An Algorithm-Based Fault Tolerance Strategy for the Bitonic Sort Parallel Algorithm

Edson T. Camargo*, Elias P. Duarte Jr.†
†Federal University of Paraná (UFPR) Curitiba PR Brazil
*Federal Technology University of Parana (UTFPR) Toledo PR Brazil
Email: edson@utfpr.edu.br, elias@inf.ufpr.br

***Abstract*—High Performance Computing (HPC) systems are employed to solve hard problems and rely on parallel algorithms which present very long execution times – up to several days. These systems are expensive in terms of the computational resources required, including energy consumption. Thus, after failures occur it is highly desirable to loose as little of the work that has already been done as possible. In this work we present an Algorithm-Based Fault Tolerance (ABFT) strategy that can be applied to make a robust version of any hypercube-based parallel algorithm. Note that we do *not* assume a physical hypercube: after nodes crash, fault-free nodes autonomously adapt themselves according to a logical topology called VCube, preserving several logarithmic properties. The proposed strategy guarantees that the algorithm does not halt even after up to $(N-1)$ nodes crash, in a system of N nodes. We use parallel sorting as a case study, describing how to make a fault-tolerant version of the Bitonic Sort parallel algorithm. The algorithm was implemented in MPI using ULMF to handle faults. Experimental results are presented showing the performance and robustness of the proposed solution.**

I. Introduction

Parallel algorithms are at the core of High Performance Computing (HPC) systems which solve hard problems in several domains. They run on a multiple processors and cores, being capable of executing 10^{15} (peta-scale) and 10^{18} (exascale) floating point operations per second (FLOPS). Those very large-scale systems usually present a small MTBF (Mean Time Between Failures) [1]. For instance, the Blue Waters petascale system has an average MTBF of only 4.2 hours [2], which reduces even further for exascale systems [3], [4]. Frequent faults can become a very serious problem if we consider that (i) these systems usually take a long time to execute, and (ii) have very high demands in terms of computational resources and energy. It is thus essential to make these systems fault-tolerant, so that they can continue running correctly after faults, and no work is lost.

There are numerous techniques to make an HPC system fault-tolerant [5], [6]. These techniques include, among others, rollback recovery [7], replication [8], computation migration [9], and Algorithm-Based Fault Tolerance (ABFT). ABFT makes use of properties of the parallel algorithm itself to survive faults during its execution [10], [11], [12], [13]. The algorithm is designed to be resilient, and must be able to detect or receive fault notifications and adapt itself after faults occur. In this work we propose a general ABFT technique to make any parallel algorithm designed for hypercubes fault-tolerant. Developers can employ the proposed technique to leverage any parallel hypercube-based algorithms to survive faults efficiently. With the proposed ABFT technique, the algorithm is able to reconfigure itself autonomously in runtime, guaranteeing that it does not halt after faults are detected: fault-free nodes continue the execution and no work is lost.

Hypercubes are intrinsically scalable topologies that have been extensively used in parallel computing both as interconnection networks [14] and to organize the communication and execution of parallel and distributed algorithms [15], [16]. The ABFT strategy proposed in this work assumes a logical topology – no physical hypercube required. The fault-tolerance technique relies on the virtual topology known as the VCube [17], [18]. A VCube is a hypercube when all nodes are fault-free, but is capable of reconfiguring itself as nodes fail (and recover) keeping several logarithmic properties. In a VCube of d dimensions nodes are organized in hierarchical clusters of increasingly larger sizes which are the basis for the proposed ABFT strategy. While in the hypercube each node is connected to d predefined nodes, in the VCube a node is connected to whichever is the first fault-free node of d clusters, if there is any. We assume the fail-stop model, in which processes crash and can be detected as so. In a system of N nodes, even if up $N-1$ nodes become faulty the system autonomously reconfigures itself at runtime and continues the execution. Nodes can also be repaired and rejoin the system.

The heart of the ABFT strategy proposed for hypercube-based parallel algorithms in this work involves the VCube in two main ways. First, after a node crashes another fault-free node will *cover* the faulty node and execute its tasks. Second, nodes communicate among themselves according to the VCube topology. As mentioned before, when all nodes are fault-free the VCube is a hypercube, furthermore as nodes crash the topology reconfigures itself keeping multiple logarithmic properties. In order to describe how to make a parallel algorithm fault-tolerant with the proposed technique, we describe parallel sorting as a case study. Although parallel sorting algorithms have been proposed for more than five decades, they have become even more relevant in the context of big data, as sequential algorithms are not feasible to sort vast amounts of data. A fault-tolerant version of the Bitonic Sort parallel algorithm is specified, and its correctness and performance are also presented.

The algorithm was implemented with the Message Pass-

ing Interface MPI [19] and User Level Failure Mitigation (ULFM) [20], [21] to handle faults. MPI is one the most popular programming models for distributed-memory HPC systems. As the name implies, MPI is based on the message passing paradigm: nodes are connected to a network over which they communicate by sending and receiving messages. Each node has access to local memory. ULFM was proposed by the MPI forum to avoid having to abort and restart MPI-based systems after failures. ULFM allows not only the implementation of resilient MPI applications, but features programming language constructs that enable the system to detect and react to failures without aborting its execution. In the context of the proposed algorithms, ULFM is used basically in the context of failure detection. Experimental results are presented showing the performance and robustness of the proposed solution.

The rest of this work is organized as follows. Section II describes related work. Section III presents the proposed technique, the system model and an overview of the VCube topology. The application of the proposed ABFT technique to parallel sorting is presented in Section IV. The implementation of the proposed fault-tolerant version of the Bitonic Sort parallel algorithm and the results obtained are presented in Section V. Conclusions and future work follow in Section VI.

II. Related Work

In this section related work is grouped according to the four following topics: fault-tolerance for HPC systems, ABFT, parallel sorting, and fault-tolerant MPI systems.

The development of techniques to improve the resiliency of HPC systems is often preceded by field work that reveal the vulnerabilities of current systems. In an extensive field work published in 2017 [22] the authors describe 23 different types of hardware and software faults that can affect HPC systems. They stress that as the number of components of these systems increase, the likelihood of failures also increases, and worse, the complexity of managing the reliability of the system also grows; the consequences of this fact are non-trivial. For instance, performing accurate root-cause analysis of failures is sometimes not possible, given the complexity and sheer number of components along some fault paths. The authors also pose disturbing questions, among them whether newer components are less reliable, due not only to ever shrinking technologies and also design goals such as energy saving. The MTBF of the systems investigated varied from 7.45 to 22.67 hours (these results are normalized on number of processors in the system). The conclusion is that in such systems every single day at least one failure is expected to occur, probably more will. In another work [2], field data is presented for the reliability of the Blue Waters petascale system and the MTBF reported is of 4.2 hours.

Rollback recovery is perhaps the most widely adopted technique to improve the reliability of HPC systems [5]. This technique consists of establishing checkpoints to which the system can roll back in case of failures, instead of restarting from the scratch [7]. Note that the application of rollback recovery to HPC systems that take a very long time to execute and have very low MTBF presents several challenges. For instance, Tiwari et al. [23] reports results for an astrophysics application that generates 160TB of data and can take 360 hours to complete its execution. For that particular application, taking checkpoints at every hour can have a major impact on the system performance, especially because checkpoint incurs on high I/O overhead. Increasing the checkpointing interval can reduce the impact on the system, but it also increases the amount of work lost after crashes, which corresponds to the interval since the last checkpoint was taken and the instant of the failure. An alternative is to employ replication instead of rollback recovery. In [24], the authors justify the use of replication given the short MTTI (Mean Time To Interrupt), which corresponds to the time spent taking checkpoints. The authors also mention that replication can be extended to deal with Byzantine faults. In [25] the authors evaluate the impact of replication on the speedup of parallel algorithms. Their purpose is to determine the optimal number of replicas to use, given the failure rate.

Algorithm-Based Fault Tolerance (ABFT) is a technique that relies on the properties of the parallel algorithm itself to recover from faults during its execution [26], [27], [28], [29]. A difference of this technique to the previous ones is that those are transparent to the application, while ABFT is not, it is actually interwoven across the application's algorithm. ABFT can only be used if the underlying systems provides – usually through primitives – mechanisms for fault detection and notification.

ABFT was originally proposed by Huang and Abraham to detect and correct errors in the context of matrix operations, caused by transient or permanent hardware faults [30]. The technique relies on the fact that for some matrix operations there is a relationship between the input *checksum* and the *checksum* computed for the output. Based on this relationship, they proposed an ABFT technique to detect, locate and correct certain types of errors of matrix operations. Chen and Dongarra also employ an ABFT technique that relies on matrix operations to make an HPC system fault-tolerant [31], [11]. The proposed technique assumes the fail-stop model and allows their HPC systems to tolerate faults that occur at runtime. Like Huang and Abraham, Chen and Dongarra also make use of the relationship that exists between checksums computed for the inputs/outputs of matrix operations, the same mentioned above. Their goal is to keep a consistent global state, so that after a fault occurs, the corresponding computations can be re-executed. However, correct processes must wait before they continue running the application. The system was implemented with the FT-MPI library, which is an MPI implementation that supports fault detection and notification.

Wang et al. [29] propose a strategy, called ABFT-hot-replacement, to prevent correct nodes from having to stop and wait for nodes with faulty data to recover. This is done using spares, redundant nodes that can replace the faulty ones. Their work is also based on the *checksum* relationship of matrix

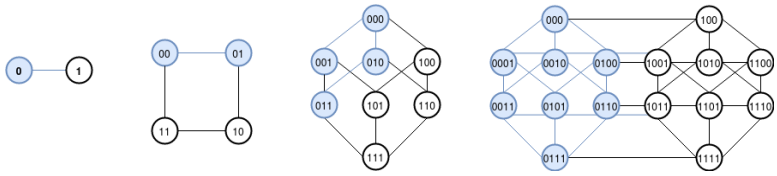

Fig. 1. The construction of d-dimensional hypercube from two $d-1$-dimensional hypercubes, $d = 2, 3, 4$.

operations, and was implemented with an MPICH system adapted to deal with application-level faults. Scholl et al. [32] refine techniques based on ABFT for operations on sparse matrices, the main purpose is to reduce fault localization time.

Checkpoint-on-Failure (CoF) [33] is a strategy that combines AFBT and rollback recovery. CoF does not save periodic checkpoints, instead they are triggered by node faults. Correct nodes save a checkpoint and stop the current execution. A new instance starts and the system (i) recovers the checkpoints of the correct nodes, and (ii) use an ABFT technique to recover data from faulty nodes.

An ABFT scheme that can be applied to a class of applications – instead of a specific application – has also been proposed [34]. The authors mention that their purpose is to overcome one of the disadvantages of ABFT, which they call its "non-universality", i.e. the fact that the technique is tied to a single specific application or algorithm. They propose an ABFT strategy that they call generic in the sense that it can be applied to a multiple parallel applications which present similar algorithmic and/or communication patterns.

In [35] the authors propose ABFT techniques to reduce the overhead of making heterogeneous systems based on GPU accelerators fault-tolerant. Their ABFT scheme takes into consideration both computing and memory storage faults. Recently, ABFT has also been applied to tolerate data errors in the context of machine learning and also computer vision [36].

Finally, closest to our work is the ABFT strategy applied to a hypercube-based parallel computer [37]. The strategy relies on the detection and location of faulty nodes at runtime, based on error detection mechanisms that are tailored for three parallel applications: matrix multiplication, Gaussian elimination, and fast Fourier transform. The main difference to our work is that they assume a physical hypercube (they employ a 16-processor Intel iPSC hypercube computer) while in our case the hypercube is a logical topology, which determines how nodes communicate assuming a fully connected underlying network which corresponds to the topologies of most networks used to run HPC systems.

Related work includes very few parallel sorting algorithms implemented with MPI [38], [39], although there are implementations on hybrid systems, such as Bitonic Sort on CUDA and MPI [40] and others that employ both distributed and shared memory [41], [42]. MPI is the *de facto* standard for developing parallel distributed memory applications [43], [19]. The MPI standards originally assumed a reliable infrastructure, and there were no features to be used to program fault-tolerant applications. Recently, the MPI-Forum has specified the ULFM [20], [21] which includes several constructs to handle faults. As an example, in our implementations we have used the `MPI_COMM_REVOKE` followed by the `MPI_COMM_SHRINK` constructs to restore the MPI application by dropping faulty processors [20]. More details are given in Section 5.

III. The Proposed Strategy

In this section we present the proposed ABFT strategy, which can be used to turn any parallel algorithm based on the hypercube fault-tolerant. Before that, we first give the definition of a hypercube and an overview of the VCube virtual topologogy.

A hypercube [44], [15], [14] can be represented as a graph $G = (V, E)$, where V is a set of vertices called nodes in this work. Each node has an identifier of length d bits, where d is the number of dimensions of the hypercube. A hypercube with dimension d has $N = 2^d$ nodes. Nodes have unique identifiers in the interval $[0, N-1]$. There is an edge $(i, j) \in E$ between two nodes i and j if and only if the identifiers of i and j differ in a single bit. A d-dimensional hypercube, $d > 1$, consists of two $(d-1)$-dimensional hypercubes. Figure 1 shows the construction of 2-, 3-, and 4-dimensional hypercubes each from two hypercubes of 1-, 2-, and 3-dimensions, respectively. The construction of a d-dimensional hypercube from a $d-1$-dimensional hypercubes can be done as follows. First duplicate the $d-1$-dimensional hypercube. Extend node identifiers by 1 bit, which is set to 0 in the ids the original nodes, and set to 1 in the ids of the newly created nodes. There is an edge connecting each node of the original $(d-1)$-dimensional hypercube with the corresponding node in newly created $(d-1)$-dimensional hypercube, as their identifiers differ in exactly one bit.

A hypercube is a regular graph with degree $logN$ (all logarithms are base 2 in this work), where N is the number of nodes. Besides the degree, the hypercube presents several other logarithmic properties, such as the diameter, which is the maximum shortest path between any two nodes in V and is also $logN$.

The strategy proposed in this work is based on the VCube virtual topology [17], [18]. The system is assumed to be fully connected, in the sense that each node can communicate with any other node directly, without the need of intermediaries. The communication between two nodes is assumed to be reliable. A VCube with N nodes is a hypercube when all nodes are fault-free. However, the topology is able to autonomously reconfigure itself upon node failures, keeping several logarithmic properties. In a VCube, nodes are organized in clusters,

TABLE I
$C_{i,s}$ FOR A SYSTEM WITH 8 PROCESSES.

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

which are defined per node. It is possible to say that VCube nodes communicate with clusters of nodes, usually the first fault-free node of each cluster. The $c(i, s)$ function returns the sequence of nodes of the s-th cluster of node i, s varies from 1 to $d = logN$, which is the VCube dimension. Thus the $c_{i,s}$ function is used to determine the nodes with which node i communicates. The expression below shows how this function is computed:

$c_{i,s} = (i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1})$

Table I shows an example of $c_{i,s}$, applied to a 3-dimensional VCube, thus $N = 8$ nodes. For example, for node $i = 0$, s varies from 1 to 3, and $c_{0,1}$ returns (1); $c_{0,2}$ returns $(2, 3)$; $c_{0,3}$ returns $(4, 5, 6, 7)$.

In a VCube nodes monitor their neighbors in testing intervals, which are periodic intervals of time determined with the local clock of each node. Global synchronization is *not* required. A node can be in one of two states: *faulty* or *fault-free*. The fail-stop model is assumed, thus nodes fail by crashing, and fault-free nodes keep information about which nodes are faulty [45]. At each testing round a fault-free node i tests its clusters $c(i, s), s = 1..logN$. Nodes of each $c(i, s)$ are tested sequentially until a fault-free node is found. If for example the first node is tested fault-free, a single test is executed. The tester then obtains information about new events – detected by the tested node since it had been tested in the previous interval. An event corresponds to a node crashing, thus its state changes from fault-free to faulty. If there are no fault-free nodes, the entire cluster is tested. Let a *testing round* be interval of time in which all fault-free nodes have completed their assigned tests. It has been shown [17] that using this strategy all fault-free nodes learn about any new event in at most log^2N rounds, in average much faster than that. Furthermore, using the test assignment proposed, at most $NlogN$ tests are executed per testing interval.

The ABFT strategy proposed in this work if node i is faulty, then another fault-free node will be its *cover* node and execute its tasks. Function *cover(i)* returns node i's cover: this is the first fault-free node in $c(i, 1)$. If that node is faulty, then the first fault-free node in cluster $c(i, 2)$ is selected, if that node is faulty but the second node of that cluster ($c(i, 2)$) is fault-free, then it is the cover. If in that cluster there is no fault-free node, the cluster size is incremented again until cluster $c(i, d)$ is considered.

A node that is covering a faulty node executes the fault-tolerant algorithm as itself and also assuming the ids of the nodes it is covering. If for instance node 0 is faulty, then its cover is node 1, which executes node 0's tasks. However, if node 1 is also faulty, then 0 is covered by node 2 because 2 is the first fault-free node in $c(0, 2)$. If node 2 is also faulty, the next to consider is node 3, and so on. In this way, a node i can execute the tasks of up to $N-1$ faulty nodes. If $N-1$ nodes are faulty, the single fault-free node in the system executes all tasks of nodes. This ABFT strategy implies the assumption that cover nodes have access to the data that would be used by the faulty nodes they are covering.

IV. FAULT-TOLERANT PARALLEL SORTING

This section presents the transformation of a parallel sorting algorithm based on the hypercube using the proposed ABFT strategy. Sorting is one of the most fundamental and well studied computing problems [46]. Given a list of N elements $L = (a_1, a_2, ..., a_i, ..., a_N)$ sorting consists of arranging these elements into a new list $L' = (a'_1, a'_2, ..., a'_i, ..., a'_N)$ so that $\forall i \mid 0 < i < N : a'_i < a'_{i+1}$. We say that L' is sorted in increasing order. Although the elements can also be sorted in decreasing order, i.e. $\forall i \mid 0 \leq i < N : a_i > a_{i+1}$, in this work we only use the increasing order.

Parallel sorting algorithms have been designed to take advantage of multiple processors to speed up sorting. Although they have been applied to diverse fields such as image processing, computational geometry and graph theory [47], [38], the recent advent of big data has renewed the interest in efficient parallel sorting strategies. Doing it in parallel may be the only choice to sort truly huge amounts of data for which sequential sorting is not viable. A large number of parallel sorting algorithms have been proposed along the past several decades [48], [44]. These algorithms have been designed for all types of parallel computing architectures and topologies, including physical and logical topologies that organize processing nodes in a meshes, rings, stars, hypercubes, among others.

Next we show how to make Bitonic Sort [49], [50] fault-tolerant with the proposed ABFT strategy. The strategy relies on the VCube logical topology to determine covers defined in the previous section, and also on how nodes communicate and share the sorting tasks. In this way the algorithms adapt themselves autonomously after faults occur, and continue running even if up to $N-1$ nodes crash. The algorithm runs in sorting rounds in which nodes execute some local processing (including locally sorting part of the data) and communicate among themselves (including sharing data to be sorted by partners).

A. Bitonic Sort Algorithm

The Bitonic Sort algorithm [49], [50] is another of the classic parallel sorting algorithms. This algorithm is based on

the comparison of elements of predefined sequences which are called "bitonic sequences". These comparisons are carried out in a way that does not depend on the input data. A bitonic sequence is a sequence of elements $seq = (a_0, a_1, ..., a_{m-1})$ with the following properties: (i) there is an index i, $0 \leq i \leq m-1$, such that $(a_0, ..., a_i)$ is monotonically increasing and $(a_{i+1}, ..., a_{m-1})$ is monotonically decreasing, or (ii) there is a cyclic rotation that satisfies (i). For example, $(2, 3, 6, 8, 7, 5, 4, 1)$ is a bitonic sequence for $i = 3$, consisting of a monotonically increasing sequence $(2, 3, 6, 8)$ followed by the monotonically decreasing sequence $(7, 5, 4, 1)$. As an example of cyclic rotation of seq consider for instance $(6, 8, 7, 5, 4, 1, 2, 3)$, in this case the element in the frontier of the increasing/decreasing (8) subsequences is in position $i = 1$. Any subsequence of a bitonic sequence is also bitonic.

Next, we describe how a bitonic sequence seq is sorted so that the resulting sequence is monotonically increasing. Initially, seq is divided in half, generating the sequences seq_1 and seq_2 both of size $m/2$. Thereafter, the first element of the seq_1 sequence is compared with the first element of sequence seq_2. The smallest element is assigned to sequence seq_1 and the largest element to sequence seq_2, that is: $seq_1 = (min\{a_0, a_{m/2}\}, min\{a_1, a_{m/2+1}\}, ..., min\{a_{m/2-1}, a_{m-1}\})$ and $seq_2 = (max\{a_0, a_{m/2}\}, max\{a_1, a_{m/2+1}\}, ..., max\{a_{m/2-1}, a_{m-1}\})$. Considering the example sequence $seq = (2, 3, 6, 8, 7, 5, 4, 1)$ presented above the result after the first step is the following: $seq_1 = (2, 3, 4, 1)$ and $seq_2 = (7, 5, 6, 8)$. Both seq_1 and seq_2 are also bitonic sequences. Note that all elements of seq_1 are less than those contained in seq_2. The next step is to apply the same method to each of the new sequences, and repeat it recursively until the sequences of 2 elements are ordered. In the end, all subsequences are joined to form the ordered original sequence.

The procedure of dividing a sequence of size m into two bitonic subsequences is called a bitonic split. The generation of an ordered sequence from bitonic subsequences is called a bitonic merge. Any bitonic sequence can be ordered by applying a bitonic split, followed by element comparisons, and bitonic merge of the ordered sequences in the end. Bitonic Sort can be adapted to different parallel topologies, including the hypercube as described next.

The N nodes that run the algorithm are organized as a d-dimensional VCube, where $d = \log N$. As in the previous algorithms, each node has a unique identifier i, $0 \leq i \leq N$. Sorting is performed in s rounds, where $1 \leq s \leq d$. In each round, node i forms a pair with the first fault-free node of cluster $c(i, s)$. Nodes i and j exchange elements with each other and make comparisons based on the elements exchanged in a given round. If $i < j$, node i keeps the smallest elements, node j keeps the largest elements. At the end of a round, the largest element of node i is less than or equal to the smallest element of node j.

Mapping a bitonic sequence to a hypercube can be done as follows. If the size of the sequence (m) is equal to the number of nodes ($m = N$), then each single element is mapped to a single hypercube node, i.e. element a_i is mapped to node i. If the size of the sequence is greater than the number of nodes ($m > N$), then m/N elements are mapped to each node. Note that if a node is faulty, then its sequence it assigned to its covering node. For the sake of simplicity, we first present the Fault-Tolerant Parallel Bitonic Sort Algorithm considering the simple case in which $m = N$ and each element is mapped to a single node. This simpler version is presented as Algorithm 1.

Algorithm 1 ***Fault-Tolerant Bitonic Sort with m=n*** (executed by node i)

```
1:  Begin
2:    d ← logN                                   {VCube dimension}
3:    while d > 0 do
4:      I ← i {Set I of identifiers to run the algorithm is initialized with
        i}
5:      F ← set of faulty nodes in the beginning of this round
6:      for all j | j ∈ F ∧ cover(j) == i do
7:        I ← I ∪ j                              {Add j to set I, covered node}
8:      for each k ∈ I in parallel do
9:        list ← a_k                             {The element assigned to node k}
10:       partner ← first fault-free node in c(k, d)
11:       if k < partner then
12:         compare_exchange min(k, partner)
13:       else
14:         compare_exchange max(k, partner)
15:     F' ← set of faulty nodes in the end of this round
16:     if F == F' then
17:       d ← d − 1
    End
```

The *compare_exchange* procedure (line 12 and 14) causes two nodes to exchange and compare elements. For example, consider the sequence $seq = (2, 3, 6, 8, 7, 5, 4, 1)$ assigned to a VCube with 8 fault-free nodes. Figure 2 shows how sorting is executed. Element a_i is mapped to node i. In the first round, a single 3-dimensional hypercube is considered and the following pairs of nodes are formed (line 10): (0, 4), (1, 5), (2, 6) and (3, 7). The node with smallest identifier keeps the smallest element and the node with the largest identifier keeps the largest element (lines 12 and 14, respectively). For example, nodes 0 and 4 keep elements 2 and 7, respectively. After the exchange and comparison procedure, both nodes keep the same elements. Nodes of other pairs, such as (2, 6) and (3, 7) do not keep their original elements after the execution of *compare_exchange*. In the next round two 2-dimensional hypercubes are considered, and new pair of nodes are formed according to line 4. Nodes exchange and compare their elements again. Note that from the start all elements of the 2-dimensional hypercube of which nodes have lower ids are all smaller than those of the other 2-dimensional hypercube. In the last round, pairs of nodes are formed in four 1-dimensional hypercubes, each of two nodes. As in the previous rounds, in case $i < j$, then node i will keep an element that is less than or equal to the element maintained by node j.

Now consider the general case in which $m > N$. In this case

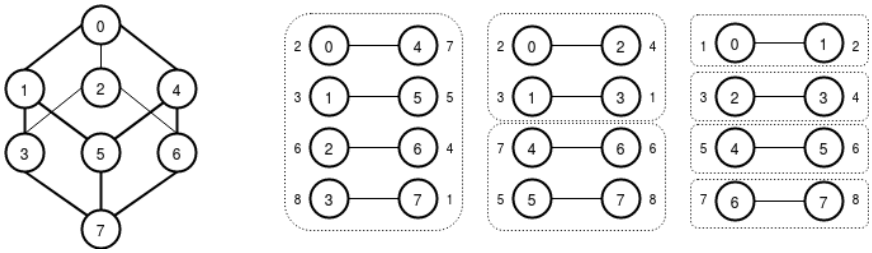

Fig. 2. *Bitonic Sort* on a 3-dimensional VCube: first, second and third sorting rounds. Node ids are shown in the circles, the element a node keeps is shown on its side.

Algorithm 2 ***Fault-Tolerant Bitonic Sort for any sequence*** (executed by node i)

```
1:  Begin
2:    d ← logN                                   {VCube dimension}
3:    for s ← 0 to d − 1 do
4:      t ← s
5:      while t >= 0 do
6:        I ← i   {Set I of identifiers to run the algorithm is initialized
          with i}
7:        F ← set of faulty nodes in the beginning of this round
8:        for all j | j ∈ F ∧ cover(j) == i do
9:          I ← I ∪ j                  {Add j to set I, covered node}
10:       for each k ∈ I in parallel do
11:         list ← a_k                 {The element assigned to node k}
12:         partner ← first fault-free node in c(k, t + 1)
13:         if (s + 1)^th bit of k ≠ t^th bit of k then
14:           compare_exchange min(k, partner)
15:         else
16:           compare_exchange max(k, partner)
17:       F' ← set of faulty nodes in the end of this round
18:       if F == F' then
19:         t ← t − 1
    End
```

each node receives m/N elements. In case there are faulty nodes, then the corresponding covers receive their elements. This version of the algorithm employs a *compare_exchange* procedure that is more general: instead of exchanging and comparing a single element, it exchanges and compares entire sequences of elements between the nodes of a pair (i, j). Then, each node makes comparisons according to the sequence indexes. Thus node i executes $seq_i[k] \leftarrow min(seq_i[k], seq_j[k])$ and node j executes $seq_j[k] \leftarrow max(seq_i[k], seq_j[k])$. As a result, if $i < j$ then all elements of seq_i are smaller than or equal to the elements of seq_j.

The version of Bitonic Sort presented as Algorithm 2 can receive as input *any* sequence, it is not restricted to bitonic ones. The first step is to transform the input sequence into a bitonic sequence. This is accomplished by repeatedly creating bitonic sequences of increasing size. To start with, note that any sequence of two elements is a bitonic sequence. Now in order to merge two bitonic sequences into a double sized bitonic sequence, a simple and cheap condition must be met: the first sequence must be increasing and the second sequence must be decreasing. Figure 3 illustrates the ordering process for the input sequence $seq = (7, 3, 6, 8, 1, 2, 5, 4)$ in a 3-dimensional hypercube. In Figure 3 symbols $\oplus$ and $\ominus$ represent the comparisons between the elements of the sequence. $\oplus$ defines that the comparisons should generate an increasing sequence and $\ominus$ a decreasing sequence.

Each node sends its local sequence to its partner. The pairs of nodes defined in line 12 in the first round for the example in the figure are: (0, 1), (2, 3), (4, 5), (6, 7). Line 13 is a clever implementation proposed in [44] to determine whether the node keeps the smallest elements or the largest elements. If the $(s+i)^{th}$ bit and the t^{th} bit are equal/different, then the node keeps the smallest/largest element, respectively. Otherwise, the node keeps the largest element. Thus considering the first pair of nodes, node 0 keeps element 3 and node 1 keeps element 7 since these nodes must generate an increasing sequence (see Figure 3). The elements contained in node 4 and 5 are maintained since they already formed an increasing sequence. In turn, the pairs nodes (2, 3), (6, 7) generate decreasing sequences. So node 2 keeps element 8 and node 3 keeps element 6. Nodes 6 and 7 keep their original elements.

In the second round, when $s \leftarrow 1$, the exchange and comparison process is repeated considering two 2-dimensional hypercubes and then four 1-dimensional hypercubes. Finally, the last round of Algorithm 2 is equivalent to Algorithm 1 (see Figures 2 and 3). Note that in the last round, the ordering process starts with the bitonic sequence $seq = (3, 6, 7, 8, 5, 4, 2, 1)$. At the end, the algorithm generates the sequence $seq = (1, 2, 3, 4, 5, 6, 7, 8)$.

Now we prove that the Fault-Tolerant Bitonic Sort algorithm completes in $logN$ rounds even if up to $N-1$ nodes fail during the execution.

Theorem 1. *The Fault-Tolerant Bitonic Sort algorithm completes sorting in $logN$ rounds even if $N-1$ nodes become faulty during those rounds.*

Proof. The proof is done by induction on the VCube dimension.

Basis: consider a hypercube with dimension $d = 1$, consisting of two nodes. In a single round sorting completes even if one of the nodes is faulty, as the fault-free node will cover the faulty node and sort the complete list.

Hypothesis: now assume that for a VCube of dimension $d = r$ sorting is completed in r rounds, even if up to $2^r - 1$ nodes become faulty.

Induction step:

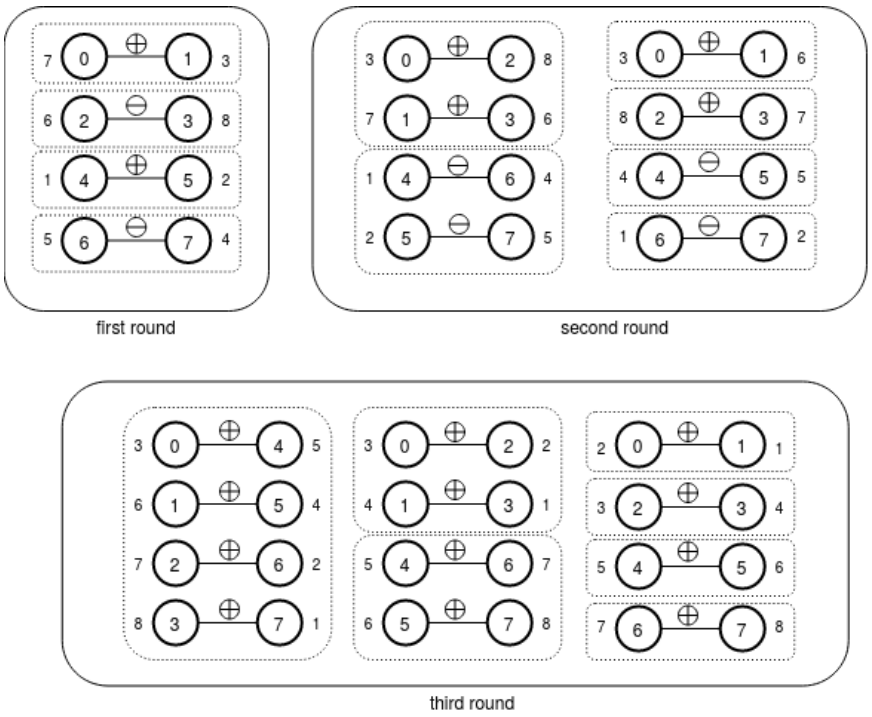

Fig. 3. An example execution of *Bitonic Sort* for an input sequence that is not bitonic on a 3-dimensional hypercube.

A VCube with dimension $d = r+1$ consists of two VCubes of dimension r. According to the induction step, in each of those two VCubes sorting completes in v rounds. Now in the first sorting round of Bitonic Sort, a fault-free node in one of those r-dimensional VCubes will find a fault-free partner in the other r-dimensional VCube, if there is one. They will act as covers for the faulty nodes within their r-dimensional VCubes and communicate with the partner of the other v-dimensional VCube in that round. If all nodes are faulty in one of those r-dimensional VCubes, the cover(s) will be on the other r-dimensional VCube and run all tasks of that round.

If a node is fault-free as it starts a round and then becomes faulty during that the round, then its tasks may not have been completed and the algorithm cannot leave the round, which is re-started. Nodes will only proceed to the next round after all fault-free nodes have executed their assigned tasks – including the tasks of the faulty nodes they are covering. □

Regarding performance, if there are no faulty nodes, then the sorting tasks are shared evenly among the N nodes. However, depending on the fault-situation, i.e. which nodes are faulty in the system, different nodes may have different shares of the sorting tasks.

V. Evaluation

In this section we describe the implementation using MPI/ULFM of the proposed fault-tolerant version of the Bitonic Sort parallel algorithm and the results obtained.

A. Implementation

The algorithm was implemented using ULFM constructors to handle faults. It is worth mentioning that by default ULFM fault detection is local, in the sense that a fault is detected as a node tries to communicate with another node that has become faulty. From this point on in this section we use the term "node" corresponding to an MPI process. Thus, it is only possible to determine that a node has become faulty by trying to communicate with that node. After a node detects some fault, the ULFM also provides means for that node to communicate the information to the remaining nodes, as described below.

Function `FaultDetection()` is executed in the beginning of each round so that all faulty nodes can be detected. Initially, this function invokes primitive `MPI_Barrier` to synchronize all correct nodes, which communicate to check if there are new faults. If there is at least one faulty node `MPI_Barrier` returns an error which can be either `MPI_ERR_PROC_FAILED` or `MPI_ERR_REVOKED`. Next, all fault-free nodes execute a function to agree on the set of faulty nodes: `MPI_Comm_agree()`. In case some node is faulty, this function notifies all nodes that the MPI communicator is invalid. Next, the communicator is revoked by running primitive `MPI_Comm_revoke()`.

Next, primitives `MPI_Comm_failure_ack()` and `MPI_Comm_failure_get_ acked()` are invoked to identify which nodes within the communicator are faulty. After that, function `MPI_Comm_shrink()` creates a new MPI communicator, removing all faulty nodes. Another step is then executed, which allows the nodes in the new

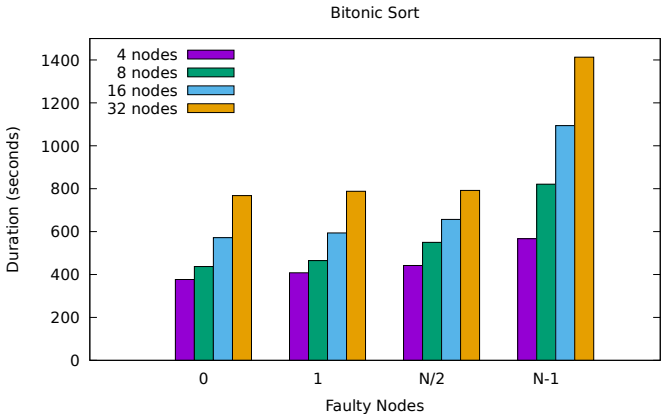

Fig. 4. Results measured for the fault-tolerant version of Bitonic Sort.

communicator to keep the same identifier (rank) they had before the failure.

Each node keeps locally an array with the state of all nodes (faulty or fault-free), which is update by function `FaultDetection()`. This array is employed by the fault-tolerant parallel sorting algorithms presented in Section IV to determine which nodes are fault-free.

The algorithm was evaluated in four different scenarios. In the first scenario all nodes are correct and remain so. In the second scenario a single node fails. In the third scenario half the nodes fail. In the last scenario $n-1$ nodes fail. In order to evaluate scenarios with faults, function `FaultInject()` was employed. This function makes a random selection of nodes that will become faulty in each round. The function receives as input the number of nodes to fail, and determines the round as well as which nodes will fail randomly. Thus in principle any node can fail at any round. A node is caused to crash through the execution of the `SIGKILL` signal.

The algorithm was implemented in the C language using Open MPI and the ULFM 2.0 library [51]. The source code is available at: https://bitbucket.org/etcamargo/parallelsorting/. The experiments were executed on a machine with 32 *Intel Core i7* processors running the Linux operating system (Kernel 4.4.0).

Results obtained for the fault-tolerant version of the Bitonic sort parallel sorting algorithm are described next. It is important to mention that the purpose here is not to increase the speedup of the algorithm in comparison with existing versions, but to confirm that they are robust and keep on executing even after a massive (up to $N-1$ out of N) number of nodes fail at runtime.

B. Results

Each experiment consisted of sorting 1 billion of randomly generated integers. The total number of nodes N varied from 4, 8, 16 up to 32 nodes. Each experiment was repeated 10 times, results presented are averages.

Figure 4 shows the results obtained for Bitonic Sort. It can be seen that the execution time of Bitonic Sort increases as the number of processes grows in all scenarios, both in the fault-free and with faulty nodes. To illustrate this fact, consider the four scenarios with $N = 8$ nodes and with 0, 1, $N/2$ and $N-1$ faulty nodes; Bitonic Sort takes 437s, 465s, 550s and 821s, respectively, to sort 2^{30} integers. Furthermore, it is also possible to observe that as the number of nodes increases, the execution times do not dimish. This situation is probably due to the fact that Bitonic Sort relies heavily on having nodes exchange sequences of elements. Thus, as the number of processes grow, more sorting rounds are required with a corresponding increase of the number of messages exchanged. As mentioned in [52], the predictability of Bitonic Sort can be one of its disadvantages: the join and swap operations take more and more time as the hypercube size increases.

VI. Conclusions

In this work we introduced a novel Algorithm-Based Fault Tolerance technique applied to the Bitonic Sort hypercube-based parallel algorithm. The technique relies on features of the underlying topology, which is assumed to be a VCube – if all nodes are fault-free they communicate according to a logical hypercube, if there are faults, the topology reconfigures autonomically at runtime, preserving several logarithmic properties. The technique is based on defining covers which are nodes that execute the tasks of those that are faulty, as well as defining the communication pattern under faults according to the VCube.

The fault-tolerant version of the Bitonic Sort parallel algorithm was specified and implemented in MPI/ULFM and executed on 4, 8, 16 and 32 nodes. Results were obtained for the algorithm under four different scenarios: fault-free, 1 single fault, half the nodes are faulty, and all but a single node is fault-free. The algorithm executed correctly on all experiments, i.e. being able to detect and survive the occurrence of faults not loosing any of the work that was done before faults occurred.

Future work includes the application of the proposed ABFT strategy to other parallel hypercube-based algorithms. We also do believe several other types of algorithms besides those for parallel sorting can benefit from the ability to detect/reconfigure/continue their executions despite the occurrence of faults at runtime. Another related future work is to develop similar ABFT techniques for parallel algorithms based on other topologies – exploring features of the topology to allow fast and reliable algorithm reconfiguration is certainly a promising field of work. The development of similar techniques for the shared memory paradigm is also relevant future work, as there are several mission-critical systems and applications that take a long time to execute based on that important and popular paradigm.

Acknowledgements

This work was partially supported by the Brazilian Ministry of Education (CAPES) Finance Code 001 and Brazilian Research Council (CNPq) grant 308959/2020-5.

REFERENCES

[1] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sîrbu, A. Bartolini, and A. Borghesi, "A machine learning approach to online fault classification in hpc systems," *Future Generation Computer Systems*, vol. 110, pp. 1009 – 1022, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1932045X

[2] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 610–621.

[3] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Commun. ACM*, vol. 58, no. 7, p. 56–68, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2699414

[4] M. A. Al-Hashimi, O. A. Abulnaja, M. E. Saleh, and M. J. Ikram, "Evaluating power and energy efficiency of bitonic mergesort on graphics processing unit," *IEEE Access*, vol. 5, pp. 16 429–16 440, 2017.

[5] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[6] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[7] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, p. 375–408, Sep. 2002. [Online]. Available: https://doi.org/10.1145/568522.568525

[8] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "Using group replication for resilience on exascale systems," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 210–224, 2014. [Online]. Available: https://doi.org/10.1177/1094342013505348

[9] S. Filiposka, A. Mishev, and K. Gilly, "Multidimensional hierarchical VM migration management for HPC cloud environments," *J. Supercomput.*, vol. 75, no. 8, pp. 5324–5346, 2019. [Online]. Available: https://doi.org/10.1007/s11227-019-02799-5

[10] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, June 1984.

[11] Z. Chen and D. Jack, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Trans. Parallel Distrib. Syst.*, pp. 1628–1641, 2008.

[12] J. Hursey and R. L. Graham, "Building a fault tolerant MPI application: A ring communication example," in *IPDPS Workshops*, 2011, pp. 1549–1556.

[13] N. Bagherpour, S. Hammarling, N. Higham, J. Dongarra, and M. Zounon, "D6.6 - algorithm-based fault tolerance techniques," Oct. 31 2017. [Online]. Available: https://www.nlafet.eu/wp-content/uploads/2016/01/NLAFET-D6.6-171031.pdf

[14] B. Parhami, *Introduction to Parallel Processing: Algorithms and Architectures*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[15] F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Elsevier Science, 2014. [Online]. Available: https://books.google.com.br/books?id=IuY_CQAAQBAJ

[16] I. Foster, I. Foster, and J. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, ser. Literature and Philosophy. Addison-Wesley, 1995. [Online]. Available: https://books.google.com.br/books?id=r5JsQgAACAAJ

[17] E. P. Duarte, L. C. E. Bona, and V. K. Ruoso, "Vcube: A provably scalable distributed diagnosis algorithm," in *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Nov 2014, pp. 17–22.

[18] E. P. Duarte, Jr. and T. Nanya, "A hierarchical adaptive distributed system-level diagnosis algorithm," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 34–45, Jan. 1998.

[19] G. E. Fagg and J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2000, pp. 346–353. [Online]. Available: http://dl.acm.org/citation.cfm?id=648137.746632

[20] W. Bland, A. Bouteiller, T. Hérault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *IJHPCA*, vol. 27, no. 3, pp. 244–254, 2013.

[21] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi, "Fault tolerance of mpi applications in exascale systems: The ulfm solution," *Future Generation Computer Systems*, vol. 106, pp. 467 – 481, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1930860X

[22] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3126908.3126937

[23] D. Tiwari, S. Gupta, and S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *DSN*, 2014, pp. 25–36.

[24] K. B. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. T. Pedretti, R. Brightwell, R. Riesen, and P. G. B. an Dorian Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *SC*, 2011, p. 44.

[25] Z. Hussain, T. Znati, and R. Melhem, "Enhancing reliability-aware speedup modelling via replication," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 528–539.

[26] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: a fault tolerant implementation without checkpointing," in *ICS*, 2011, pp. 162–171.

[27] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *PPoPP*, 2012, pp. 225–234.

[28] J. Hursey and R. L. Graham, "Building a fault tolerant MPI application: A ring communication example," in *IPDPS Workshops*, 2011, pp. 1549–1556.

[29] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas, "Building algorithmically nonstop fault tolerant MPI programs," in *HiPC*, 2011, pp. 1–9.

[30] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers (TOC)*, vol. C-33, no. 7, pp. 518–528, Jun. 1984.

[31] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *IPDPS*, 2006, pp. 10 pp.–.

[32] A. Schöll, C. Braun, M. A. Kochte, and H. Wunderlich, "Efficient algorithm-based fault tolerance for sparse matrix operations," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 251–262.

[33] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi," in *The 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 477–488.

[34] U. Kabir and D. Goswami, "An abft scheme based on communication characteristics," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 515–523.

[35] J. Chen, S. Li, and Z. Chen, "Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2016, pp. 1–2.

[36] S. Roffe and A. D. George, "Evaluation of algorithm-based fault tolerance for machine learning and computer vision under neutron radiation," in *2020 IEEE Aerospace Conference*, 2020, pp. 1–9.

[37] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham, "Algorithm-based fault tolerance on a hypercube multiprocessor," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1132–1145, 1990.

[38] M. H. Durad, M. N. Akhtar, and Irfan-ul-Haq, "Performance analysis of parallel sorting algorithms using mpi," in *2014 12th International Conference on Frontiers of Information Technology*, 2014, pp. 202–207.

[39] E. Camargo and E. Duarte Jr, "Running resilient mpi applications on a dynamic group of recommended processes," *Journal of the Brazilian Computer Society*, vol. 24, 12 2018.

[40] S. White, N. Verosky, and T. Newhall, "A cuda-mpi hybrid bitonic sorting algorithm for gpu clusters," in *2012 41st International Conference on Parallel Processing Workshops*, 2012, pp. 588–589.

[41] T. Alghamdi and G. Alaghband, "High performance parallel sort for shared and distributed memory mimd," in *International Conference on Applied Computing 2019*, 11 2019, pp. 113–122.

[42] R. K., N. N. Chiplunkar, and K. Rajanikanth, "A cpu-gpu cooperative sorting approach," in *2019 Innovations in Power and Advanced Computing Technologies (i-PACT)*, vol. 1, 2019, pp. 1–5.

[43] MPI Forum, "Document for a standard message-passing interface 3.1," University of Tennessee, Tech. Rep., 2015.

[44] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[45] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222–238, 1983.

[46] T. H. Cormer, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introductions to Algorithms'*. The MIT Press, 2009.

[47] M. J. M. J. Quinn, *Parallel programming in C with MPI and OpenMP*. pub-MCGRAW-HILL:adr: McGraw-Hill, 2003.

[48] S. G. Akl, *Parallel Sorting Algorithms*, 1st ed. USA: Academic Press, 1985.

[49] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[50] D. Lee and K. E. Batcher, "On sorting multiple bitonic sequences," in *1994 International Conference on Parallel Processing Vol. 1*, vol. 1, Aug 1994, pp. 121–125.

[51] MPI-Forum, "User-level failure mitigation," https://bitbucket.org/icldistcomp/ulfm2/src/ulfm/, 2020, acessado em 26/09/2020.

[52] Y. Lan and M. A. Mohamed, "Parallel quicksort in hypercubes," in *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*, ser. SAC '92. New York, NY, USA: ACM, 1992, pp. 740–746. [Online]. Available: http://doi.acm.org/10.1145/130069.130085