Algorithm-based fault-tolerant parallel sorting

Edson T. Camargo*

Federal University of Technology – Parana (UTFPR),
Toledo, PR, Brazil
Email: edson@utfpr.edu.br
*Corresponding author

Elias P. Duarte Junior

Department of Informatics,
Federal University of Parana (UFPR),
Curitiba, PR, Brazil
Email: elias@inf.ufpr.br

Abstract: High performance computing (HPC) systems often require substantial resources, and can take up to several hours or days to execute. Upon a failure, it is important to loose as little computation as possible. In this work we present an algorithm-based fault tolerance (ABFT) strategy for hypercube-based parallel algorithms. The strategy assumes the virtual VCube topology, which has several logarithmic properties that are preserved even as nodes fail. The 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 three parallel sorting algorithms: HyperQuickSort, QuickMerge and Bitonic Sort. The algorithms were implemented in MPI using ULMF to handle faults. Experimental results are presented showing the performance and robustness of the solution for sorting up to a billion integers in scenarios with faults.

Keywords: high performance computing; HPC; algorithm-based fault tolerance; ABFT; user level failure mitigation; ULFM; fault tolerance.

Reference to this paper should be made as follows: Camargo, E.T. and Duarte Jr., E.P. (xxxx) 'Algorithm-based fault-tolerant parallel sorting', *Int. J. Critical Computer-Based Systems*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Edson T. Camargo is an Associate Professor at the Federal University of Technology – Paraná (UTFPR), Toledo, Brazil. He received his PhD in Computer Science from the Federal University of Parana (UFPR) in 2017. During his PhD, he spent one year as a PhD student at the Universitá della Svizzera Italiana (USI). He is a member of the Brazilian Computer Society where served as the Chair of the Special Committee on Fault Tolerant Systems (CE-TF). His research interests include computer networks, parallel computing and distributed systems, their dependability and algorithms.

Elias P. Duarte Junior is a Full Professor at the Federal University of Parana, Brazil. His research interests include computer networks and distributed systems, their dependability and algorithms. With over 300 peer-reviewer papers, and 130 students supervised, he is an associate editor of the *Computing* (Springer) journal and *IEEE Transactions on Dependable and Secure Computing*, and also has served as the Chair of more than 25 conferences and workshops, including TPC Chair of GLOBECOM'2024, SRDS'2018 and ICDCS'2021. He chaired the Brazilian National Laboratory on Computer Networks from 2012–2016, a member of the Brazilian Computing Society and a senior member of the IEEE.

This paper is a revised and expanded version of a paper entitled 'An algorithm-based fault tolerance strategy for the Bitonic Sort parallel algorithm' presented at the 10th IEEE Latin American Symposium on Dependable Computing (LADC), Florianópolis, Brazil, 22–25 November 2021.

1 Introduction

High performance computing (HPC) systems are based on parallel algorithms which solve hard problems across multiple fields. Those systems run on several processors and cores, and present high performance, being capable of executing up to 10^{15} (peta-scale) and 10^{18} (exascale) floating point operations per second (FLOPS). On the other hand, those very large-scale systems usually present a small mean time between failures (MTBF) (Netti et al., 2020). For instance, for the Blue Waters system – which is a peta-scale machine – the average MTBF is of only 4.2 hours (Martino et al., 2014). Further reductions are expected for exascale systems (Reed and Dongarra, 2015; Al-Hashimi et al., 2017). Faults are a serious problem, as those systems usually take a long time to execute, and have very high demands in terms of computational resources and energy. It is thus imperative to design those systems to be fault-tolerant, in the sense that they can continue executing as expected after faults occur, and as little work as possible is lost.

There are several techniques to make an HPC system fault-tolerant (Egwutuoha et al., 2013; Herault and Robert, 2015). Those techniques include, among others, rollback recovery (Elnozahy et al., 2002), replication (Bougeret et al., 2014), computation migration (Filiposka et al., 2019), and algorithm-based fault tolerance (ABFT). ABFT relies on properties of the parallel algorithm itself to tolerate faults during its execution (Huang and Abraham, 1984; Chen and Jack, 2008; Hursey and Graham, 2011; Bagherpour et al., 2017). The algorithm is designed to be robust, and must be able to detect or receive fault notifications and adapt itself after faults occur.

This work proposes 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. A parallel hypercube algorithm enhanced with the proposed technique is able to reconfigure itself autonomously in run time, guaranteeing that it does not halt after faults are detected: fault-free nodes continue the execution and as little work as possible is lost.

Hypercubes are scalable topologies by definition, and have been extensively used in parallel computing both as interconnection networks (Parhami, 1999) and to organise the communication and execution of parallel and distributed algorithms (Leighton, 2014; Foster et al., 1995). 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 (Duarte et al., 2014; Duarte and Nanya, 1998). 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 organised 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 can fail by crashing, and correct processes identify which processes are faulty. In a system of N nodes, even if up $N-1$ nodes become faulty the system autonomously reconfigures itself at runtime, and is thus able to continue the execution.

The heart of the proposed ABFT strategy 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. However, 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. Fault-tolerant versions of three parallel sorting algorithms are presented and compared: HyperQuickSort, QuickMerge and Bitonic Sort.

All three algorithms were implemented with the message passing interface (MPI) (Fagg and Dongarra, 2000) and user level failure mitigation (ULFM) (Bland et al., 2013; Losada et al., 2020) to handle faults. MPI is one of the most relevant 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. Fault tolerance is achieved with ULFM, which was proposed by the MPI Forum to avoid having to completely halt 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 basically used for failure detection. Experimental results are presented showing the performance and robustness of the proposed fault-tolerant algorithms.

The rest of this work is organised as follows. Section 2 describes related work. Section 3 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 4. The implementations of the fault-tolerant parallel sorting algorithms and experimental results are presented in Section 5. Conclusions follow in Section 6.

2 Related work

This section presents related work in four different fields. The first is fault-tolerant HPC systems, next ABFT, then fault-tolerance strategies for MPI systems, and finally parallel sorting.

The development of techniques to improve the resiliency of HPC systems is often preceded by the investigation of system vulnerabilities. In an extensive field work published in 2017 (Gupta et al., 2017), 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 make disturbing observations, for instance whether recently developed 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 normalised 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, but that number will probably be higher. In another work (Martino et al., 2014), field data is presented for the reliability of the Blue Waters peta-scale 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 (Egwutuoha et al., 2013). This technique consists of establishing checkpoints to which the system can roll back in case of failures, instead of restarting from the very beginning (Elnozahy et al., 2002). It is a challenge to apply rollback recovery to HPC systems that take a very long time to complete its execution and have a low MTBF. For instance, Tiwari et al. (2014) 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 checkpointing incurs on high I/O overhead. Having a larger checkpointing interval can reduce the overhead, 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 to checkpointing is to employ replication instead of rollback recovery. Ferreira et al. (2011) justify the use of replication given the short mean time to interrupt (MTTI), which corresponds to the time spent taking checkpoints. The authors also mention that replication can be extended to deal with Byzantine faults. Hussain et al. (2020) 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.

ABFT relies on the properties of the parallel algorithm itself to recover from faults during its execution (Davies et al., 2011; Du et al., 2012; Hursey and Graham, 2011; Wang et al., 2011). 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 (1984) to detect and correct errors in the context of matrix operations, caused by transient or permanent hardware

faults. 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 that relationship, the authors designed an ABFT technique to detect, locate and correct certain types of errors of matrix operations.

Another ABFT technique applied to matrix operations was proposed by Chen and Dongarra also in the context of fault-tolerant HPC systems (Chen and Dongarra, 2006; Chen and Jack, 2008). That technique assumes the fail-stop model and allows the HPC system 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. (2011) proposed the ABFT-hot-replacement strategy to prevent correct nodes from having to stop and wait for nodes with faulty data to recover. Spare nodes which are redundant nodes can replace those that became faulty. That work is also based on the *checksum* relationship of matrix operations, and was implemented with an MPICH system adapted to deal with application-level faults. Schöll et al. (2016) refine techniques based on ABFT for operations on sparse matrices, the main purpose is to reduce fault localisation time.

Checkpoint-on-failure (CoF) (Bland et al., 2012) 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 recovers the checkpoints of the correct nodes, and use an ABFT technique to recover data from faulty nodes.

Kabir and Goswami (2016) proposed an ABFT scheme that can be applied to an entire class of applications – instead of a specific application. The authors mention that their purpose is to overcome one of the disadvantages of ABFT, called ‘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.

Chen et al. (2016) 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 (Roffe and George, 2020).

In a recent work, Kosaian and Rashmi (2021) have proposed a fault-tolerant neural network based on ABFT techniques. Instead of having a single strategy that is always applied, the strategy selects the most efficient approach to each neural network layer. The choice is done depending on whether compute or memory-bandwidth resources are the limiting factor. Another ABFT strategy has been proposed for convolutional neural networks (Zhao et al., 2020). The focus of that strategy is on the protection of the inference process against soft errors, and it is based on checksums.

Yet another recent application of ABFT has been proposed for image compression (Bao and Zhang, 2020) in the context of the in the JPEG2000 Standard. A fault-tolerant strategy is presented to implement the forward discrete wavelet transform (FDWT) on

GPUs. Using image matrix descriptions, the propagation of errors across a multi-level FDWT is proposed. By analysing the relationship between the data and silent data corruption (SDC) errors, an on-line fast error detection and correction method is proposed.

Zhu et al. (2020) observe that it is possible to implement ABFT techniques as libraries (e.g., libraries for numerical algorithms) in a way to make it application-independent. They propose a strategy called FT-PBLAS, a library for fault-tolerant parallel linear algebra computations, because it provides a series of fault-tolerant modules. The strategy supports error detection and recovery mechanisms based on a block-checksum approach.

Finally, closest to our work is the ABFT strategy applied to a hypercube-based parallel computer (Banerjee et al., 1990). 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 typical HPC network topologies.

MPI is the *de facto* standard for developing parallel distributed memory applications (MPI Forum, 2015; Fagg and Dongarra, 2000). 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 (Bland et al., 2013; Losada et al., 2020) 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 (Bland et al., 2013). Rocco et al. (2022) present Legio, an alternative framework to build parallel MPI applications fault-tolerant. The authors claim that, in comparison with ULFM, Legio is easier to use and show that the overhead is negligible.

Related work on parallel sorting algorithms includes a few works reporting the implementation of the algorithms with MPI (Durad et al., 2014; Camargo and Duarte, 2018). There are also implementations on hybrid systems, such as Bitonic Sort on CUDA and MPI (White et al., 2012) and others that employ both distributed and shared memory (Alghamdi and Alaghband, 2019; Raju et al., 2019). More details are given in Section 4.

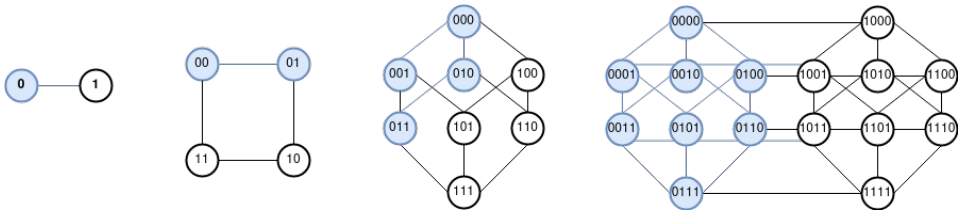
3 The proposed strategy

In this section we present the proposed ABFT strategy, which can be used to build a fault-tolerant version of any parallel algorithm based on the hypercube. Processes monitor each other and, upon detecting failures, the fault-free processes share among themselves the tasks of the failed processes. In this section, we first give the definition of a hypercube, an overview of the VCube virtual topologogy, and then describe the proposed ABFT strategy.

A hypercube (Kumar et al., 2002; Leighton, 2014; Parhami, 1999) 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.

Figure 1 The construction of d-dimensional hypercube from two $d-1$-dimensional hypercubes, d = 2, 3, 4 (see online version for colours)



A hypercube is a regular graph with degree $\log N$ (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 $\log N$.

Table 1 $C_{i,s}$ for a system with eight 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

The strategy proposed in this work is based on the VCube virtual topology (Duarte et al., 2014; Duarte and Nanya, 1998). 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 organised in clusters, 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 $d = \log N$, 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, where $\oplus$ stands for the exclusive or (XOR) operation:

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

Table 1 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 neighbours in testing intervals, which are periodic intervals of time determined with the local clock of each node. Global synchronisation 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 (Schlichting and Schneider, 1983). At each testing round a fault-free node i tests its clusters $c(i, s)$, s = 1, ..., log N. 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 Duarte et al. (2014) that using this strategy all fault-free nodes learn about any new event in at most $logN$ rounds, in average much faster than that. Furthermore, using the test assignment proposed, at most $N \log N$ tests are executed per testing interval.

According to 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 , 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.

4 Fault-tolerant parallel sorting

This section presents the application of the fault tolerance technique presented in the previous section to parallel sorting algorithms. Sorting is one of the most fundamental and well studied computing problems (Cormer et al., 2009). 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 $\forall i | 0 < i < N : a'_i < a'_{i+1}$. We say that L' is sorted in increasing order. Although the elements can also be sorted in decreasing order, i.e., $\forall i | 0 \leq i < N : a_i > a_{i+1}$, 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 (Quinn, 2003; Durad et al., 2014), 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 (Akl, 1985; Kumar et al., 2002). These algorithms have been designed for all types of parallel computing architectures and topologies, including physical and logical topologies that organise processing nodes in a meshes, rings, stars, hypercubes, among others.

Next we show how to make HyperQuickSort, QuickMerge and Bitonic Sort fault-tolerant with the proposed ABFT strategy. Recall that 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 algorithms run 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).

4.1 The fault-tolerant HyperQuickSort algorithm

The HyperQuickSort algorithm (Wagar, 1987) receives as input a list of elements $L = \{a_0, a_1, ..., a_{k-1}\}$ which are sorted by a set of hypercube nodes, $H = \{0, 1, ..., (N-1)\}$, $|H| = N$ is a power of 2. At first the algorithm splits the $|L|$ elements into equal sized sublists L_i which are assigned to the $|H|$ processes. The sorting algorithm executes in rounds: in each round, each node i is responsible for sorting the $\frac{|L|}{N}$ elements of its assigned list using sequential QuickSort. Node i then exchanges with node j a sublist of its sorted elements based on a *pivot*. The algorithm is described in detail below. HyperQuickSort takes $\log N$ rounds to complete sorting. At the end of these sorting rounds, the largest element of the list maintained by node i is less than or equal to the lowest element of the list of node $i+1$, $0 \leq i \leq |P|-2$. The pseudocode of the fault-tolerant version of HyperQuickSort is shown as Algorithm 1.

The algorithm works as follows. At the beginning of each round, node i must determine which other nodes it is covering in the round (line 6). Set I is used to keep the identifiers with which node i runs the algorithm. Initially I is set with i (line 4), and if all nodes are fault-free than there will be no more elements. However, given the set of faulty nodes (line 5), node i must check which of those nodes it is covering. Those nodes are added to set I (line 7). Node i then runs the round in parallel for all nodes whose ids are in i, i.e., itself and the nodes it is covering. After obtaining the list to be sorted (line 9), the sequential version of QuickSort is executed locally on the list (line 10). The algorithm runs in $d = \log N$ sorting rounds, in each round a different dimension of the hypercube is considered, starting from the highest dimension d in the first round, down to 1-dimensional hypercubes in the last. In each round, the fault-free node with the lowest identifier in the hypercube of the dimension being considered is called the *root* (line 11). For example, consider a 3-dimensional hypercube in which all nodes are fault-free. In the first round, the root is node 0, which has the lowest id of the hypercube with eight nodes (d = 3). In the next round, the two roots are node 0 and node 4, computed for the two 2-dimensional hypercubes, of four nodes each (d = 2). In the last round the four roots are nodes 0, 2, 4, and 6, which are computed in the four 1-dimensional hypercubes of two nodes each (d = 1).

Algorithm 1 Fault-rolerant HyperQuickSort (executed by node i)

```
1: Begin
2:   d ← log N                                              {Dimension of the hypercube}
3:   while d > 0 do
4:     I ← i                          {Set I of identifiers to run the algorithm is initialised 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 ← L_k                                         {The sublist assigned to node k}
10:      Quicksort(list)                                    {QuickSort is executed locally}
11:      root ← fault-free node with the smallest identifier in the d-dimensional hypercube to
         which node k belongs
12:      if k = root then
13:        pivot ← median(list)
14:        broadcast(pivot, d)        {the root broadcasts the pivot to all c(root, s), s = 1, ..., d}
15:      create_lists(higher_list, lower_list, list, pivot)
16:      partner ← first fault-free node in c(k, d)
17:      if k > partner then
18:        send(lower_list, partner)
19:        receive(new_higher_list, partner)
20:        list ← union(higher_list, new_higher_list)
21:      else if k < partner then
22:        send(higher_list, partner)
23:        receive(new_lower_list, partner)
24:        list ← union(lower_list, new_lower_list)
25:    F' ← set of faulty nodes in the end of this round
26:    if F = F' then
27:      d ← d − 1
28:  QuickSort(list)                                        {QuickSort is executed locally at node k}
   End
```

The root computes and broadcasts a pivot to the other processes in its clusters (lines 12–14). The pivot is employed to split a list into two sublists, one of which has all elements lower than all elements of the other list. In the pseudocode, the pivot is the median element of the list (line 13). After receiving the pivot from the root, each node splits its local list in two (line 15).

In each round, the node communicates with a partner, which is the first fault-free node of the VCube clusters described in Section 3. Thus the partners of node k are the first fault free nodes of clusters $c(k,1)$, ..., $c(k,d)$. In the first round each node communicates with its partner in the largest cluster ($c(k,d)$) (line 16). Then, the cluster size decreases (line 27). Figure 2 shows the cluster sizes for a 3-dimensional hypercube. In the first round, the partner of node k is the first fault-free node in cluster $c(k,3)$; in the next round it is the first fault-free node is cluster $c(k,2)$, finally in the last round it is the the first fault-free node is cluster $c(k,1)$.

Now consider a pair consisting of a node and its partner. The node with the lowest id sends the list with elements greater than the pivot to the partner which obviously has the highest id, and receives the list of elements lower than the pivot from the partner (lines 17–24). After these lists are exchanged, each node concatenates the received list

with the sublist that had been retained (line 20 and 24). Finally, each node sorts its new list locally with QuickSort (line 10).

Figure 2 Hypercube with 3, 2 and 1 dimensions and the respective roots

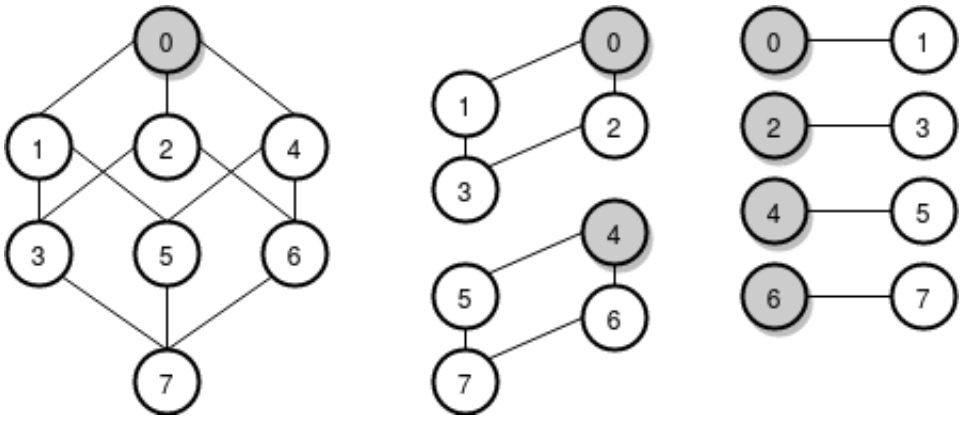


Figure 2 shows an example execution of HyperQuickSort for eight fault-free nodes. It takes three rounds to complete sorting these elements. In the first round, the whole system which corresponds to a 3-dimensional hypercube (d = 3) is considered, and node 0 is the root, the node with the smallest id. In this round, the following pairs of nodes are established according to function $c(i, 3)$: (0, 4), (1, 5), (2, 6) and (3, 7). These pairs exchange and sort their lists according to the pivot received from the root. In the second sorting round (d = 2), nodes 0 and 4 are the roots, the nodes with the smallest identifiers in the two 2-dimensional hypercubes. Each root broadcasts its pivot to the nodes of its clusters. Now the following pairs of nodes are computed with function $c(i, 2)$ to exchange and sort their lists: (0, 2), (1, 3), (4, 6) and (5, 7). Finally, in the third round, the whole process is repeated considering 1-dimensional hypercubes (d = 1), the following pairs of nodes are computed with function $c(i, 1)$ to exchange and sort their lists: (0, 1), (2, 3), (4, 5) and (6, 7). The original list is now completely sorted.

Our ABFT strategy makes HyperQuickSort autonomously fault-tolerant as described next. Nodes have access to a list of faulty nodes that VCube provides. At the beginning of each sorting round, each process k sets its partner as the first fault-free node in $c(k, d)$. This is a main difference to the regular HyperQuickSort algorithm, in which partners are fixed and if a partner is faulty it is not replaced. As mentioned above, in our strategy whenever a node is faulty, it will be covered by another node that becomes responsible executing the tasks of the faulty node. As an example, consider a 3-dimensional hypercube and the first sorting round. Node 0's partner is the first fault-free process in $c(0, 3)$ = 4, 5, 6, 7. If process 4 is fault-free, then 0 and 4 are partners and exchange their lists according to lines 17–24 in Algorithm 1 (assuming 0 also is fault-free). However, if 4 is faulty, then 0's partner is node 5. If all these nodes are faulty, i.e., 5, 6 and 7 are faulty, then 0 does not have a partner in the this sorting round. It simply sorts its local list.

Now we prove that the fault-tolerant HyperQuickSort algorithm completes in $\log N$ rounds even if up to $N - 1$ nodes fail during the execution.

Theorem 1: The fault-tolerant HyperQuickSort algorithm completes sorting in $\log N$ rounds even if $N - 1$ nodes become faulty during those rounds.

The proof is done by induction on the hypercube 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 hypercube of dimension $d = r$ sorting is completed in r rounds, even if up to $2^r - 1$ nodes become faulty.
- Induction step: A hypercube of dimension $d = r + 1$ consists of two hypercubes of dimension r. According to the induction step, in each of those two hypercubes sorting completes in v rounds. Now in the first sorting round of HyperQuickSort, a fault-free node in one of those r-dimensional hypercubes will find a fault-free partner in the other r-dimensional hypercube, if there is one. They will act as covers for the faulty nodes within their r-dimensional hypercubes and communicate with the partner of the other v-dimensional hypercube in that round. If all nodes are faulty in one of those r-dimensional hypercubes, the cover(s) will be on the other r-dimensional hypercube 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.

4.2 Fault-tolerant QuickMerge

QuickMerge is parallel sorting algorithm proposed by Quinn (1989). As HyperQuickSort does, QuickMerge distributes equally an input list L of elements to be sorted among a set of nodes of hypercube H, $|H| = N$ is a power of 2. Also, each node i locally sorts a sublist L_i in a sorting round d and then splits the sublist using a pivot, and exchange part of the sublist with a partner. The difference between QuickMerge and HyperQuickSort is on lines 13–17 of Algorithm 2. Node i, the first fault-free node with lowest identifier, generates a list of elements which are called *splitters*, and broadcasts (line 17) the list to *all* nodes. Note that this list (called $splitter[]$ in the algorithm) is generated once, before sorting actually begins. The list contains $2^d - 1$ elements, in which are all the pivots employed in all rounds. For example, considering a 3-dimensional hypercube and a sublist with $|L|/8$ = 1,024 elements, $splitter[]$ will contain the seven elements in positions 128, 256, 384, 512, 640, 768 and 896 of the sublist. It is important to note that the elements of the $splitter[]$ array are in ascending order.

Algorithm 2 Fault-tolerant QuickMerge (executed by node i)

```
1: Begin
2:   d ← log N                                          {Dimension of the hypercube}
3:   r ← log N                                          {Round index}
4:   q ← 0                                              {increase each round helping to select the splitter}
5:   while r > 0 do
6:     I ← i                                            {Set I of identifiers to run the algorithm is initialised 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 ← L_k                                     {The sublist assigned to node k}
12:      Quicksort(list)                                {QuickSort is executed locally at node i}
13:      if r = d (this is the first round) then
14:        if k = fault-free node with the smallest identifier in the d-dimensional hypercube
           (root) then
15:          for v ← 1 to N − 1 (Create the list of pivots) do
16:            splitter[v] ← list[(v ∗ size(list))/N]
17:        broadcast(splitter[])                        {the root broadcasts the list of pivots to all nodes}
18:      partner ← first fault-free node in c(i, r)
19:      l ← k ⊙ (2^d − 2^(d−q)) ⊕ 2^(d−q−1)           {operations bit a bit: and, or respectively}
20:      create_lists(higher_list, lower_list, list, splitter[l])
21:      if i > partner then
22:        send(lower_list, partner)
23:        receive(new_higher_list, partner)
24:        list ← union(higher_list, new_higher_list)
25:      else if i < partner then
26:        send(higher_list, partner)
27:        receive(new_lower_list, partner)
28:        list ← union(lower_list, new_lower_list)
29:    F′ ← set of faulty nodes in the end of this round
30:    if F = F′ then
31:      r ← r − 1
32:      q = q + 1
33:  QuickSort(list)                                    {QuickSort is executed locally at node k}
   End
```

Although $2^d - 1$ pivots are chosen and disseminated (lines 13 to 17), only d of them are employed by each process. In each round, pairs of processes are formed to exchange sublists (line 18). Then, according to its identifier, each node chooses one of the elements of $splitter[]$ as pivot (line 19) to split its list in two: a list of elements larger than the pivot and the other with elements smaller than the pivot (line 20). In lines 21 to 28 a pair of nodes i and j exchange sublists: if $i > j$ node i sends the sublist with elements smaller than the pivot to node j and receives from j the list with elements larger than the pivot. Node i concatenates the local sublist with elements larger than the pivot with the sublist received from j (lines 24 and 28) and reorders the resulting list (line 12). Thus, at the beginning of each round, node i has an ordered list of elements larger than the pivot. In turn, j has an ordered list of elements smaller than or equal to the pivot.

We actually detected a bug in the specification of the algorithm in the original paper by Quinn (1989), the bug is in the selection of the pivots, as described below.

In order to guarantee that sorting is done correctly, all nodes that form a pair that exchange sublists must adopt the same pivot, so that at the end of a round those nodes have all the elements that are greater than/smaller than or equal to that pivot. For example, in Figure 2, in the first round the clusters are formed in a 3-dimensional hypercube (the whole system) which has eight nodes. At the end of this round, considering that all nodes are fault-free, nodes 0, 1, 2 and 3 have all elements that are smaller than or equal to the pivot, while the nodes with which they form pairs in this round, respectively nodes 4, 5, 6 and have the elements that are greater than the pivot. The bug of the original specification is that it allows different pivots to be chosen by different nodes of a pair. Back to the example, in the second round the clusters are formed in two 2-dimensional hypercubes, the same pivot is employed by all fault-free nodes of each 2-dimensional hypercube, but each of those 2-dimensional hypercubes can use a pivot that is different from the other. Finally, in the third round a single pivot is employed by all pairs of nodes of the clusters formed in the four 1-dimensional hypercubes.

In the present work we fix the bug of the original QuickMerge specification. Our version is shown as Algorithm 2. The main difference to the original specification is in line 19. Considering the same example as before, the following pivots are now selected: all nodes in the first round select $splitter[4]$. In the second round, the processes of the first 2-dimensional hypercube select $splitter[2]$ as the pivot, while the nodes of the second 2-dimensional hypercube select $splitter[6]$. Then, in the last round the nodes of the four 1-dimensional hypercubes select the following pivots: $splitter[1]$, $splitter[3]$, $splitter[5]$ and $splitter[7]$.

As except for the selection of pivots QuickMerge employs exactly the same sorting procedure as HyperQuickSort, Theorem 1 also proofs the correctness of the fault-tolerant version of QuickMerge.

In terms of performance, as stated in Quinn (1989), HyperQuickSort and QuickMerge have different strategies for dividing the elements among the nodes. HyperQuickSort requires more messages, as in each round the corresponding root computes and broadcasts the pivot. Quinn also notes that due to the pivot selection strategy, HyperQuickSort generates a more balanced distribution of elements between the nodes. Thus QuickMerge sends longer messages and always takes more time to sort the same elements in comparison with HyperQuickSort. In order to improve of performance of the algorithm, Quinn (1989) suggests a modified QuickMerge version, in which the $splitter[]$ list of pivots is not anymore computed/broadcast by a single node in the beginning: this is done by all nodes. After a node has received the $splitter[]$ lists from all other fault-free nodes, it computes a list of averages, of which an entry is the average of the corresponding entries of all $splitter[i]$ lists received. In order to modify the version presented as Algorithm 2 all that is required is to remove line 14, and add code to receive and compute the $splitter[]$ list of averages. Note that this version of the algorithm is more expensive than the original version, as it involves all nodes broadcasting messages to all other nodes. In Section 5 we report results from implementations of both these versions, and none is more efficient that HyperQuickSort.

4.3 Bitonic Sort algorithm

The Bitonic Sort algorithm (Batcher, 1968; Lee and Batcher, 1994) 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:

1 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

2 there is a cyclic rotation that satisfies 1.

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 organised 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.

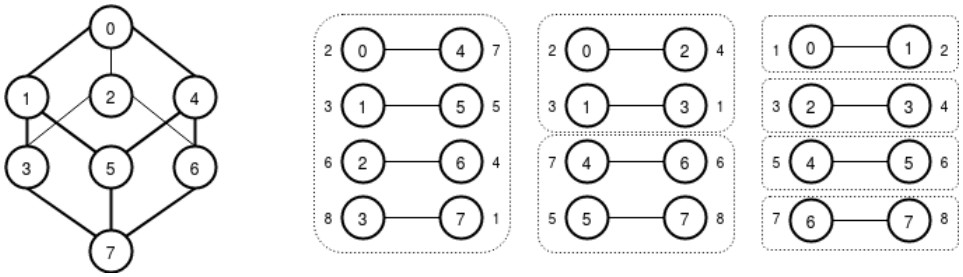
Mapping a bitonic sequence to a hypercube can be done as follows. If the size of the sequence (m) is equal to the number of nodes ($m = N$), then each single element is mapped to a single hypercube node, i.e., element a_i is mapped to node i. If the size of the sequence is greater than the number of 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 3.

Algorithm 3 Fault-tolerant Bitonic Sort with $m = n$ (executed by node i)

```
1:  Begin
2:    d ← log N                                              {VCube dimension}
3:    while d > 0 do
4:      I ← i                {Set I of identifiers to run the algorithm is initialised 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
```

Figure 3 *Bitonic Sort* on a 3-dimensional VCube: first, second and third sorting rounds



Note: Node ids are shown in the circles, the element a node keeps is shown on its side.

The *compare_exchange* procedure (lines 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 eight fault-free nodes. Figure 3 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, 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.

Algorithm 4 Fault-tolerant Bitonic Sort for any sequence (executed by node i)

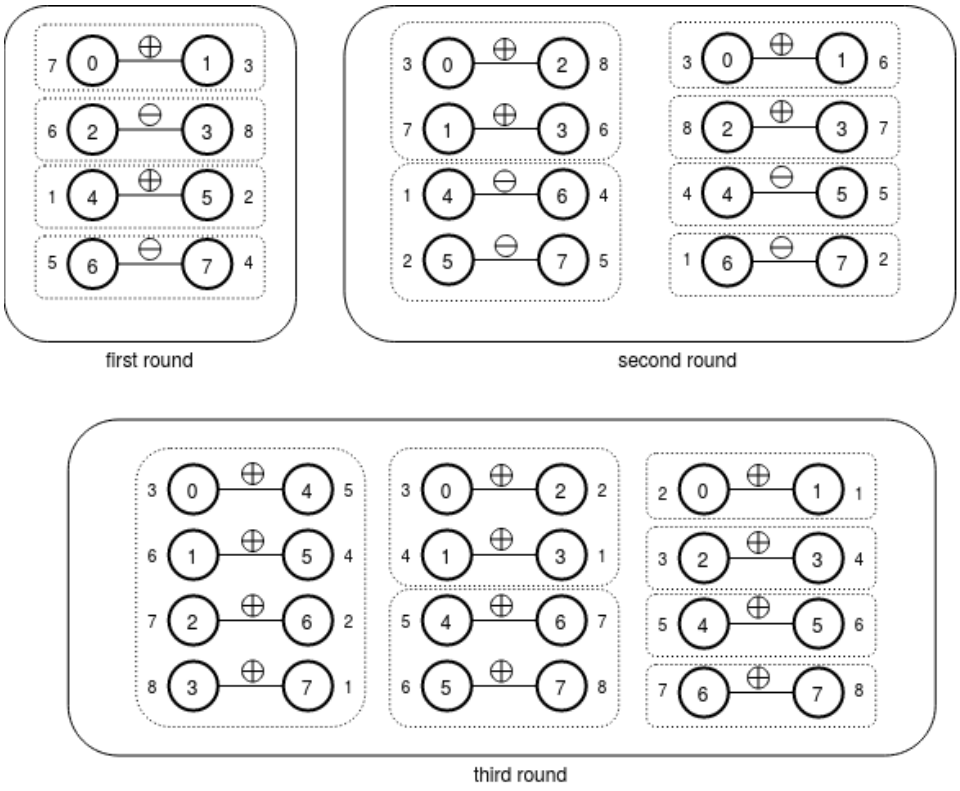
```
1: Begin
2:   d ← log N                                              {VCube dimension}
3:   for s ← 0 to d − 1 do
4:     t ← s
5:     while t ≥ 0 do
6:       I ← i                          {Set I of identifiers to run the algorithm is initialised with i}
7:       F ← set of faulty nodes in the beginning 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
```

The version of Bitonic Sort presented as Algorithm 4 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 4 illustrates the ordering process for the input sequence seq = (7, 3, 6, 8, 1, 2, 5, 4) in a 3-dimensional hypercube. In Figure 4 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 Kumar et al. (2002) to determine whether the node keeps the smallest elements or the largest elements. If the $(s+i)^{\text{th}}$ 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 4). 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.

Figure 4 An example execution of *Bitonic Sort* for an input sequence that is not bitonic on a 3-dimensional hypercube



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 4 is equivalent to Algorithm 3 (see Figures 3 and 4). 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 $\log N$ rounds even if up to $N-1$ nodes fail during the execution.

Theorem 2: The fault-tolerant Bitonic Sort algorithm completes sorting in $\log N$ rounds even if $N - 1$ nodes become faulty during those rounds.

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: 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.

5 Evaluation

In this section we describe the implementation using MPI/ULFM of the proposed fault-tolerant version of the three parallel sorting algorithms and the results obtained.

5.1 Implementation

The algorithms were 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, 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 synchronise 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 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 4 to determine which nodes are fault-free.

The algorithm was implemented in the C language using Open MPI and the ULFM 2.0 Library (MPI Forum, 2020). 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).

The algorithms were 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.

Results obtained for the fault-tolerant versions of the parallel sorting algorithms are described next. It is important to mention that the purpose here is not to increase the speedup of the algorithms 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.

5.2 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.

As mentioned above, four scenarios were evaluated. After showing results for a baseline scenario with

1 no faults.

Three different scenarios varied in terms of the number of nodes that became faulty

2 a single node

3 half the node

4 $N - 1$ nodes.

Figure 5 shows the performance of our fault-tolerant version of HyperQuickSort in the four fault scenarios executed by 4, 8, 16 and 32 nodes to sort 1 billion integers randomly generated. In the first scenario with no faults, the algorithm took approximately 150 seconds to complete when $N = 4$ nodes. As the number of nodes increases, the time to sort decreases progressively to approximately 72 seconds when 32 nodes are used.

Figure 5 Fault-tolerant HyperQuickSort executed by $N = 4, 8, 16, 32$ nodes to sort 2^{30} integers (see online version for colours)

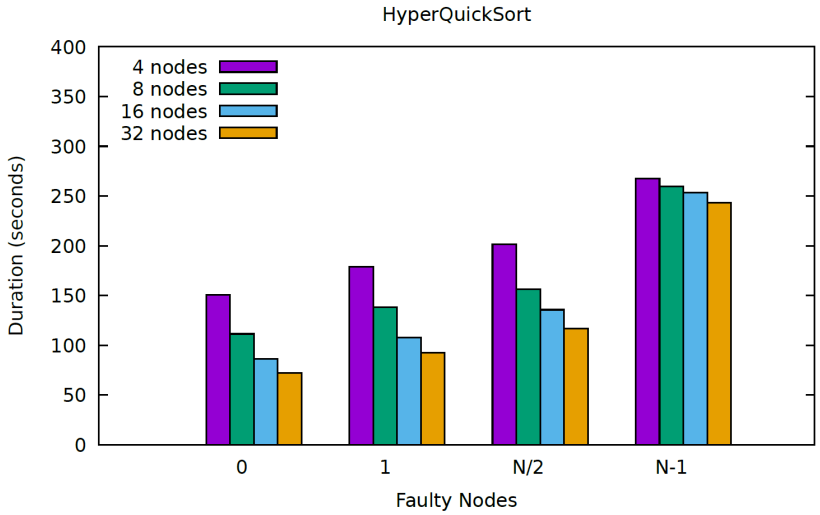


Figure 6 Fault-tolerant QuickMerge executed by $N = 4, 8, 16, 32$ nodes to sort 2^{30} integers (see online version for colours)

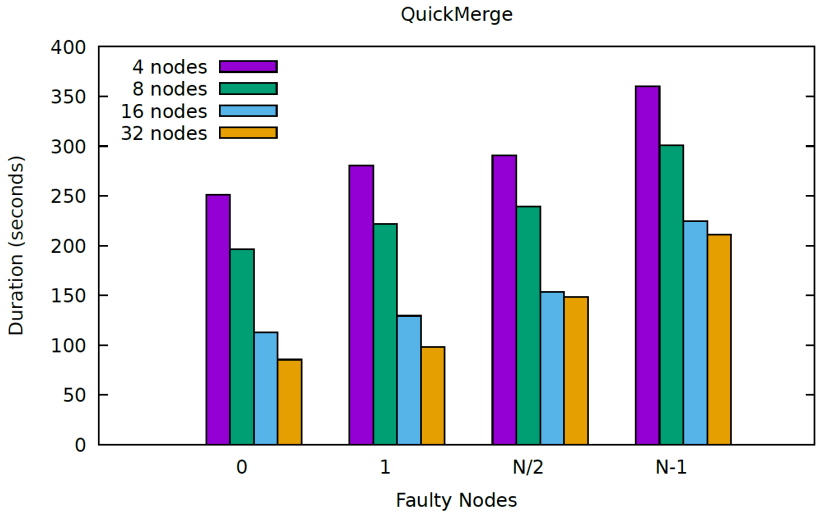


Figure 7 Fault-tolerant modified QuickMerge executed by N = 4, 8, 16, 32 nodes to sort 2^{30} integers (see online version for colours)

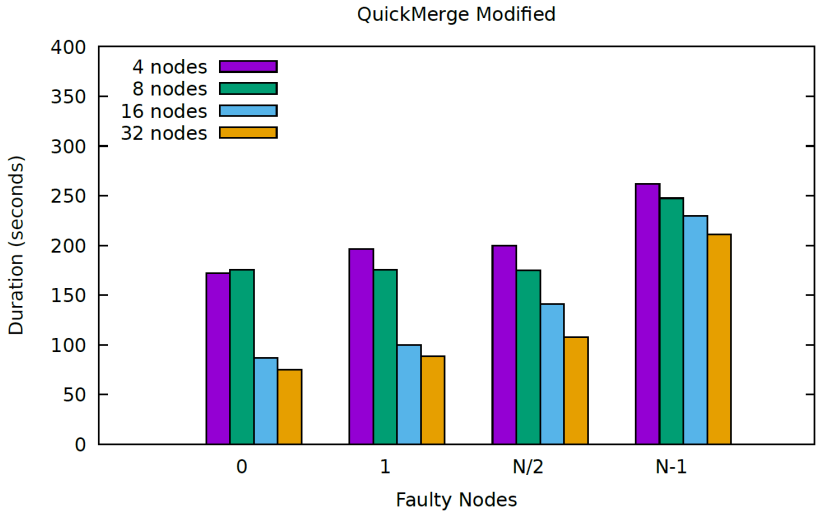
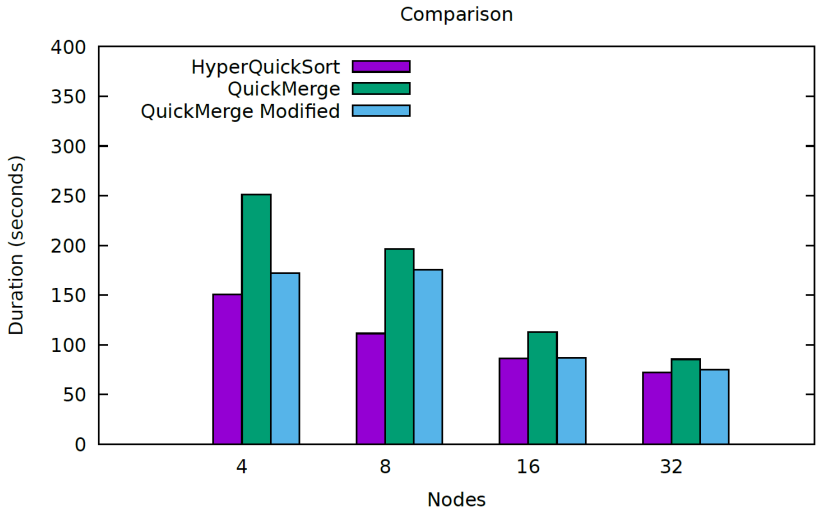


Figure 8 Comparison of HyperQuickSort, QuickMerge and modified QuickMerge in the fault-free scenario (see online version for colours)

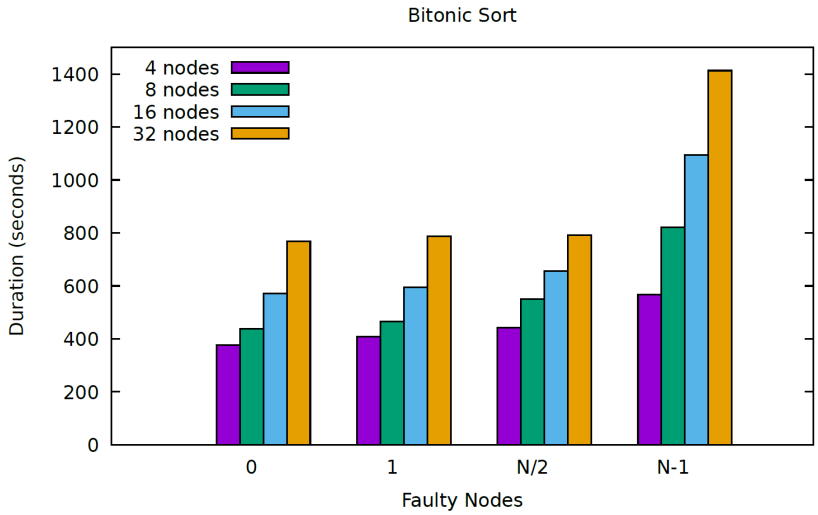


In the scenarios with faults, the execution time of the algorithm also decreases progressively as the total number of nodes used increases. However, the occurrence of faults increases the overall time to run the algorithm, since the correct nodes take over the computation of the faulty nodes. In the scenario with 1 fault, the execution time for N = 4 nodes is approximately 178 seconds, which decreases to approximately 92 seconds as N increases to 32 nodes (1 of which fails). In the third scenario in which half the nodes become faulty during the execution, the execution time is approximately 201 and 116 seconds for 4 and 32 nodes, respectively. In the last scenario, as $N - 1$ nodes are faulty, a single correct node executes until the end of the experiment and the

time to sort measured in all cases is very similar, as expected. It is important to highlight that although the execution time increases as the number of faulty nodes increases, the algorithm continues its execution despite the occurrence of failures, as expected.

Next we report an experiment executed to evaluate the fault-tolerant version of QuickMerge, both the original and modified versions were implemented. In this experiment we also evaluated how well balanced the solution is. We compared the fault-tolerant versions of QuickMerge with HyperQuickSort. The algorithms were executed on 10 different samples of 2^{30} (1 billion) randomly generated integers all in the interval from -2^{31} to $2^{31} - 1$. For each sample each algorithm was executed three times, and averages are presented. The same four different scenarios of the previous experiment were used here, i.e., the number of faulty nodes in each is 0, 1, $N/2$ and $N - 1$.

Figure 9 Results measured for the fault-tolerant version of Bitonic Sort (see online version for colours)



Figures 6 and 7 show the execution time of the fault-tolerant versions of QuickMerge and modified QuickMerge algorithms, respectively. Note that the execution time for 4, 8, 16 and 32 nodes presents a similar pattern to that of HyperQuickSort, i.e., as the number of nodes increases, the execution time decreases. Furthermore, faults increase the execution time. Nevertheless, QuickMerge presents worse results when it is compared to HyperQuickSort (Figure 5). HyperQuickSort presents the lowest (best) execution times in all cases. For example, in the fault-free scenario with 4 nodes, the execution time is approximately 253, 167 and 156 seconds for QuickMerge, modified QuickMerge and HyperQuickSort, respectively. Actually, the difference between the execution times measured for modified QuickMerge and HyperQuickSort is small for 4, 16 and 32 processes, as shown in Figure 8.

The main difference between HyperQuickSort, QuickMerge and modified QuickMerge algorithms is in the selection and distribution of the pivots. Recall that the pivots directly affect the sizes of the sequences of elements exchanged between the nodes with an impact both on the time to sort local sequences and also on the number of

messages produced and the time required to exchange those messages. In other words, the pivot selection strategy has an impact on how well balanced the tasks assigned to the different nodes are, in terms of the sizes of the sequences they get to sort.

The largest difference to the ideal sequence size produced was less than 0.2%. The modified QuickMerge algorithm presents results that are orders of magnitude better than the other two algorithms. Unfortunately this fact does not make modified QuickMerge the best algorithm, as it is expensive to reach this balancing. The fault-tolerant version of the original QuickMerge algorithm presented the worst results, followed by HyperQuickSort.

Figure 9 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 437 s, 465 s, 550 s and 821 s, 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 Lan and Mohamed (1992), 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.

6 Conclusions

In this work we introduced a novel ABFT technique applied to hypercube-based parallel sorting algorithms. 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 versions of the HyperQuickSort, QuickMerge and Bitonic Sort parallel algorithms were 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 algorithms 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. Building an application programming interface (API) can make the task of programming a fault-tolerant version of an arbitrary algorithm easier. 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 Research Council (CNPq) Grant 308959/2020-5, and the São Paulo Research Foundation (FAPESP), CINEMA Project, Process 21/06923-0.

References

Akl, S.G. (1985) *Parallel Sorting Algorithms*, 1st ed., Academic Press, USA.

Alghamdi, T. and Alaghband, G. (2019) 'High performance parallel sort for shared and distributed memory MIMD', in *International Conference on Applied Computing 2019*, November, pp.113–122.

Al-Hashimi, M.A., Abulnaja, O.A., Saleh, M.E. and Ikram, M.J. (2017) 'Evaluating power and energy efficiency of bitonic mergesort on graphics processing unit', *IEEE Access*, Vol. 5, pp.16429–16440.

Bagherpour, N., Hammarling, S., Higham, N., Dongarra, J. and Zounon, M. (2017) *D6.6 – Algorithm-based Fault Tolerance Techniques*, Technical Report, NLAFET Consortium – H2020–FETHPC–2014: GA 671633.

Banerjee, P., Rahmeh, J.T., Stunkel, C., Nair, V.S., Roy, K., Balasubramanian, V. and Abraham, J.A. (1990) 'Algorithm-based fault tolerance on a hypercube multiprocessor', *IEEE Transactions on Computers*, Vol. 39, No. 9, pp.1132–1145.

Bao, C. and Zhang, S. (2020) 'Algorithm-based fault tolerance for discrete wavelet transform implemented on GPUs', *Journal of Systems Architecture*, Vol. 108, p.101823.

Batcher, K.E. (1968) 'Sorting networks and their applications', in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68*, Spring, ACM, New York, NY, USA, pp.307–314.

Bland, W., Du, P., Bouteiller, A., Herault, T., Bosilca, G. and Dongarra, J. (2012) 'A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI', in *The 18th International Conference on Parallel Processing, Euro-Par '12*, Springer-Verlag, Berlin, Heidelberg, pp.477–488.

Bland, W., Bouteiller, A., Hérault, T., Bosilca, G. and Dongarra, J. (2013) 'Post-failure recovery of MPI communication capability: design and rationale', *IJHPCA*, Vol. 27, No. 3, pp.244–254.

Bougeret, M., Casanova, H., Robert, Y., Vivien, F. and Zaidouni, D. (2014) 'Using group replication for resilience on exascale systems', *The International Journal of High Performance Computing Applications*, Vol. 28, No. 2, pp.210–224.

Camargo, E. and Duarte Jr., E. (2018) 'Running resilient MPI applications on a dynamic group of recommended processes', *Journal of the Brazilian Computer Society*, December, Vol. 24.

Chen, Z. and Dongarra, J. (2006) 'Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources', in *IPDPS*, 10pp.

Chen, Z. and Jack, D. (2008) 'Algorithm-based fault tolerance for fail-stop failures', *IEEE Trans. Parallel Distrib. Syst.*, pp.1628–1641.

Chen, J., Li, S. and Chen, Z. (2016) 'GPU-ABFT: optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs', in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp.1–2.

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

Davies, T., Karlsson, C., Liu, H., Ding, C. and Chen, Z. (2011) 'High performance linpack benchmark: a fault tolerant implementation without checkpointing', in *ICS*, pp.162–171.

Du, P., Bouteiller, A., Bosilca, G., Herault, T. and Dongarra, J. (2012) 'Algorithm-based fault tolerance for dense matrix factorizations', in *PPoPP*, pp.225–234.

Duarte Jr., E.P. and Nanya, T. (1998) 'A hierarchical adaptive distributed system-level diagnosis algorithm', *IEEE Transactions on Computers*, January, Vol. 47, No. 1, pp.34–45.

Duarte, E.P., Bona, L.C.E. and Ruoso, V.K. (2014) 'Vcube: a provably scalable distributed diagnosis algorithm', in *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, November, pp.17–22.

Durad, M.H., Akhtar, M.N. and Irfan-ul-Haq (2014) 'Performance analysis of parallel sorting algorithms using MPI', in *2014 12th International Conference on Frontiers of Information Technology*, pp.202–207.

Egwutuoha, I.P., Levy, D., Selic, B. and Chen, S. (2013) '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.

Elnozahy, E.N.M., Alvisi, L., Wang, Y-M. and Johnson, D.B. (2002) 'A survey of rollback-recovery protocols in message-passing systems', *ACM Comput. Surv.*, September, Vol. 34, No. 3, pp.375–408.

Fagg, G.E. and Dongarra, J. (2000) '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*, Springer-Verlag, London, UK, pp.346–353.

Ferreira, K.B., Stearley, J., Laros III, J.H., Oldfield, R., Pedretti, K.T., Brightwell, R., Riesen, R., Bridges, P.G. and Arnold, D. (2011) 'Evaluating the viability of process replication reliability for exascale systems', in *SC*, p.44.

Filiposka, S., Mishev, A. and Gilly, K. (2019) 'Multidimensional hierarchical VM migration management for HPC cloud environments', *J. Supercomput.*, Vol. 75, No. 8, pp.5324–5346.

Foster, I., Foster, I.T. and Foster, J. (1995) *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, Literature and Philosophy*, Addison-Wesley.

Gupta, S., Patel, T., Engelmann, C. and Tiwari, D. (2017) '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, SC '17*, Association for Computing Machinery, New York, NY, USA, pp.1–12.

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

Huang, K-H. and Abraham, J.A. (1984) 'Algorithm-based fault tolerance for matrix operations', *IEEE Transactions on Computers (TOC)*, June, Vol. C-33, No. 7, pp.518–528.

Hursey, J. and Graham, R.L. (2011) 'Building a fault tolerant MPI application: a ring communication example', in *IPDPS Workshops*, pp.1549–1556.

Hussain, Z., Znati, T. and Melhem, R. (2020) 'Enhancing reliability-aware speedup modelling via replication', in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp.528–539.

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

Kosaian, J. and Rashmi, K.V. (2021) 'Arithmetic-intensity-guided fault tolerance for neural network inference on GPUs', in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.1–15.

Kumar, V. (2002) *Introduction to Parallel Computing*, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Lan, Y. and Mohamed, M.A. (1992) 'Parallel quicksort in hypercubes', in *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990s, SAC '92*, ACM, New York, NY, USA, pp.740–746.

Lee, D. and Batcher, K.E. (1994) 'On sorting multiple bitonic sequences', in *1994 International Conference on Parallel Processing*, August, Vol. 1, pp.121–125.

Leighton, F.T. (2014) *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*, Elsevier Science.

Losada, N., González, P., Martín, M.J., Bosilca, G., Bouteiller, A. and Teranishi, K. (2020) 'Fault tolerance of MPI applications in exascale systems: the ULFM solution', *Future Generation Computer Systems*, Vol. 106, pp.467–481.

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

MPI Forum (2015) *Document for A Standard Message-Passing Interface 3.1*, Technical Report, University of Tennessee.

MPI Forum (2020) *User-Level Failure Mitigation* [online] https://bitbucket.org/icldistcomp/ulfm2/src/ulfm/ (accessed 26 September 2020).

Netti, A., Kiziltan, Z., Babaoglu, O., Sîrbu, A., Bartolini, A. and Borghesi, A. (2020) 'A machine learning approach to online fault classification in HPC systems', *Future Generation Computer Systems*, Vol. 110, pp.1009–1022.

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

Quinn, M.J. (1989) 'Analysis and benchmarking of two parallel sorting algorithms: HyperQuickSort and QuickMerge', *BIT Numerical Mathematics*, June, Vol. 29, No. 2, pp.239–250.

Quinn, M.J. (2003) *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill.

Raju, K., Chiplunkar, N.N. and Rajanikanth, K. (2019) 'A CPU-GPU cooperative sorting approach', in *2019 Innovations in Power and Advanced Computing Technologies (i-PACT)*, Vol. 1, pp.1–5.

Reed, D.A. and Dongarra, J. (2015) 'Exascale computing and big data', *Commun. ACM*, June, Vol. 58, No. 7, pp.56–68.

Rocco, R., Gadioli, D. and Palermo, G. (2022) 'Legio: fault resiliency for embarrassingly parallel MPI applications', *J. Supercomput.*, February, Vol. 78, No. 2, pp.2175–2195.

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

Schlichting, R.D. and Schneider, F.B. (1983) 'Fail-stop processors: an approach to designing fault-tolerant computing systems', *ACM Trans. Comput. Syst.*, Vol. 1, No. 3, pp.222–238.

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

Tiwari, D., Gupta, S. and Vazhkudai, S.S. (2014) 'Lazy checkpointing: exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems', in *DSN*, pp.25–36.

Wagar, B. (1987) 'HyperQuickSort: a fast sorting algorithm for hypercubes', in *Proc. of the 2nd Conf. on Hypercube Multiprocessors*, pp.292–299.

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

White, S., Verosky, N. and Newhall, T. (2012) 'A CUDA-MPI hybrid bitonic sorting algorithm for GPU clusters', in *2012 41st International Conference on Parallel Processing Workshops*, pp.588–589.

Zhao, K., Di, S., Li, S., Liang, X., Zhai, Y., Chen, J., Ouyang, K., Cappello, F. and Chen, Z. (2020) 'FT-CNN: algorithm-based fault tolerance for convolutional neural networks', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 7, pp.1677–1689.

Zhu, Y., Liu, Y. and Zhang, G. (2020) 'FT-PBLAS: PBLAS-based fault-tolerant linear algebra computation on high-performance computing systems', *IEEE Access*, Vol. 8, pp.42674–42688.