs $\frac{1}{2}$, and the XOR calculations in the re-execution cost $3c_o$. A similar analysis can be performed when the first error arises after the second XOR calculation. The total expected overhead is $\frac{1}{2}(2c_o + \frac{1}{2} + 3c_o) + \frac{1}{2}(3c_o + 1 + 3c_o)$ which reduces to

$$O_{ex} = 75\% + 5.5c_o \quad (6)$$

If no memory errors occur, the overhead is clearly just $3c_o$.

6 EXTENSION TO PARALLEL SORTING

Our fault tolerant strategy also applies to parallel sorting algorithms based on *introsort*. To the best of our knowledge, HykSort [34] is the most well-known parallel sorting.

6.1 HykSort

Given a global array, the size of which far exceeds the memory limit of one computing node, HykSort performs parallel sorting using introsort, following the procedure below:

- (1) Each process loads a part of the array; that is, the global array is partitioned into multiple local arrays, each local array kept by one process.
- (2) Each process sorts the allotted local array using introsort.
- (3) Each process partitions the assigned local array into k partitions based on $k - 1$ global pivots.

- (4) Processes exchange partitions, ensuring processes with lower ranks receive partitions with lesser elements and processes with higher ranks receive partitions with greater elements.
- (5) Each process merges the k sorted partitions it received.
- (6) The sorting completes if the following requirements are met: (a) the local array of each process is well-sorted, and (b) $e_i < e_j$, where e_i denotes any element from process i , e_j denotes any element from process j , and $0 \leq i < j < p$ where p is the total number of processes; otherwise, it goes back to Step 3.

6.2 Fault Tolerant HykSort

6.2.1 Protection for Comparisons. 90% coverage. Our fault tolerant introsort is constructed to be naturally contained in the GNU Standard C++ Library. Without any code changes to HykSort, 90% of comparisons of HykSort can be achieved by linking to the library as before. We denote the global array length, the number of processes, and the number of partitions respectively as N , p and k . Local array length is given by $n = \frac{N}{p}$. The number of comparisons in local sorting, partitioning, and merging are respectively $n \log n$ (lower bound), $k \log n$ and $n \frac{\log p}{\log k}$ [34]. Since our fault tolerant introsort only protects local sorting, the *error coverage* is the ratio of the number of comparisons in introsort to the total number of comparisons in HykSort:

$$Cov = \frac{n \log n}{n \log n + k \log n + n \frac{\log p}{\log k}} = \frac{1}{1 + \frac{k}{n} + \frac{\log p}{\log n \log k}} \quad (7)$$

HykSort uses $k = 128$ by design. The only variables here are n and p . We calculate the lower bound of error coverage according to standard experimental settings [34]. In weak scaling, the setting with the lowest coverage is: $n = 2^{25}$, $p = 2^{18} = 262,144$. Incorporating these numbers into the formula, we get 90.7% coverage. In strong scaling, $N = 2^{38}$ and $p = 2^{18}$ yields the lowest coverage at 88.6%. These calculations show our fault tolerant introsort protects about 90% of the comparisons in HykSort.

100% coverage. To protect the comparisons outside local sorting, we propose the use of TMR for the negligible amount ($k \log n$) of comparisons in partitioning in step 3; we adapt Ravikumar's analysis [32] to protect step 5 of HykSort, which is the *k-sorted-merge*. We present the following corollary:

Given k sorted subarrays, the subarrays can be merged into a single sorted array with at most e comparison errors. The runs of the output will be at most $e + 1$.

Proof: Ravikumar proves that if there are e comparison errors in a standard mergesort, the output array will have a runs of at most $e + 1$ [32]. Suppose all e comparison errors happen after k sorted subarrays are finished: the later procedure will be exactly a *k-sorted-merge*, and the corollary holds true. $\square$

Based on this analysis, a similar error correction technique for heapsort output, *e-sorted-merge*, can be applied to the *k-sorted-merge*. Obviously, if $e > k - 1$, the correction technique complexity ($n \log(e + 1)$) is reduced to the original problem complexity ($n \log k$). We recommend taking advantage of our prior knowledge of the error rate, $e < \log n$, and using *e-sorted-merge* to protect *k-sorted-merge* if $e < k - 1$. If $e > k - 1$, TMR may be used for *k-sorted-merge*.

6.2.2 Protection for Memory. To tolerate memory errors, we introduce XOR-based re-execution (XR) to HykSort as following:

- Apply XR to local sorting (introsort).
- When all local sorts are complete, reduce the local XORs across all processes to get a global XOR.
- When global sorting is complete, each process calculates a local XOR and reduce to obtain another global XOR.
- If the two global XORs vary, re-execute HykSort.

The logic is clear: XR applied to introsort protects the memory errors in local sorting and global XORs, outside local sorting. As for the communication induced errors, though some of them (for example, bit-flip errors) will be covered by XOR, other communication errors may not be covered and need to be studied in detail which is out of scope of this paper.

7 EVALUATIONS

7.1 Experimental Setup

Computing platform & compiling & running. We use Tardis—a cluster equipped with 16 computing nodes. Each node is powered by 2 AMD Opteron 6200 series CPUs totaling 32 cores with 32GB ECC memory. All nodes are connected via an InfiniBand switch. A second platform we use is Bebop [1] which contains 664 nodes. Each node has 36 cores by 2 Intel Xeon E5-2695v4 processors with 128GB memory. Programs are compiled by *gcc* – 7.1.0 with optimization flag *-O3*. By default, each result reported is the mean of 10 measurements unless otherwise specified. If not mentioned explicitly, results are on Tardis.

Input sizes. For sequential trials, we generate 2^{29} *double precision floating-point* numbers uniformly distributed in $(0, 1)$; each double is 8 bytes, totaling 4GB, if input size is not mentioned explicitly. For parallel testing, we generate 2^{25} uniformly distributed *single precision floating-point* numbers per core, totalling 128MB, on Tardis and 2^{26} uniformly distributed *double precision floating-point* numbers on each process, totalling 512MB per core on Bebop.

Error injection. Comparison and memory errors are infrequent in the real world, and experimenting with them is not practical. Our experiment is thus based on error injection at the source code level. We do not use fault injectors because they simulate larger scope of errors beyond ours. Also, they introduce extra overhead which makes our experiment not accurate. Our injection can simulate all errors discussed with negligible overhead. To inject comparison errors to quicksort, we set a counter to track partitioning and flip a paired comparison result every time the counter increases by a fixed amount. Similarly, comparison errors are injected into heapsort by flipping a paired comparison every few invocations of `adjust_heap()`. Note that the number of comparison errors can vary since different inputs can have a varying effect on counter increase, but it is set to be at most $\log n$ as directed by our original limit. To inject memory errors, we first randomly determine whether the error occurs during the sorting of the first partition or the second partition at recursion level 1, which determines if the error occurs before or after the second checksum, and we then override a randomly picked element with a randomly generated value in $(0, 1)$. We only inject a single memory error in our experiments, as one is enough to invalidate the result of the sort. Without

Table 2: Unsortedness/Runs distribution for erroneous quicksort/heapsort based on 1024 trials. The Q and H rows show the distribution of unsortedness and runs respectively.

Interval	0 - 21	22 -25	26	27	28	29	30	31- ∞
Q	0	21	109	236	310	348	0	0
H	0	265	371	274	99	12	3	0

Table 3: Evaluation of conservative protection compared to two naïve protections, with and without comparison errors. Given the execution time of unprotected introsort is 102.9 seconds.

	Error occurs?	Overhead		Sorted?
		Time (s)	Pct. (%)	
CP	No	1.1	1.1%	✓
	Yes	1.9	1.8%	✓
NP1	No or Yes	14.3	13.9%	✓
NP2	No	0.9	0.9%	✓
	Yes	31.0	30.1%	✓

loss of generality, we inject the error in the first execution of a fault tolerant strategy; i.e., no error occurs in the re-execution.

7.2 Validity of Theorems

The efficiency of proposed protection is based on both Theorem 1 and Theorem 2 holding true. We illustrate the accuracy of both theorems in this evaluation. Our evaluation is based on a random array containing 2^{29} elements. Here we inject a fixed number of errors (29) and the experiment is repeated 1024 times. Each theorem targets a different sorting algorithm, so we provide two variants of the experiment. To evaluate Theorem 1, we set the recursion depth limit of introsort as n to force introsort to use quicksort and not heapsort. To evaluate Theorem 2, we set the depth limit as 0 so that heapsort is used and quicksort is not.

The evaluation results are shown in Table 2. We see unsortedness and runs of all 1024 trials are within their theoretical bounds, $29 = e$ and $59 = 2e + 1$ respectively. More specifically, the second row shows the unsortedness distribution of quicksort outputs. In the 1024 trials, there were 21 trials with unsortedness between 22-25, 109 trials with unsortedness 26, and so on; there were no trials with unsortedness greater than the bound, 29. The third row shows the runs distribution of heapsort outputs similarly.

7.3 Protection Against Errors

7.3.1 Comparison Errors Only. This experiment aims to justify two claims: (1) Our protection is resilient to comparison errors; (2) Our protection incurs far less overhead than baseline naïve protections. We evaluate conservative protection (CP), and two naïve protection strategies. The first naïve strategy (NP1) is detailed in Section 5.1.1. The second naïve protection (NP2) first sorts the array using unprotected introsort, then checks if the resulting array is sorted or not based on a TMR-protected scan of the array; the result array is returned if it is sorted, otherwise TMR-protected introsort is used to sort the result array again. We run each protection with and without errors; the results are shown in Table 3.

Table 4: Evaluation of XOR-based re-execution and pure re-execution with and without memory errors.

Overhead →	Error occurs?			
	No		Yes	
	Time (s)	Pct. (%)	Time (s)	Pct. (%)
<i>PR</i>	104.4	101.5%	208.2	202.3%
<i>XR</i>	2.9	2.8%	80.3	78.0%
Sorted?	✓		✓	

Table 5: Overhead of our full protection combining *CP* and *XR* and the naïve baseline combining *NP1* and *PR* with four combinations of errors: no errors (0), memory errors only (*M*), comparison errors only (*C*), and both errors (*M* + *C*).

Error Combination →	0	<i>M</i>	<i>C</i>	<i>M</i> + <i>C</i>
<i>CP</i> + <i>XR</i>	3.7%	85.6%	4.5%	85.8%
<i>NP1</i> + <i>PR</i>	130.4%	246.6%	130.4%	246.6%

Claim 1 is justified: in all the trials conservative protection yields well-sorted outputs in all cases. Claim 2 is also well-founded since our strategies incur at most 1.8% overhead, which is far less than the 13.9% overhead of *NP1* and 30.1% of *NP2* when errors arise.

7.3.2 Memory Errors Only. Here we aim to justify the claim: Our protection for memory errors, XOR-based re-execution (*XR*), incurs far less overhead than pure re-execution (*PR*).

To validate the claim, we perform an experiment that compares the overhead of our *XR* and that of *PR* with and without error occurrence. We evaluate each configuration using 50 trials and report the average. The results are shown in Table 4. Claim is validated, as we see *XR* incurs significantly lower overhead than *PR*: 2.8% v.s. 101.5% when no error occurs and 78% v.s. 202.3% otherwise.

7.3.3 Both Error Types. We create a strategy for tolerating both error types concurrently by combining one comparison protection strategy with one memory protection strategy. FT-iSort is the combination of conservative protection (*CP*) and XOR-based re-execution (*XR*). To show the effectiveness of FT-iSort, we compare it with a baseline that combines naïve TMR comparison error protection (*NP1*) with pure re-execution (*PR*). Table 5 shows that our full protection incurs far less overhead than naïve full protection. FT-iSort overhead is only a small fraction of the naïve protection overhead: the ratio of the former to the latter is respectively 2.8% when no error occurs, 34.7% when memory errors occur, 3.5% when comparison errors occur, and 34.8% when both errors occur.

7.3.4 Input sizes, data types and platforms. We present how our conservative protection behaves with varying input sizes, data types and different platforms in Figure 7. Specifically, we randomly generated double-precision, single-precision floating point numbers and integers with input sizes scaling from 2^{28} through 2^{31} on both Tardis and Bebop, taking up to 16GB memory for double-precision, 8GB for single-precision floating point number and 8GB for integer, respectively. Comparison errors are injected to each experiment. In Figure 7, we observe that all runs have overheads lower than 4%. Surprisingly, the execution time of conservative protection for

single-precision floating point numbers on Tardis is slightly faster than error free run of original unprotected sorting. This is likely due to the pretty small overhead of our solution and objective observation errors in experiments.

Figure 7: Overhead of conservative protection with different input sizes, data types and platforms

7.3.5 Increasing number of injected errors. The above experiments inject comparison errors within the error bound, $\log n$. In this section, we showcase the performance of both conservative protection and aggressive protection with increasing number of comparison errors both in the average and the worst case scenario. Input is 2^{31} random floating-point numbers in double precision.

In order to inject more average case errors, we just set the error bounds to be 20, 40, ..., 100 using the injection we mentioned in Section 7.1.

We manually emulate the worst case errors since the above error injection can only inject average case comparison errors. The worst case error for aggressive protection and conservative protection is that the error makes an element misplaced to a far-away location to its correct place in the array. In this scenario, insertion sort in aggressive protection has to move the element to its right location by going through almost the whole array; also, Algorithm 2 in conservative protection will have the greatest number of comparisons in binary search and data movements. Specifically, we manipulate a sorted array to be unsorted by assigning the first several elements to be the last several elements. I. e., one assigned element is simulating one worst case error because it should be at the end while the comparison error makes it placed at the beginning. Then, we let the manipulated array go to insertion sort (aggressive protection) and Algorithm 2 (conservative protection). We manipulated up to 100 worst case errors for conservative protection and up to 15 for aggressive protection because aggressive protection will take a much longer time than conservative protection with more than 15 errors. The results are shown in Figure 8. As can be seen, if the comparison errors happen in the average case, the aggressive protection is slightly faster than conservative protection. Also, their overhead curves are relatively flat with increasing errors. However, if the errors happen in the worst case, only conservative protection will have flat curve. Aggressive protection becomes more and more slower than conservative protection with over 5 injected errors.

7.4 Protecting Parallel Sorting

To illustrate the applicability of our fault tolerance strategy for parallel sorting based on introsort, we link HykSort with the library containing our fault tolerant introsort; this protects only the

Figure 8: Overhead of aggressive protection and conservative protection when errors happen in average case and in the worst case

Figure 9: Weak (left) and strong scaling (right) of HykSort (Default), HykSort with *CP + Mem*, and HykSort with *NP1* on Tardis.

Figure 10: Weak scaling of HykSort (Default), HykSort with *CP + Mem*, and HykSort with *NP1* on Bebop.

comparisons within introsort, and TMR can be used to protect the comparisons outside introsort to achieve 100% comparison error coverage. We study how our fault tolerance strategies *CP*, as well as the naïve protection strategy *NP1*, impact HykSort’s performance in the presence of comparison errors. We also implement the memory protection, *XR*, for *CP*; we did not inject memory errors because once they are injected, the baseline, *PR*, needs two more re-executions and our *XR* solution just needs one more which makes our solution an easy winner as shown in Table 4. So we just compare the performance when comparison errors are injected though this is already unfair to our solution since we are comparing *CP* with *XR* who can detect memory error to *NP1* who cannot. In

the evaluation, comparison errors are injected and 2-way HykSort is used. HykSort without protection will fail to sort correctly and comparison errors are not injected for it. We perform scaling test with various protection strategies and compare them to unmodified HykSort, as shown in Figures 9 and 10, where *+Mem* indicates that memory protection is added. In all trials, HykSort protected with any strategy sorts the array correctly. We observe the overhead of *CP* is up to only 10.9% across both platforms while *NP1* incurs at least 17% overhead; and HykSort using our fault tolerance strategy scales as well as the original HykSort.

8 RELATED WORK

Though the study of fault tolerance applied to introsort is non-existent, in the category of general sorting much progress has been made studying effective tolerance of comparison errors and memory errors [30]. The problem is first raised as binary search with errors (Rényi-Ulam games) [38] where questioner may get yes-or-no answer from a lying responder. Later on, it is studied in the sorting problem. Lower bounds on the minimum query strategy to get sorted output have been proved [25]. However, the theory cannot be directly applied to build a fault tolerant sorting algorithm because the algorithm fixes its query strategy. Then, analysis continue to focus on specific sorting algorithms such as quicksort and mergesort. Alonso et al. [5] have theoretically studied the number of inversions in the output of quicksort when random comparison errors are generated based on probability. Ravikumar [32] proposes a fault tolerant mergesort that can tolerate up to $\Theta(\frac{\log n}{\log \log n})$ erroneous comparisons. Leighton et al. [26] have proved tight lower bounds on the number of redundant comparators and replicators to build a fault tolerant sorting network, given the probability of destructive errors [6]. The above works consider comparison errors. To cover memory errors, Finocchi et al. propose an $O(n \log n)$ sorting algorithm that can tolerate up to $O(\sqrt{n \log n})$ memory errors [18]; they also present a method that is able to ensure the sorting of uncorrupted data is correct [19]. These works are of value in terms of theoretical analyses and pure algorithm designs. However, none of them can be efficiently applied to real world HPC environment due to non-negligible constants in their complexity bounds and sophisticated mechanism. For example, the study [17] shows that fault tolerant sorting algorithm proposed by Finocchi et al. introduces 1.5X to 2X more running time than the original sorting.

9 CONCLUSION

We quantitatively prove the impact of comparison errors on the output of *quicksort* and *heapsort*. We propose fault tolerant algorithms that protect introsort against both comparison errors and memory errors, and provide an efficient and practicable fault tolerant introsort based on the default sorting function in the GNU Standard C++ Library. We extend these algorithms to parallel sorting as well.

ACKNOWLEDGMENTS

The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant No. 1619253. This work was also supported by National Science Foundation CCF 1513201 and National Key Research and Development Programs No. 2017YFB0202100. We acknowledge the computing resources provided on Bebop, which is operated by the Laboratory Computing Resource Center at Argonne National Laboratory. We thank the anonymous reviewers for their insightful suggestions.

REFERENCES

- [1] [n. d.]. Bebop. ([n. d.]). Retrieved August 16, 2019 from <https://www.lcrc.anl.gov/systems/resources/bebop/>
- [2] [n. d.]. Soft error. ([n. d.]). Retrieved August 16, 2019 from https://en.wikipedia.org/wiki/Soft_error
- [3] [n. d.]. Top 10 Algorithms. ([n. d.]). Retrieved August 16, 2019 from <https://cs.gmu.edu/~henryh/483/top-10.html>
- [4] [n. d.]. Triple Modular Redundancy. ([n. d.]). Retrieved August 16, 2019 from https://en.wikipedia.org/wiki/Triple_modular_redundancy
- [5] Laurent Alonso, Philippe Chassaing, Florent Gillet, Svante Janson, Edward M Reingold, and René Schott. 2004. Quicksort with unreliable comparisons: a probabilistic analysis. *Combinatorics, Probability and Computing* 13, 4-5 (2004), 419–449.
- [6] Shay Assaf and Eli Upfal. 1990. Fault tolerant sorting network. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. IEEE, 275–284.
- [7] Jon Bentley. 2016. *Programming pearls*. Addison-Wesley Professional, 110.
- [8] Jon L Bentley and M Douglas McIlroy. 1993. Engineering a sort function. *Software: Practice and Experience* 23, 11 (1993), 1249–1265.
- [9] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2003. Automated application-level checkpointing of MPI programs. In *ACM Sigplan Notices*, Vol. 38. ACM, 84–94.
- [10] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations* 1, 1 (2014), 5–28.
- [11] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, et al. 2018. Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 68.
- [12] Jieyang Chen, Sihuan Li, and Zizhong Chen. 2016. GPU-ABFT: Optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–2.
- [13] Jieyang Chen, Xin Liang, and Zizhong Chen. 2016. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 993–1002.
- [14] Zizhong Chen. 2009. Optimal real number codes for fault tolerant matrix operations. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 29.
- [15] Zizhong Chen. 2013. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 167–176.
- [16] Curtis R Cook and Do Jin Kim. 1980. Best sorting algorithm for nearly sorted lists. *Commun. ACM* 23, 11 (1980), 620–624.
- [17] Umberto Ferraro-Petrillo, Irene Finocchi, and Giuseppe F Italiano. 2006. The price of resiliency: a case study on sorting with memory faults. In *European Symposium on Algorithms*. Springer, 768–779.
- [18] Irene Finocchi, Fabrizio Grandoni, and Giuseppe F Italiano. 2006. Optimal resilient sorting and searching in the presence of memory faults. In *International Colloquium on Automata, Languages, and Programming*. Springer, 286–298.
- [19] Irene Finocchi and Giuseppe F Italiano. 2004. Sorting and searching in the presence of memory faults (without redundancy). In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, 101–110.
- [20] Al Geist. 2016. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum* 10 (2016), 2–3.
- [21] Al Geist. 2016. Supercomputing’s monster in the closet. *IEEE Spectrum* 53, 3 (2016), 30–35.
- [22] Garth Gibson, Bianca Schroeder, and Joan Digney. 2007. Failure tolerance in petascale computers. *CTWatch Quarterly* 3, 4 (2007), 4–10.
- [23] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.
- [24] Lorenz Hübschle-Schneider and Peter Sanders. 2018. Communication Efficient Checking of Big Data Operations. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 650–659.
- [25] KB Lakshmanan, Bala Ravikumar, and K Ganesan. 1991. Coping with erroneous information while sorting. *IEEE Trans. Comput.* 40, 9 (1991), 1081–1084.
- [26] Tom Leighton and Yuan Ma. 1999. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM J. Comput.* 29, 1 (1999), 258–273.
- [27] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. 2018. Modeling Soft-Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 27–38. <https://doi.org/10.1109/DSN.2018.00016>
- [28] Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. 2017. Correcting soft errors online in fast fourier transform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 30.
- [29] David R Musser. 1997. Introspective sorting and selection algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993.
- [30] Andrzej Pelc. 2002. Searching games with errors—Fifty years of coping with liars. *Theoretical Computer Science* 270, 1-2 (2002), 71–109.
- [31] James S Plank, Kai Li, and Michael A Puening. 1998. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 9, 10 (1998), 972–986.
- [32] Bala Ravikumar. 2002. A fault-tolerant merge sorting algorithm. In *International Computing and Combinatorics Conference*. Springer, 440–447.
- [33] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 243–254.
- [34] Hari Sundar, Dhairya Malhotra, and George Biros. 2013. Hyksort: a new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 293–302.
- [35] Hari Sundar, Dhairya Malhotra, and Karl W Schulz. 2013. Algorithms for high-throughput disk-to-disk sorting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 93.
- [36] Li Tan, Shuaiwen Leon Song, Panruo Wu, Zizhong Chen, Rong Ge, and Darren J Kerbyson. 2015. Investigating the interplay between energy efficiency and resilience in high performance computing. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 786–796.
- [37] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z Zhang, Darren Kerbyson, and Zizhong Chen. 2016. New-sum: A novel online abft scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 43–55.
- [38] Stanislaw M Ulam. 1991. *Adventures of a Mathematician*. Univ of California Press, 281.
- [39] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. 2017. Silent data corruption resilient two-sided matrix factorizations. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 415–427.
- [40] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. 2011. Fault tolerant matrix-matrix multiplication: correcting soft errors on-line. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*. ACM, 25–28.
- [41] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. 2016. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 31–42.
- [42] James F Ziegler and Helmut Puchner. 2004. *SER—history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran the original sort, re-executed sort (NP1 in paper), TMR protected sort (NP2 in paper) and our proposed FT-iSort on Tardis, a cluster equipped with 16 computing nodes. Each node is powered by 2 AMD Opteron 6200 series CPUs totaling 32 cores with 32GB ECC memory. All nodes are connected via an InfiniBand switch. Programs are compiled by gcc - 7.1.0 with optimization flag -O3. Programs are experiments with and without injected error. Each result reported is the mean of 10 measurements unless otherwise specified. Running time and correctness of the sorting results are recorded for all runs. For parallel run, we base our implementation on HykSort.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:

<https://github.com/sli049/FT-iSort>

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: 2 AMD Opteron 6200 series CPUs totaling 32 cores with 32GB ECC memory, InfiniBand switch,

Operating systems and versions: Cent OS 7 with linux kernel 3.10.0-957.5.1.el7.x86_64

Compilers and versions: Gcc 7.1.0

Applications and versions: HykSort

Libraries and versions: OpenMPI 2.1.1

Key algorithms: introsort

Input datasets and versions: generated random floating point number

Output from scripts that gathers execution environment information.

```
XDG_SESSION_ID=23908
HOSTNAME=tardis
SHELL=/bin/bash
TERM=xterm-color
HISTSIZE=1000
SSH_CLIENT=169.235.30.120 48272 22
```

```
SSH_TTY=/dev/pts/1
USER=USER
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=0
↪ 1;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;0
↪ 1:mi=01;05;37;41:su=37;41:sg=30;43:ca=30;41:tw=3
↪ 0;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tg
↪ z=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lh
↪ a=01;31:*.lz4=01;31:*.lzh=01;31:*.lзма=01;31:*.t
↪ lz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.z
↪ ip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;
↪ 31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31
↪ :*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31
↪ :*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:
↪ *.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:
↪ *.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31
↪ :*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:
↪ *.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:
↪ *.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:
↪ *.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35
↪ :*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;3
↪ 5:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;
↪ 35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01
↪ ;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01
↪ ;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;
↪ 35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01
↪ ;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;3
↪ 5:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;3
↪ 5:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;3
↪ 5:*.aac=01;36:*.au=01;36:*.flac=01;36:*.mid=01;3
↪ 6:*.midi=01;36:*.mka=01;36:*.mp3=01;36:*.mpc=01;
↪ 36:*.ogg=01;36:*.ra=01;36:*.wav=01;36:*.axa=01;3
↪ 6:*.oga=01;36:*.spx=01;36:*.xspf=01;36:
LD_LIBRARY_PATH=/usr/local/lib:
PATH=/home/USER/openmpi/mypmi/bin:/home/USER/gccsc18
↪ /bin:/usr/local/bin:/home/XXXhidenameXXX901/slur
↪ m/bin:/home/XXXhidenameXXX901/slurm/sbin:/usr/lo
↪ cal/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/usr/
↪ local/go/bin:/home/USER/bin
MAIL=/var/spool/mail/USER
_=/usr/bin/env
PWD=/home/USER/SC19AD/Author-Kit
LANG=en_US.UTF-8
MODULEPATH=/act/modulefiles:/usr/share/Modules/modul
↪ efiles:/etc/modulefiles
LOADEDMODULES=
HISTCONTROL=ignoredups
HOME=/home/USER
SHLVL=2
LOGNAME=USER
SSH_CONNECTION=169.235.30.120 48272 169.235.30.199 22
MODULESHOME=/usr/share/Modules
LESSOPEN=||/usr/bin/lesspipe.sh %s
```

```

XDG_RUNTIME_DIR=/run/user/1618
BASH_FUNC_module()=( ) { eval `usr/bin/modulecmd
↪ bash $*`
}
+ lsb_release -a
./collect_environment.sh: line 10: lsb_release:
↪ command not found
+ uname -a
Linux tardis 3.10.0-957.5.1.el7.x86_64 #1 SMP Fri Feb
↪ 1 14:54:57 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
+ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                 32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):              2
NUMA node(s):          4
Vendor ID:              AuthenticAMD
CPU family:             21
Model:                  1
Stepping:               2
CPU MHz:                1400.000
CPU max MHz:            2100.0000
CPU min MHz:            1400.0000
BogoMIPS:               4200.32
Virtualization:         AMD-V
L1d cache:              16K
L1i cache:              64K
L2 cache:               2048K
L3 cache:               6144K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
NUMA node2 CPU(s):     16-23
NUMA node3 CPU(s):     24-31
Flags:                  fpu vme de pse tsc msr pae mce
↪ cx8 apic sep mtrr pge mca cmov pat pse36 clflush
↪ mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt
↪ pdpe1gb rdtscp lm constant_tsc art rep_good nopl
↪ nonstop_tsc extd_apicid amd_dcm aperfmperf pni
↪ pclmulqdq monitor ssse3 cx16 sse4_1 sse4_2 popcnt
↪ aes xsave avx lahf_lm cmp_legacy svm extapic
↪ cr8_legacy abm sse4a misalignsse 3dnowprefetch
↪ perfctr_core perfctr_nb cpb hw_pstate
↪ retpoline_amd ssbd ibpb vmcall arat npt lbrv
↪ svm_lock nrip_save tsc_scale vmcb_clean
↪ flushbyasid decodeassists pausefilter pfthreshold
+ cat /proc/meminfo
MemTotal:               57529948 kB
MemFree:                7709328 kB
MemAvailable:           55450528 kB
Buffers:                14906268 kB

```

```

Cached:                 12490616 kB
SwapCached:             4316 kB
Active:                  11496092 kB
Inactive:                15928176 kB
Active(anon):            33724 kB
Inactive(anon):          29424 kB
Active(file):            11462368 kB
Inactive(file):          15898752 kB
Unevictable:             0 kB
Mlocked:                 0 kB
SwapTotal:               11313148 kB
SwapFree:                11206796 kB
Dirty:                   0 kB
Writeback:               0 kB
AnonPages:               23948 kB
Mapped:                  46532 kB
Shmem:                   35764 kB
Slab:                    21318604 kB
SReclaimable:            20986704 kB
SUnreclaim:              331900 kB
KernelStack:             7888 kB
PageTables:               7064 kB
NFS_Unstable:             0 kB
Bounce:                  0 kB
WritebackTmp:            0 kB
CommitLimit:             40078120 kB
Committed_AS:            723844 kB
VmallocTotal:            34359738367 kB
VmallocUsed:              397560 kB
VmallocChunk:            34316759036 kB
HardwareCorrupted:       0 kB
AnonHugePages:           0 kB
CmaTotal:                 0 kB
CmaFree:                  0 kB
HugePages_Total:         0
HugePages_Free:          0
HugePages_Rsvd:          0
HugePages_Surp:          0
Hugepagesize:            2048 kB
DirectMap4k:             271872 kB
DirectMap2M:             8099840 kB
DirectMap1G:             50331648 kB
+ inxi -F -c0
./collect_environment.sh: line 14: inxi: command not
↪ found
+ lsblk -a
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda 8:0 0 10.9T 0 disk
└─sda1 8:1 0 10.9T 0 part /data2
sdb 8:16 0 111.8G 0 disk
├─sdb1 8:17 0 1G 0 part /boot
├─sdb2 8:18 0 3.9G 0 part
└─sdb3 8:19 0 106.4G 0 part
    └─centos-root 253:0 0 50G 0 lvm /
        └─centos-swap 253:1 0 10.8G 0 lvm [SWAP]

```

FT-iSort: Efficient Fault Tolerance for Introsort

```

└─centos-home 253:2    0 45.6G 0 lvm /data
+ lsscsi -s
[0:2:0:0]    disk    LSI      MR9261-8i      2.70
↳ /dev/sda  11.9TB
[3:0:0:0]    disk    ATA      INTEL SSDSC2CW12 400i
↳ /dev/sdb  120GB
+ module list
++ /usr/bin/modulecmd bash list
No Modulefiles Currently Loaded.
+ eval
+ nvidia-smi
./collect_environment.sh: line 18: nvidia-smi:
↳ command not found
+ lshw -short -quiet -sanitize
+ cat
WARNING: you should run this program as super-user.
WARNING: output may be incomplete or inaccurate, you
↳ should run this program as super-user.
H/W path      Device Class      Description
=====
/0              system      Computer
/0/0            bus        Motherboard
/0/0            memory     56GiB System memory
/0/1            processor  AMD Opteron(TM)
↳ Processor 6272
/0/2            processor  AMD Opteron(TM)
↳ Processor 6272
/0/100          bridge     RD890 Northbridge
↳ only dual slot (2x16) PCI-e GFX Hydra part
/0/100/2        bridge     RD890/RD9x0/RX980
↳ PCI to PCI bridge (PCI Express GFX port 0)
/0/100/2/0      storage    MegaRAID SAS
↳ 2108 [Liberator]
/0/100/9        bridge     RD890/RD9x0/RX980
↳ PCI to PCI bridge (PCI Express GPP Port 4)
/0/100/9/0      enp4s0    network    82574L Gigabit
↳ Network Connection
/0/100/a        bridge     RD890/RD9x0/RX980
↳ PCI to PCI bridge (PCI Express GPP Port 5)
/0/100/a/0      enp3s0    network    82574L Gigabit
↳ Network Connection
/0/100/b        bridge     RD890/RD990 PCI
↳ to PCI bridge (PCI Express GFX2 port 0)
/0/100/b/0      ib0       network    MT27500 Family
↳ [ConnectX-3]
/0/100/11       storage
↳ SB7x0/SB8x0/SB9x0 SATA Controller [IDE mode]
/0/100/12       bus        SB7x0/SB8x0/SB9x0
↳ USB OHCI0 Controller
/0/100/12.1     bus        SB7x0 USB OHCI1
↳ Controller
/0/100/12.2     bus
↳ SB7x0/SB8x0/SB9x0 USB EHCI Controller
/0/100/13       bus        SB7x0/SB8x0/SB9x0
↳ USB OHCI0 Controller

```

```

/0/100/13.1     bus        SB7x0 USB OHCI1
↳ Controller
/0/100/13.2     bus
↳ SB7x0/SB8x0/SB9x0 USB EHCI Controller
/0/100/14       bus        SBx00 SMBus
↳ Controller
/0/100/14.1     storage
↳ SB7x0/SB8x0/SB9x0 IDE Controller
/0/100/14.3     bridge
↳ SB7x0/SB8x0/SB9x0 LPC host controller
/0/100/14.4     bridge     SBx00 PCI to PCI
↳ Bridge
/0/100/14.4/1   display    ASPEED Graphics
↳ Family
/0/100/14.4/2   bus        FW322/323
↳ [TrueFire] 1394a Controller
/0/100/14.5     bus
↳ SB7x0/SB8x0/SB9x0 USB OHCI2 Controller
/0/101          bridge     Family 15h
↳ Processor Function 0
/0/102          bridge     Family 15h
↳ Processor Function 1
/0/103          bridge     Family 15h
↳ Processor Function 2
/0/104          bridge     Family 15h
↳ Processor Function 3
/0/105          bridge     Family 15h
↳ Processor Function 4
/0/106          bridge     Family 15h
↳ Processor Function 5
/0/107          bridge     Family 15h
↳ Processor Function 0
/0/108          bridge     Family 15h
↳ Processor Function 1
/0/109          bridge     Family 15h
↳ Processor Function 2
/0/10a          bridge     Family 15h
↳ Processor Function 3
/0/10b          bridge     Family 15h
↳ Processor Function 4
/0/10c          bridge     Family 15h
↳ Processor Function 5
/0/10d          bridge     Family 15h
↳ Processor Function 0
/0/10e          bridge     Family 15h
↳ Processor Function 1
/0/10f          bridge     Family 15h
↳ Processor Function 2
/0/110          bridge     Family 15h
↳ Processor Function 3
/0/111          bridge     Family 15h
↳ Processor Function 4
/0/112          bridge     Family 15h
↳ Processor Function 5

```

/0/113	bridge	Family 15h
↪ Processor Function 0		
/0/114	bridge	Family 15h
↪ Processor Function 1		
/0/115	bridge	Family 15h
↪ Processor Function 2		
/0/116	bridge	Family 15h
↪ Processor Function 3		
/0/117	bridge	Family 15h
↪ Processor Function 4		
/0/118	bridge	Family 15h
↪ Processor Function 5		
/0/3	system	PnP device PNP0c02
/0/4	system	PnP device PNP0c02
/0/5	system	PnP device PNP0b00
/0/6	system	PnP device PNP0c02
/0/7	system	PnP device PNP0c02
/0/8	system	PnP device PNP0c02
/0/9	communication	PnP device PNP0501
/0/a	communication	PnP device PNP0501
/0/b	system	PnP device PNP0c02
/0/c	system	PnP device PNP0c01