International Journal of Networking and Computing – `www.ijnc.org`, ISSN 2185-2847
Volume 12, Number 1, pages 47–72, January 2022

Task-Level Resilience: Checkpointing vs. Supervision

Jonas Posner, Lukas Reitz, and Claudia Fohry
Research Group Programming Languages / Methodologies
University of Kassel, Germany
Email: {jonas.posner | lukas.reitz | fohry}@uni-kassel.de

Received: July 16, 2021
Revised: September 30, 2021
Accepted: November 29, 2021
Communicated by Masahiro Shibata

Abstract

With the advent of exascale computing, issues such as application irregularity and permanent hardware failure are growing in importance. Irregularity is often addressed by task-based parallel programming implemented with work stealing. At the task level, resilience can be provided by two principal approaches, namely checkpointing and supervision. For both, particular algorithms have been worked out recently. They perform local recovery and continue the program execution on a reduced set of resources. The checkpointing algorithms regularly save task descriptors explicitly, while the supervision algorithms exploit their natural duplication during work stealing and may be coupled with steal tracking to minimize the number of task re-executions. Thus far, the two groups of algorithms have been targeted at different task models: checkpointing algorithms at dynamic independent tasks, and supervision algorithms at nested fork-join programs.

This paper transfers the most advanced supervision algorithm to the dynamic independent tasks model, thus enabling a comparison between checkpointing and supervision. Our comparison includes experiments, running time predictions, and simulations of job set executions. Results consistently show typical resilience overheads below 1% for both approaches. The overheads are lower for supervision in practically relevant cases, but checkpointing takes over for order millions of processes. [1]

Keywords: Fault Tolerance, Resilience, Work Stealing, Asynchronous Many-Task Programming, Runtime Systems

1 Introduction

As supercomputing applications deploy an increasing number of cluster nodes, their likelihood of experiencing hardware failure such as permanent node loss grows [2, 3, 4]. Resilience is typically provided by checkpoint/restart, which, in its traditional form, transparently saves the whole program state on disc and after failure restarts the program from the latest checkpoint [5]. Other approaches operate at the application level; they include application-level checkpointing [6] and algorithm-based fault tolerance (ABFT [7]). Application-level approaches are harder to use than

[1]This paper is an extended version of Jonas Posner, Lukas Reitz, and Claudia Fohry: Checkpointing vs. Supervision Resilience Approaches for Dynamic Independent Tasks. IEEE Proceedings International Parallel and Distributed Processing Symposium (IPDPS) Workshops (APDCM), 2021 [1].

traditional checkpoint/restart, but they cause less overhead. Moreover, they allow to continue the program execution after failure.

This paper considers resilience techniques at the intermediate level of an Asynchronous Many-Task (AMT) runtime system. Intermediate-level approaches may combine the above benefits, but received less attention so far. AMT runtimes are a particularly interesting target for them, since the clearly defined interfaces of tasks support task re-execution after failure.

AMT programs may be coded in environments such as OpenMP [8], HPX [9], Chapel [10], Cilk [11], GLB [12], Legion [13], and many others, and are getting increasingly popular. The environments differ widely in their task models, i.e., in their mechanisms for task generation and cooperation. Several environments require all tasks to be known from the beginning (called *static tasks*). We consider environments for *dynamic tasks*, which allow the tasks to generate child tasks during their execution. Important subclasses are dynamic independent tasks (e.g., GLB), nested fork-join programs (e.g., Cilk), dataflow-based task models (e.g., Legion, HPX), and models that exchange data through a (physically or logically) shared memory via side effects (e.g., OpenMP, Chapel).

In all AMT environments, programmers must specify tasks and their dependencies. Then a runtime system assigns the tasks to physical resources, called workers. In our setting, workers correspond to processes that run on multiple cluster nodes.

Dynamic tasks are usually implemented with *work stealing.* Therein each worker stores its initial tasks and their descendants in a local pool. When the pool runs empty, the worker (called *thief*) attempts to steal tasks from a co-worker (called *victim*). More specifically, the local pool holds *task descriptors* that include task inputs and, if needed, a reference to the task code.

Enabling task re-execution after failure requires to duplicate the task descriptors beforehand. For dynamic tasks, this can be accomplished in one of two principal ways: Either one saves the task descriptors explicitly for the purpose of fault tolerance, or one relies on their natural duplication during work stealing. The former approach corresponds to task-level checkpointing, whereas the latter enables one worker (the supervisor) to take over tasks from their usual owner if necessary. Outside our scope, static tasks permit simpler resilience schemes, see Section 7.

Following the checkpointing and supervision approaches, a few specific algorithms have been developed recently. They refer to different task models: All previous checkpointing algorithms refer to dynamic independent tasks (DIT), and all previous supervision algorithms refer to nested fork-join programs (NFJ). This leads to two research questions: 1) How do the two approaches compare to each other in terms of running time overhead? and 2) Can the approaches be generalized to other task models such as those based on dataflow and side effects?

This paper gives a comprehensive answer to the first question, leaving most of the second one for future research. We start by porting the most advanced supervision algorithm from NFJ to DIT, thereby both showing that this is possible, and creating a basis for a subsequent comparison. Our results immediately apply to the DIT model, which is used, for example, in tree search and optimization algorithms [14, 15, 16]. Moreover, they may provide guidance to future research on the second question.

In the following, we further explain the above concepts, and then outline the specific contributions of this paper. Several topics will be expanded in Section 2.

DIT programs start with one or several initial tasks. Each task may spawn any number of children, giving rise to a computation tree. Tasks are not allowed to communicate, with the exception of parameter passing from parents to children. There is no result return, but the final result is calculated from task results by reduction with an associative and commutative operator (e.g., integer summation). This reduction is the only synchronization point in a program. To speed up the calculation, each worker accumulates its own local results in a *worker result.*

NFJ programs always start with a single task. As in the previous model, task spawning gives rise to a computation tree, and parents may pass parameters to children. However, parents always wait for a result being reported back from their children. By integrating task results upwards in the tree, the final result is eventually calculated in the root. Parameter passing and result return are the only means of inter-task communication.

Most NFJ runtimes such as that of Cilk adopt a *work-first* policy. Therein, after spawning

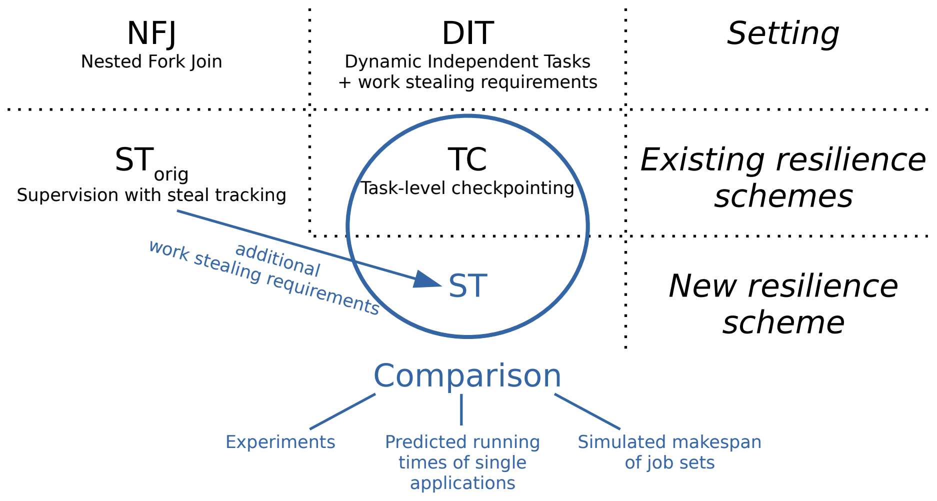

NFJ
Nested Fork Join
DIT
Dynamic Independent Tasks
+ work stealing requirements
Setting
ST
orig
Supervision with steal tracking
TC
Task-level checkpointing
Existing resilience schemes
additional
work stealing requirements
ST
New resilience scheme
Comparison
Experiments
Predicted running times of single applications
Simulated makespan of job sets

Figure 1: Overview of acronyms and contributions

a task, a worker places the continuation of the parent task into the local pool and branches into the child [17]. In contrast, DIT systems such as the Global Load Balancing library GLB adopt a *help-first* policy, in which the worker places the child task into the local pool and continues with the parent. Related to that, Cilk work stealing takes a single task, whereas GLB work stealing takes multiple tasks, called a *task bag*.

The task-level *checkpointing algorithms for DIT* regularly save the local pool contents of each worker, alongside its current worker result, in a resilient in-memory store [18, 19]. They adopt uncoordinated checkpointing, i.e., the workers write checkpoints independently. The algorithms involve sophisticated protocols to maintain consistency during stealing. A typical representative is the AllFT scheme from Posner *et al.* [19].

In the *supervision algorithms for NFJ*, each victim keeps the descriptors of stolen tasks until result return, and initiates task re-execution if the thief fails [20, 21, 22, 23]. For efficiency, re-execution of a whole subtree should be avoided, though. The algorithm by Kestor *et al.* [23] is able to identify all descendants that have been stolen away to healthy nodes. It achieves this by *steal tracking*, in which it piggybacks local history information onto normal communication. Upon failure, the information is collected at the supervisor of the failed task, which identifies the orphaned descendants and manages their incorporation into the task's re-computation.

From now on, TC denotes the task-level checkpointing algorithm from [19], and ST_{orig} denotes the combination of supervision and steal tracking from [23]. For an overview of acronyms see Figure 1. Both TC and ST_{orig} share the ability to recover locally from multiple failures such that the program execution need not be interrupted. A comparison of the algorithms has thus far not been possible, as TC refers to DIT, and ST_{orig} refers to NFJ.

Therefore, this paper first transfers ST_{orig} to DIT, and then performs the comparison. Experiments refer to the Global Load Balancing library GLB [12, 24]. As illustrated in Figure 1, this paper makes the following contributions:

1. We transfer ST_{orig} to the DIT setting, where we name it ST for brevity. Novel features include a transparent fork-join style synchronization between victims and thieves, result accumulation at the granularity of task bags, and the definition of history information that is appropriate for help-first scheduling.

2. We experimentally compare TC and ST by running five benchmarks on up to 640 workers. We observe that the resilience overheads in failure-free runs are smaller for ST, but restore is faster for TC.

3. We derive formulas for the overall running times of the two schemes, including failure handling.

The formulas depend on Mean Time Between Failures (MTBF), number of workers, and steal rate.

4. Based on the formulas, we predict running times in larger-scale settings than in our experiments. First, we predict the execution times of single long-running applications under failures. Second, we perform simulations to determine the makespans of job sets, in which either all jobs are made resilient via TC or ST (*protected* jobs), or none of the jobs uses any resilience scheme (*unprotected* jobs). Our results strongly suggest that program protection by TC or ST is effective, and that the difference between the two is rather low. We find that ST performs slightly better in all currently realistic scenarios, but TC takes over in systems with the order of millions of processes.

The remainder of this paper is organized as follows. Section 2 states assumptions and provides background. Thereafter, Section 3 describes the redesign of ST for DIT (contribution 1). Experiments are explained and discussed in Section 4 (contribution 2). Then, Section 5 derives the running time formulas (contribution 3), and Section 6 presents predictions and simulations (contribution 4). The paper finishes with related work and conclusions in Sections 7 and 8, respectively.

2 Assumptions and Background

Before explaining TC and $\mathrm{ST}_{\mathrm{orig}}$, we define the failure types that they can handle, and the dynamic independent tasks setting.

2.1 Failure Model

Our schemes handle permanent (also called fail-stop) failures of workers, and assume reliable network communication. Different workers that run on the same node are allowed to fail independently, although in practice they will usually go lost together. Any number of workers may fail at any time, including unsuitably correlated times such as during restore. However, we do not permit failure of the resilient store (for TC), and failure of the root worker (for ST). These cases lead to program abort if no further precautions are taken. Failure never compromises the correctness of a computed result.

We presume that all workers are notified of failures, possibly with a delay. Recovery is performed locally and does not interrupt task processing at unaffected workers. After a failure, the program continues on the smaller number of intact workers.

2.2 Dynamic Independent Tasks Setting

As described in Section 1, **dynamic independent tasks** cooperate through parameter passing and the contribution of task results to a final result. Tasks must not have side effects, and are supposed to behave deterministically. Listing 1 shows an example code calculating the number of valid placements of N queens on an $N \times N$ chessboard (NQueens). It is invoked by calling `nqueens(new PosList(), 0)`. Upon termination, the result may be queried from the system. Each `nqueens` call is a task.

In the following, we state **requirements on work stealing**. They originate from the design of TC [19], but are also adopted for ST. Further requirements for ST will be added in Section 3. From now on, *DIT* stands for the dynamic independent tasks model described above, in combination with the following work stealing requirements:

- The help-first policy is used.
- Workers must be equal, disregarding more advanced features such as multi-threaded workers or hierarchical work stealing [25].

```
void nqueens(PosList queens, int depth) {
  if (depth == n) {
    incrementResult();
  } else {
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        if (isValidPosition(queens, i, j)) {
          spawn nqueens(add(queens, i, j), depth+1);
        }
      }
    }
  }
}
```

Listing 1: Dynamic independent tasks: NQueens

- Work stealing must be *cooperative*; i.e., the thief sends a steal request to the victim and the victim responds actively by sending tasks (called *loot*) or a reject message.
- Workers alternate between task processing and communication *phases*. Only in the latter they may answer steal requests and accept loot. At the beginning of each communication phase, a worker must have finished all tasks that it has previously taken from the pool, including result accumulation and the insertion of child tasks into the pool.
- Steals must not leave the local pool empty, and a maximum of one steal between the same thief and victim may be in progress at a time.

2.3 TC for DIT Setting

In this section, we describe the core concepts of TC for DIT. Further information can be found in references [19] and [24].

In TC, workers write backups independently, mainly

- at regular time intervals (called *regular backups*),
- during stealing (called *steal backups*),
- after task adoption during restore, and
- at the beginning and end of their computation.

Each backup contains the current contents of the local pool and the current worker result. Recall that backups are written in communication phases, and thus they capture the worker's entire state except for open communication. Backups are saved in a resilient in-memory store, which must support concurrent accesses and transactions. A steal protocol guarantees consistency among victim, thief, and their respective backups. It involves three messages (compared to two in non-resilient stealing), two backups, and a temporary loot saving.

When workers are notified of failures, their actions depend on the relationship with the failed worker. The majority of workers simply perform some bookkeeping operations, whereas most of the recovery is accomplished by a single designated worker called buddy. The buddy adopts the failed worker's tasks and takes care of any loot sent from it. If the buddy fails, TC regulates its succession in a resilient manner.

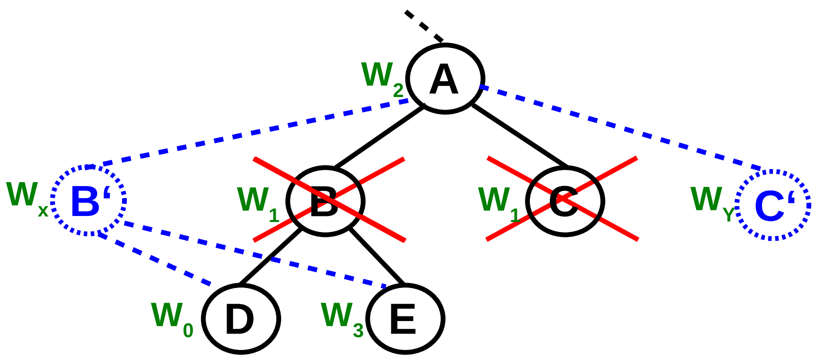


Figure 2: Recovery in $\mathrm{ST_{orig}}$

2.4 $\mathbf{ST_{orig}}$ for NFJ

Listing 2 shows an example code for NFJ, which computes Fibonacci numbers and is invoked by calling `fib(n)`. The parent task waits for the results of all children with an explicit `sync`, otherwise there would be an implicit `sync` at the end of the function.

```
int fib(int n) {
  if (n < 2) return n;
  int x = spawn fib(n-1);
  int y = spawn fib(n-2);
  sync;
  return x + y;
}
```

Listing 2: Nested Fork-Join: Fibonacci

$\mathrm{ST_{orig}}$ refers to a cluster implementation of NFJ [23]. The initial task (here `fib(n)`) is processed by worker 0. At each spawn, a worker branches into the child and places the continuation of the parent task into its local pool (work-first). Continuations technically have the form of stack *frames*.

Each steal takes the oldest frame from the local pool. Thus, the thief processes the parent frame or an ancestor, and the victim processes the child. When a child is finished, the victim keeps the result. When a thief encounters a `sync`, it returns the parent frame to the victim, where it is matched with the child result using a frame ID. Depending on timing, the frame is either sent back to the thief or kept at the victim. The other worker steals a new frame. Matching may have to be applied transitively at a chain of victims.

At each steal, the victim keeps a copy of the stolen frame. If a failure occurs, it initiates re-computation using this copy. This enables recovery from any number of failures, except failure of worker 0. However, a naive re-spawn of the children would cause potentially expensive re-computations of their entire subtrees.

Therefore, the major achievement of $\mathrm{ST_{orig}}$ is the incorporation of *all* intact subcomputations beneath a faulty one. Figure 2 illustrates this concept, with thieves (continuations) drawn below victims. In the example, worker W_1 has stolen tasks `B` and `C` from worker W_2. (It took `C` when `B` was finished, but `D` and `E` had not yet returned.) Similarly, `D` and `E` were stolen by W_0 and W_3, respectively. When W_1 fails, the recovery is led by node `A`, that is, by worker W_2. This worker initiates the re-computations of `B` and `C` (called `B'` and `C'`, respectively), and incorporates the intact subcomputations `D` and `E`, as marked by blue dotted lines.

The feasibility of the approach relies on the following concepts:

- A *steal tree* [26] is a graph with nodes representing frames and edges representing steals, as in the solid line parts of Figure 2. Each node is labeled with a *frame ID*, the *history* of this frame (see below), and the *rank* of the processing worker. Frame ID and history are computed at the victim, and then piggybacked onto the loot delivery message from victim to thief, and stored at the thief. Frame IDs are quadruples:

$$frame\ ID = (stage,\ level,\ step,\ victim\ rank),$$

where *stage* denotes the number of frames that the victim itself had stolen before it was stolen from, and level and step identify the particular frame taken from the victim during this stage. For example, the call `fib(n)` gives rise to two children at the next *level*, and three *steps* for the three continuations encountered (the three subcomputations corresponding to a complete `fib` function, and the remainders after each spawn, respectively). Note that each ID uniquely identifies a frame.

The *history* of a frame encompasses the IDs of the frame itself, all predecessors in the steal tree, and all pending older siblings of frame/predecessors (e.g., `B` for `C`).

- Upon failure, each worker checks whether it is a victim of the failed worker and has not yet received the result (pending steal). If so, it issues a system-wide call to collect all histories that include the lost frame (e.g., W_2 collects the histories of `D`, `E`). It compresses these histories into a *replay tree*, which supports rapid access to orphaned grandchildren.
- At any following steal, the replay tree is given away to an *alias* worker. Prior to processing the tree, this worker communicates its rank to the orphaned grandchildren. The tree processing itself differs from normal operation. In particular, 1) the stealing of previously unstolen subframes is suppressed, 2) the stealing of lost subframes is enforced (a replay tree is constructed for them beforehand), and 3) the subframes available in orphans are discarded, and the orphan frames are patched instead. Details can be found in [23].

$\mathrm{ST_{orig}}$ can handle any number of non-root failures. Resiliency during recovery is achieved via bookkeeping of aliases. Moreover, a ForwardUnify protocol reduces data losses in return chains. Details and a discussion of correctness can be found in [23].

3 Redesign of ST for DIT Setting

Summarizing the previous definitions, our transformation of $\mathrm{ST_{orig}}$ into ST must handle the following differences between DIT and NFJ:

(i) All DIT tasks synchronize with a single ancestor (one-level async-finish structure), whereas each NFJ task synchronizes with its immediate parent.

(ii) DIT results are calculated independently from the spawn tree, by accumulating and combining worker results, whereas NFJ tasks are calculated upwards in the tree.

(iii) Multiple initial DIT tasks may be assigned to one or several workers, whereas NFJ always deploys a single initial task.

(iv) DIT stealing obtains child tasks (help-first), whereas NFJ stealing obtains parent frames (work-first). The DIT tasks are processed from beginning to end, whereas the NFJ tasks are split into continuations.

(v) DIT stealing refers to task bags as opposed to single tasks, and, unlike in NFJ, these bags need not be taken from the pool bottom.

To handle the differences, the design of ST imposes two additional work stealing requirements in addition to the DIT guarantees:

1) Staged operation: Each worker must repeatedly steal a task bag, process all tasks from this bag (possibly with the help of thieves), send back the result, steal the next task bag, and so on.

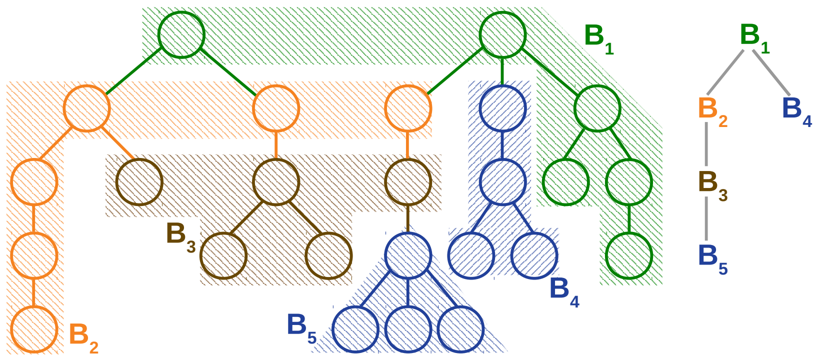


Figure 3: Task bags and steal tree in ST

Thus, it processes exactly one task bag in each stage (but may store others whose results are still open). Note that stages and phases are different concepts.

2) Determinism: Repeated executions of the same operations on the same local pool must always yield the same pool contents. Determinism concerns the selection of tasks to be stolen, the extraction of tasks to be processed, and the insertion order of spawned tasks.

To handle difference **(i)**, ST imposes an artificial fork-join structure, requesting that thieves report back to their victims when they have finished a bag. Unlike in NFJ, this structure is not visible at the program level. As illustrated in Figure 3, the new synchronization granularity is finer than normally in DIT, but coarser than in NFJ.

In Figure 3, two initial tasks are processed by four workers marked by different colors. Work stealing gives rise to task bags $B_1 \ldots B_5$. For instance, B_2 is stolen by the orange worker from the green one. The orange worker reports back when B_2, including B_3 and B_5, is finished.

The steal tree is defined analogously to $\mathrm{ST}_{\mathrm{orig}}$, except that nodes represent task bags. For our example, it is shown in Figure 3 (right). Analogously to $\mathrm{ST}_{\mathrm{orig}}$, nodes are labeled by bag ID, history, and worker rank.

To handle difference **(ii)**, results are accumulated per task bag, and are included when the thief reports back to the victim. This is actually simpler than in $\mathrm{ST}_{\mathrm{orig}}$, as nothing more needs to be done with a finished bag (unlike for $\mathrm{ST}_{\mathrm{orig}}$'s frames). Thus, after a result return, the thief always proceeds to steal, whereas $\mathrm{ST}_{\mathrm{orig}}$ distinguishes two cases. Similarly, ST does not require transitive matching.

The above structure causes the async-finish synchronization to be superfluous, and we therefore omit it.

To handle difference **(iii)**, an artificial root node is inserted into the steal tree if the initial tasks are assigned to different workers (not shown in the figure). This node is labeled with a bag of all initial tasks and assigned to worker 0, whereas its children hold the initial task bags of the different workers. Like a victim, the root node acts as a supervisor and waits for its children's results.

To handle differences **(iv)** and **(v)**, we need bag IDs instead of frame IDs, with the following new definition:

$$bag\ ID = (stage,\ step,\ substep,\ loot\ size,\ victim\ rank).$$

In the definition:

- *stage* is the same as in $\mathrm{ST}_{\mathrm{orig}}$ (using the staged operation requirement),
- *step* is the number of tasks that the victim has processed in this stage before extracting the bag,
- *substep* is the number of tasks that the victim has given away at this step before extracting the current bag, and
- *loot size* is the number of tasks in the bag.

Analogously to ST_{orig}, an ID uniquely describes a bag. The history is defined as in ST_{orig}, except that we reduce the data volume by omitting the stages and substeps of siblings.

Recovery is performed analogously to ST_{orig}. For example, where ST_{orig} discards a frame, ST removes in the corresponding step as many tasks from the pool as indicated by the loot size. It is the same tasks as in the original execution, according to the determinism requirement. Like ST_{orig}, the scheme can handle any number of non-root failures, following the case-by-case analysis in [23].

Our implementation of ST is sketched in Section 4.1. It has about the same code size/complexity as TC.

4 Experimental Evaluation

This section compares and analyzes the running times of TC and ST, first in failure-free runs, and then under failures. We start by outlining our implementations and describing the experimental setting.

4.1 Implementation

An important DIT implementation is the GLB library [12] of the "APGAS for Java" programming framework [27]. This framework follows the well-known Partitioned Global Address Space (PGAS) programming model [28]. As TC was previously implemented by extending GLB [19, 24], we did the same with ST. The previous codes [29], as well as the codes for this paper [30] are publicly available.

GLB realizes a work stealing variant called lifeline-based global load balancing [31]. Therein, thieves find work by contacting several random workers, followed by a few lifeline buddies. The latter record unsuccessful steal requests and possibly answer them later. Thus, a worker may receive loot from lifeline buddies while it is still processing another task bag. To ensure staged operation, our ST implementation rejects such loot with a certain protocol [32]. Moreover, we omit the loot sizes from bag IDs, because GLB presumes a predetermined size such as steal-half. As explained by Bungart and Fohry [33], the loss of a large number of workers may dissect the lifeline graph. In accordance with TC, we did not implement graph repair for simplicity.

GLB runs one or several workers per node. These workers communicate asynchronously by sending active messages. GLB determines that the local pool data structure is provided by users. Our benchmarks ensure determinism: Each worker repeatedly takes a block of n tasks from its pool, processes these tasks, and inserts any spawned children immediately. Loot is extracted from the opposite end of the pool.

While GLB itself is not resilient, TC and ST use the resilience mode of APGAS, which automatically invokes failure handlers at each worker [19] and deploys the `IMap` data structure of Hazelcast [34] as an in-memory resilient store. As the resilience mode incurs a certain overhead, it was switched off for the GLB runs.

4.2 Experimental Setting

The experiments were run on two clusters:

- **Kassel** [35] has a partition with 12 homogeneous Infiniband-connected nodes, each with two 6-core Intel Xeon E5-2643-v4 CPUs and 256 GB of main memory. We started up to 144 workers and used a close mapping: For example, we started up to 12 workers on one node, 24 workers on two nodes, etc.

- **Goethe-HLR** [36] has a partition with homogeneous Infiniband-connected nodes, each with two 20-core Intel Xeon Skylake Gold 6148 CPUs and 192 GB of main memory. We used up to 16 nodes for a total of 640 workers, also with a close mapping.

As benchmarks, we deployed frequently used ones with both static and dynamic tasks: Unbalanced Tree Search (UTS) [37], Betweenness Centrality (BC) [38], NQueens [39], and two synthetic ones introduced below (DynamicSyn and StaticSyn). UTS dynamically generates a highly irregular tree and counts the number of tree nodes, BC calculates a centrality score for each node of a graph, and NQueens was already sketched in Section 2.

The synthetic benchmarks perform some placeholder computation and support smooth weak scaling. For this, the user provides a desired running time T^{BASE}. Then a GLB run takes time $T^{NO} = T^{BASE} + \epsilon$, with ϵ reflecting the costs of work stealing. Moreover, the user can influence the number m of tasks per worker, as well as specify a fluctuation range v for the task durations.

StaticSyn and BC deploy static tasks, which are evenly distributed across workers at the beginning. Task durations in StaticSyn are varied per worker. For example, for an average task duration of 10 ms and $v = 20\%$, one worker may obtain m tasks with a duration of 8 ms each, and another m tasks with a duration of 12 ms each (random times within the fluctuation range).

DynamicSyn, UTS, and NQueens deploy dynamic tasks, starting the computation with a single task on worker 0. DynamicSyn generates a perfect w-ary task tree, where w and m are automatically chosen/adjusted.

In all benchmarks except BC, task results are single `long` values and the reduction operator is scalar sum. In BC, task results are `long` arrays and the reduction operator is component-wise sum.

We used existing non-resilient and TC implementations of UTS, BC, and NQueens [29]. In the TC implementations, we kept the regular backup interval at 10 seconds, as suggested in reference [19]. We focused on small task sizes, to obtain clearer results. The benchmark parameters and the GLB parameter n were set as follows:

- UTS: geometric tree shape, branching factor 4, tree depth 18 (Kassel) or tree depth 19 (Goethe-HLR), random seed 19, $n = 511$;
- BC: number of graph nodes 2^{18}, random seed 2, $n = 511$;
- NQueens: $N = 17$, threshold 11 (Kassel), or $N = 18$, threshold 12 (Goethe-HLR), $n = 511$;
- StaticSyn: $T^{BASE} = 100$ s, $m = 6000$, $v = 20\%$, $n = 1$;
- DynamicSyn: $T^{BASE} = 100$ s, $m = 1,000,000$, $v = 20\%$, $n = 511$.

Table 1 displays the average task execution times for the above benchmark configurations on Goethe-HLR.

StaticSyn	**DynamicSyn**	**UTS**	**BC**	**NQueens**
17 ms	100 µs	360 ns	120 ms	115 ns

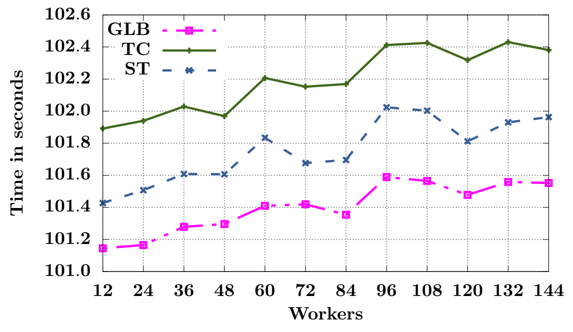
Table 1: Average task execution times on Goethe-HLR

4.3 Failure-Free Runs

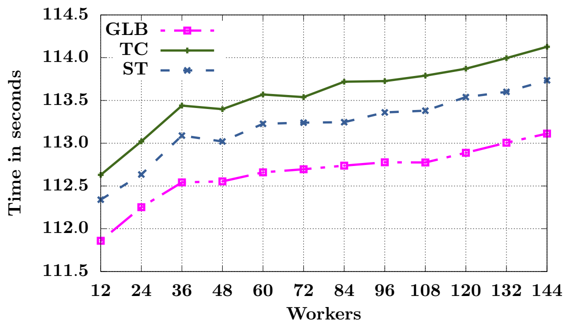
4.3.1 Running Times of Synthetic Benchmarks

Figure 4 (top) depicts the running times of StaticSyn (left) and DynamicSyn (right), reporting averages over 5 runs. Note that the scales do not start from zero.

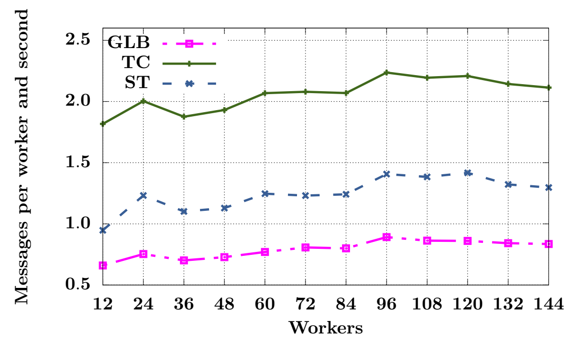
It can be observed that all overheads of TC/ST over non-resilient GLB are below 1%, which is comparable to the overheads of work stealing (the latter corresponds to the difference between the GLB running times and T^{BASE}). The ST overheads are consistently about half of the TC overheads. For StaticSyn, they are a maximum of 0.43 s (ST) vs. 0.86 s (TC). For DynamicSyn, they are a maximum of 0.65 s (ST) vs. 1.10 s (TC). The curves run roughly in parallel. As the GLB overheads result from work stealing, this suggests that the resilience costs increase proportionally to the steal rate.



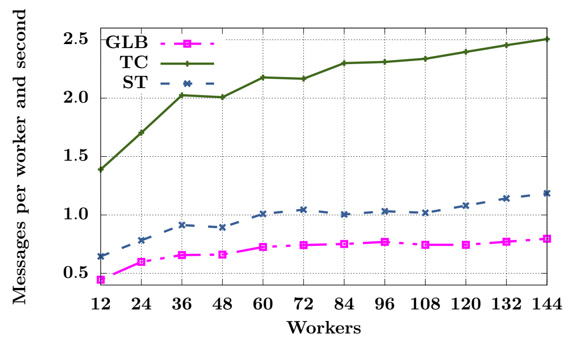
(a) StaticSyn: running time



(b) DynamicSyn: running time



(c) StaticSyn: Messages per worker and second



(d) DynamicSyn: Messages per worker and second

Figure 4: Weak scaling on Kassel: StaticSyn (left), DynamicSyn (right), running times (top), and messages per worker and second (bottom)

4.3.2 Number of Messages

Figure 4 (bottom) presents the number of messages sent for the same program runs as above. Again, the ST curve is clearly located beneath the TC one, indicating that part of the performance difference is owing to differences in the communication overheads. The ST curve is closer to the GLB one, however, suggesting that another part of the difference is owing to ST's computation costs for history maintenance.

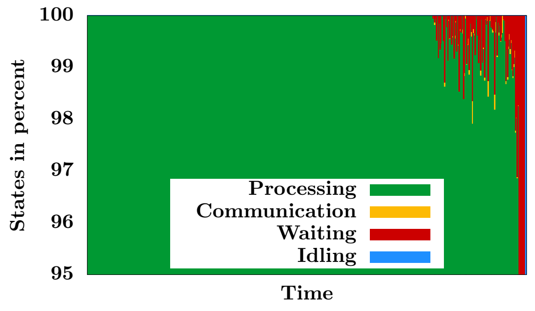
The concrete numbers in Figure 4 meet our expectations: while GLB issues two messages per steal, ST issues three, and TC issues seven (see Sections 3 and 2.3, respectively). This results in factors 1.5 and 3.5 for the respective message numbers.

4.3.3 Histograms

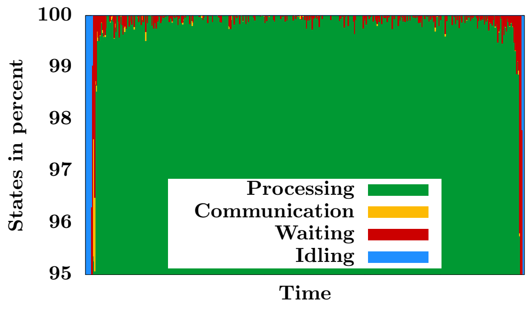
Figure 5 depicts histograms of the processor time usage of the workers for DynamicSyn and StaticSyn. For each particular time, the histograms represent the share of workers in the following states:

- *processing*: worker processes tasks (green),
- *communication*: worker is involved in stealing or backup writing (orange),
- *waiting*: worker is waiting for a response to a steal request (red), and
- *idling*: worker is initially or finally inactive (blue).

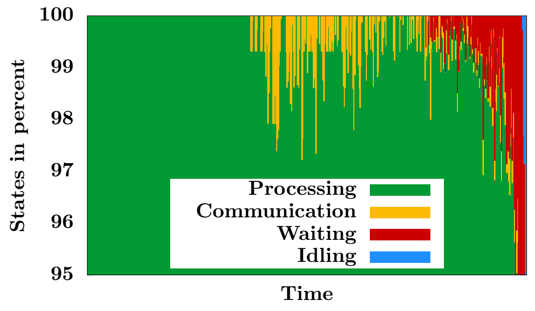
Note that work stealing results in communication and waiting states. The histograms refer to a run with $T^{BASE} = 20$ s and 144 workers on Kassel. A small T^{BASE} value was used to pronounce



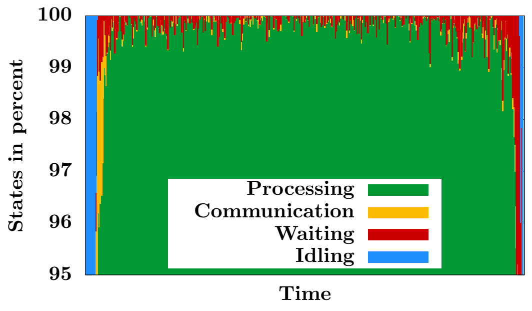
(a) StaticSyn: GLB



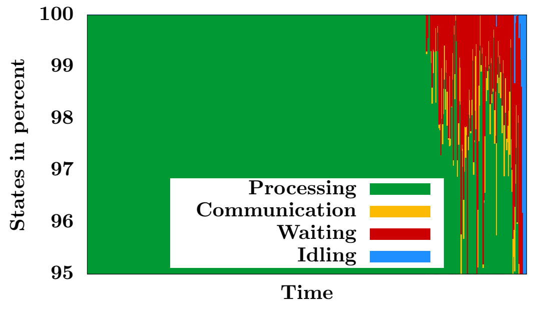
(b) DynamicSyn: GLB



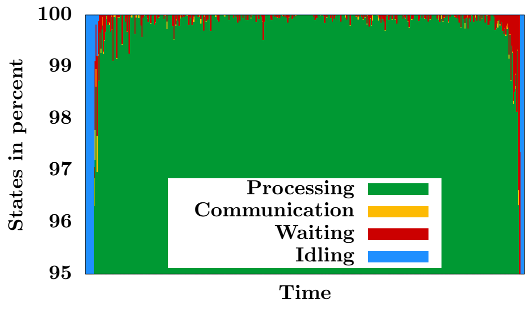
(c) StaticSyn: TC



(d) DynamicSyn: TC



(e) StaticSyn: ST



(f) DynamicSyn: ST

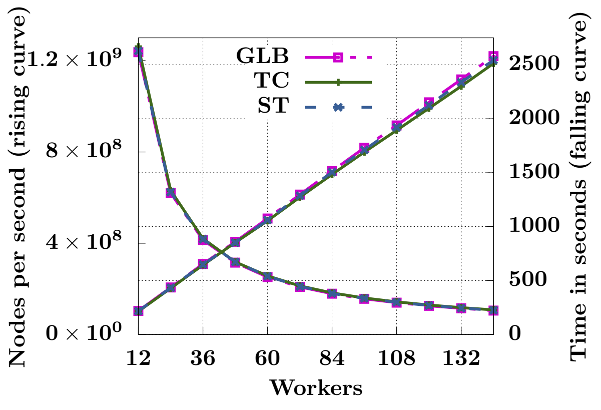
Figure 5: Histograms of processor time usage with 144 workers on Kassel. *Left side:* StaticSyn, $T^{BASE} = 20$ s. *Right side:* DynamicSyn, $T^{BASE} = 20$ s.

the start and end phases in the figures. Similarly, the histograms were cut at 95% to save space. The omitted parts are almost exclusively green.

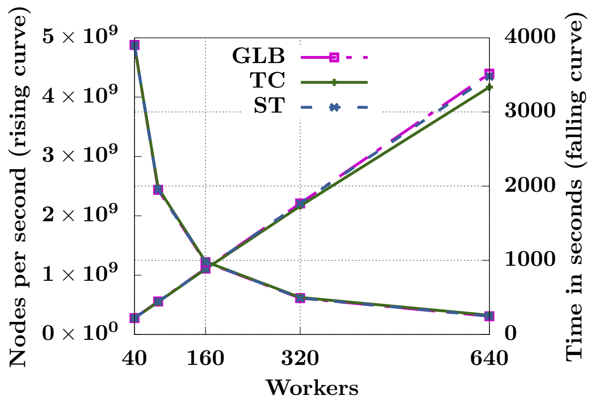
Interpreting the histograms, StaticSyn (Figure 5, left side) starts all workers in processing state. As the task distribution is even, they remain in this state for most of the time. Work stealing arises only in the end phase, owing to the variations in task durations. The number of steals increases until more and more workers enter idle state.

Although all StaticSyn histograms share this same pattern, the work stealing states take more room in TC and ST. The reason can be seen in their higher numbers of messages, as discussed before. The TC histogram exhibits several communication spikes in the middle, which are owing to regular backup writing. The reason that this is not one spike is because we have offset the backups slightly to reduce competition in accessing the resilient store.

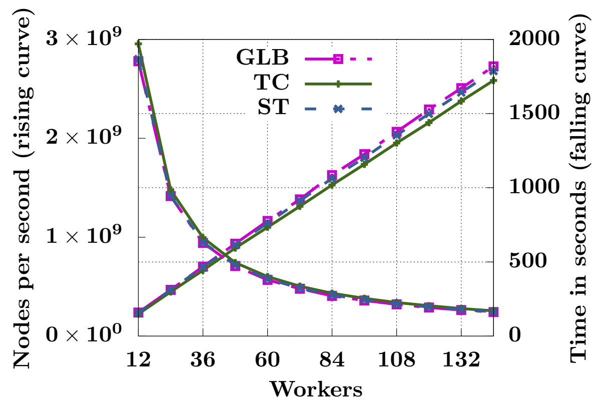
DynamicSyn (Figure 5, right side) exhibits a start phase in which the initial task is transformed into initial task bags for all workers. A noticeable number of steals occurs in the main phase, as the benchmark is irregular. Again, the work stealing takes more room in TC and ST. In accordance to StaticSyn, the difference is more distinct for TC.



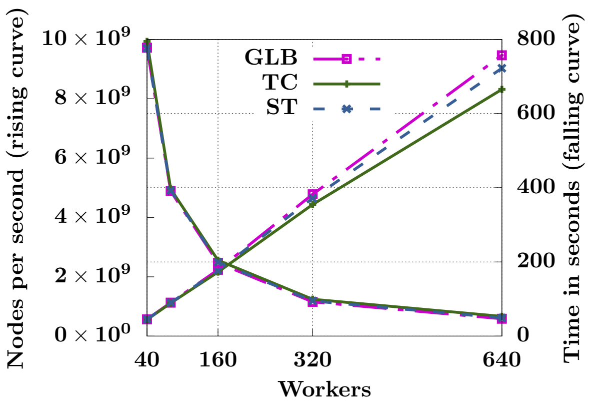
(a) Kassel: UTS



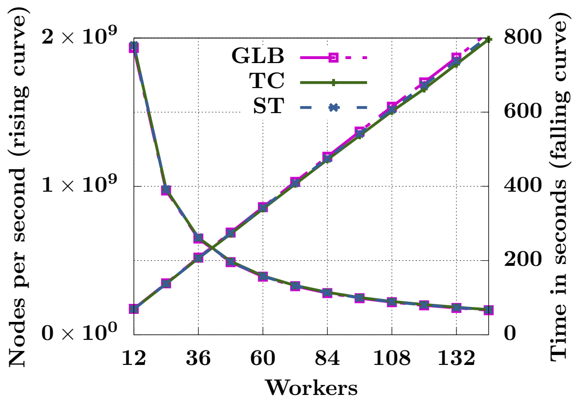
(b) Goethe: UTS



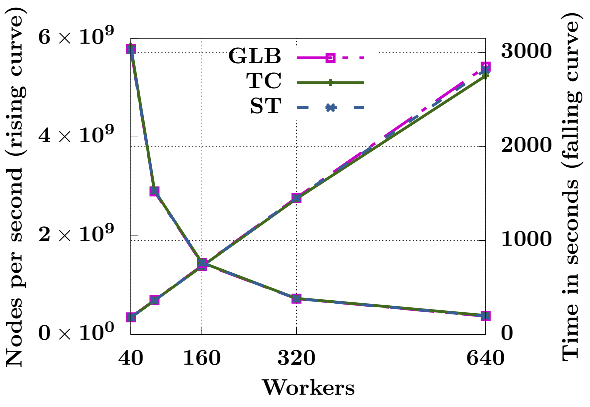
(c) Kassel: BC



(d) Goethe: BC



(e) Kassel: NQueens



(f) Goethe: NQueens

Figure 6: Strong scaling performance of GLB, TC, and ST on Kassel (left side), and Goethe-HLR (right side).

4.3.4 UTS, BC, and NQueens

Figure 6 depicts the running times on Kassel and Goethe-HLR. Unlike before, this figure employs strong scaling to convey an impression of the magnitudes. The figure presents two curves: a falling one describing the running times, and a rising one describing the number of processed nodes (of the respective benchmarks) per second. All TC and ST curves are close to the GLB ones, indicating again that the resilience overheads are low.

4.4 Recovery Time

The recovery time consists of the time to perform the actual recovery, the time for reprocessing the lost tasks, and the surplus time owing to the loss of computing power in the subsequent computation. Table 2 summarizes these times for a concrete, but typical example of a DynamicSyn run. The table reports averages of 100 runs, into which we injected 1 or 2 failures, such that random workers failed at random times. As indicated in the table, the major differences between TC and ST are in the time required for reprocessing lost tasks, which is much higher for ST than for TC, and in the overhead of failure-free runs, which is higher for TC.

	1 failure		**2 failures**	
	TC	**ST**	**TC**	**ST**
Failure-free overhead	2.96 s	1.14 s	2.96 s	1.14 s
Actual recovery	0.39 s	0.35 s	0.67 s	0.70 s
Reprocessing	0.05 s	1.36 s	0.08 s	2.84 s
Lost computation	1.51 s	1.49 s	3.11 s	3.28 s
Total running time (incl. above costs)	119.95 s	119.38 s	121.81 s	123.84 s

Table 2: Recovery times of DynamicSyn run on Goethe-HLR with 40 workers, injecting 1 or 2 failures

5 Estimation of Running Times

5.1 Assumptions and Notation

In this section, we estimate the running times of TC and ST for the case that $x \ll p$ failures occur at independent and identically distributed times. We derive formulas that will later be used to determine conditions under which either TC or ST is superior. We use the following notation:

- p: number of workers,
- r: steal rate (average number of steals per worker and second),
- j: regular backup rate (number of regular backups per worker and second),
- $T^{NO}(p)$: running time of non-resilient work-stealing algorithm on a given program call (a program call is an invocation with fixed inputs),
- $T_x^{alg}(p)$: expected running time of $alg = \{ST \mid TC\}$ when x failures are encountered during this program call, and
- $MTBF$: Mean Time Between Failures (system-wide).

5.2 Running Time of TC

Backed by $x \ll p$, we assume that the x failures affect different workers. At each of them, the respective failure strikes with equal probability at any particular time during the program's execution. This is for the reason that hardware components live much longer than what the program run takes, and thus their susceptibility to failure is about constant. From general properties of uniform distributions, the expected time for the occurence of a worker's failure is at half of its running time (and thus the overall computing power p available for our computation is reduced to $p-(1/2)$). Similarly, summing up the x uniformly distributed times, which are independent from the above assumption, implies that an expected $x/2$ of the overall computing power is lost (the expected value of a sum equals the sum of the expected values). The corresponding share of work must be taken over by the other workers, leading to a proportional increase in running time. Similarly, the other workers must repeat an expected half of the work from the last backup interval of each failed worker. As the average interval length is $b = 1/(r+j)$, we obtain

$$T_x^{TC}(p) = \frac{p}{p-(x/2)} T_0^{TC}(p) + \sum_{i=1}^{x} \frac{b}{2(p-i)} + xR^{TC} \,,$$

where R^{TC} is the cost of the actual recovery procedure. R^{TC} is essentially independent of p, as can be easily observed from the algorithm description in Section 2.3. Furthermore, $T_0^{TC}(p)$ differs from $T^{NO}(p)$ by the checkpointing overhead. Most of this overhead increases proportionally to the steal and regular backup rates, and thus

$$T_0^{TC}(p) = (1 + c_0 r + c_1 j)\, T^{NO}(p)$$

for some constants c_0 and c_1. With $c_2 = R^{TC}$, we obtain

$$T_x^{TC}(p) = \frac{p}{p-(x/2)} (1 + c_0 r + c_1 j)\, T^{NO}(p) + \sum_{i=1}^{x} \frac{b}{2(p-i)} + xc_2 \,. \tag{1}$$

5.3 Running Time of ST

Upon each failure, the failed worker's share of previous work is lost, which on average is $1/p$-th of the overall previous work [23]. Similarly, the worker's $1/p$-th share of future work must be taken over by the other workers, resulting in a proportional increase in running time. We obtain

$$T_x^{ST}(p) = \frac{p}{p-x} T_0^{ST}(p) + \sum_{i=1}^{x} R^{ST}(p-i) \,,$$

where $R^{ST}(p)$ denotes the overhead of the recovery procedure with p workers and is analyzed below. $T_0^{ST}(p)$ differs from $T^{NO}(p)$ by the costs to maintain and communicate the history information, as well as by the costs to report back the results of task bags. Similar to ST, most of these costs increase proportionally to the steal rate, and thus

$$T_0^{ST}(p) = (1 + c_3 r)\, T^{NO}(p) \,.$$

$R^{ST}(p)$ covers the overheads of the following actions:

a) scan all locally stored frames and their histories, searching for frames to/from the failed worker (at each worker),

b) participate in the system-wide history collection (at each worker per lost frame), and

c) compress the collected histories into a replay tree (at one worker per lost frame).

To estimate the costs of actions a) to c), we make some observations about the typical size of the steal tree parameters: number of vertices $n = \Theta(p)$ (as each worker processes one frame in steady state), node degree $d = \Theta(1)$ (as random steals spread evenly across the tree), tree height $h = \Theta(\log\ p)$, and history length $l = \Theta(dh) = \Theta(\log\ p)$. On this basis, step a) requires time $\Theta(ld) = \Theta(\log\ p)$; step b) collects a maximum of $\mathrm{O}(pld) = \mathrm{O}(p\ \log\ p)$ data, resulting in time $\mathrm{O}(\log\ p)$ per worker; and step c) processes the collected data for an average time of $\mathrm{O}(\log\ p)$ per worker. Therefore, we estimate $R^{ST}(p-i) \approx R^{ST}(p) = c_4 \log(p)$, and obtain

$$T_x^{ST}(p) = \frac{p}{p-x} (1 + c_3 r)\, T^{NO}(p) + xc_4 \log p. \tag{2}$$

5.4 Estimation of Constants

We experimentally determined approximations for the constants c_0 to c_4, based on the single-failure cases of formulas (1) and (2). They are displayed in Table 3. Except for c_4, the values were directly measured, averaging over 25 runs of the DynamicSyn benchmark on 200 workers. For c_4, we first calculated $R^{ST}(p)$ for several p, as the difference between $T_0^{ST}(p)$ and $T_1^{ST}(p)$ in 100 runs of the DynamicSyn benchmark on $20, 40, \ldots, 240$ workers. Thereafter, we approximated c_4 through a regression analysis using the least squares method. The fitted $R^{ST}(p)$ function has an R^2 value of 0.944898.

Constants	c_0	c_1	c_2	c_3	c_4
Value	38 ms	41 ms	558 ms	24 ms	219 ms

Table 3: Experimentally determined values of the constants

5.5 Experimental Validation

By inserting the above constants from single-failure runs into formulas (1) and (2), we obtain predictions for multi-failure cases. To confirm these, we injected up to 12 failures into DynamicSyn runs with $T^{BASE} = 100$ s and 40 workers on Goethe-HLR. Again, we let random workers fail at random times. Figure 7 depicts our results, reporting averages of 100 runs, alongside the running time predictions. It can be observed that predictions and measurements are very close.

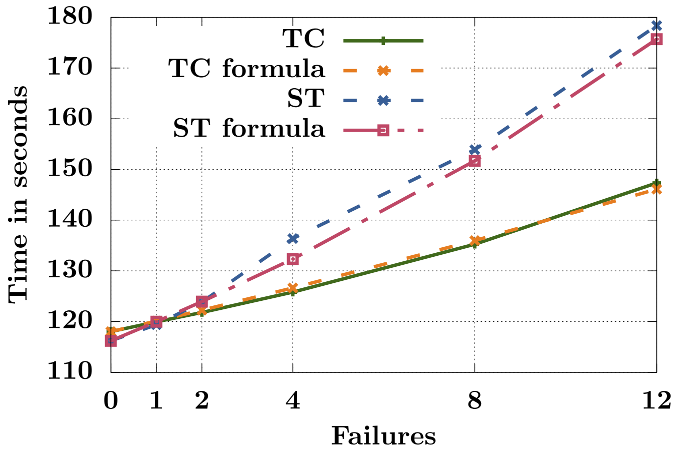


Figure 7: Measurements and predictions for failures out of 40 workers

6 Prognosis

Based on the formulas from Section 5, this section considers two types of failure-prone settings: 1) single long-running applications, and 2) sets of short applications. For both, we determine conditions under which either TC or ST are superior.

6.1 Long-Running Applications

Using our notation from Section 5, we consider a program call that likely experiences exactly one failure; that is, a program call with running time

$$MTBF = T^{NO}(p) \approx T_0^{TC}(p) \approx T_0^{ST}(p)\,.$$

Below, we derive conditions under which, say, TC outperforms ST for this call. The same conditions also apply to program calls with longer running times. To see this, consider the program executions as being composed of sections of length MTBF. If TC is superior in a single MTBF section, then TC is superior in every MTBF section, provided that the conditions are stable. The conditions involve $MTBF$, r, j, and p. The first three are normally stable (or quite stable) inside an application, and p only declines slightly for a reasonably low number of failures. We determine the conditions by solving the inequality

$$T_1^{TC}(p) < T_1^{ST}(p) \ ,$$

substituting equations (1) and (2) from Section 5 and setting $T^{NO}(p) = MTBF$. We obtain two solutions:

- $p > k_1/k_0$ and

  $$MTBF < (p - 0.5)\frac{(p-1)(c_4 \log(p) - c_2) + (b/2)}{k_0 p^2 - k_1 p} \text{ , and}$$

- $p < k_1/k_0$ and

  $$MTBF > (p - 0.5)\frac{(p-1)(c_4 \log(p) - c_2) + (b/2)}{k_0 p^2 - k_1 p} \text{ ,}$$

where $k_0 = 1 + c_0 r + c_1 j + c_3 r - 1$, and $k_1 = 1 + c_0 r + c_1 j - 0.5 c_3 r - 0.5$.

Under these conditions, TC outperforms ST; otherwise, ST is superior. By inserting our estimates for c_0 to c_4 from Table 3, we obtain the exemplary values in Table 4, which are break-even points between TC and ST superiority. The table only refers to the first solution; the second one translates into $p < 28$. Note that the table display the worker MTBF for clarity, which is calculated as $p \cdot MTBF$ [3].

We conclude that ST is usually superior, but TC takes over for the order of millions of workers.

Workers (p)	r	j	Worker MTBF
1,000	0.28	0.033	< 13.9 hours
100,000	0.28	0.033	< 122.9 days
1,000,000	0.28	0.033	< 4.2 years
1,000	0.05	0.033	< 7.1 days
100,000	0.05	0.033	< 3.0 years
1,000,000	0.05	0.033	< 17.7 years

Table 4: Scenarios in which TC outperforms ST

6.2 Sets of Jobs

When scheduling job sets on a supercomputer, one often strives for a low overall completion time, known as *makespan*. We considered sets of independent parallel jobs that were known a priori, and simulated their execution in faulty environments. In each simulation run, we protected all jobs in the respective set in the same manner. We considered three options:

- All jobs were unprotected, i.e., the jobs did not use any resilience scheme, but instead aborted and were re-spawn in the event of a failure.
- All jobs were protected by TC.
- All jobs were protected by ST.

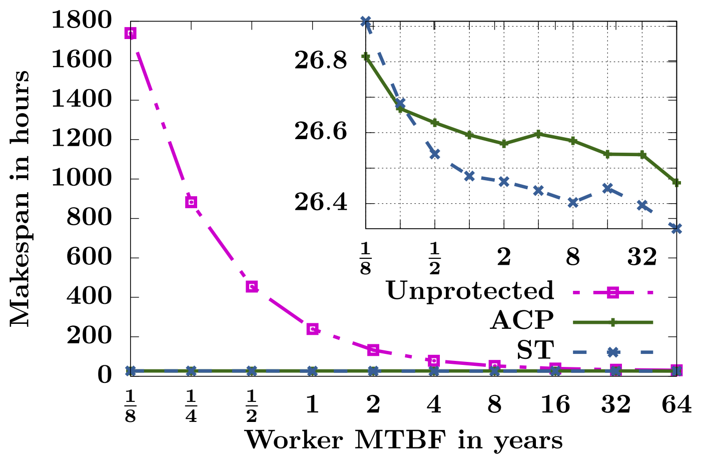


Figure 8: Mira simulation

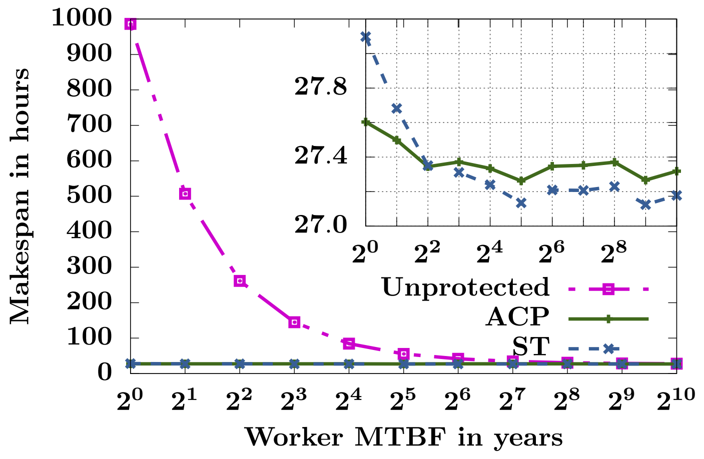


Figure 9: Exa simulation

For our experiments, we adapted the job scheduling simulator from [40, 41], which was originally designed for studying silent errors. The simulator starts the jobs in priority order, we used random priorities. We randomly injected fail-stop failures into our simulation runs. For the TC/ST jobs, we simulated the running time increases with the formulas from Section 5. For unprotected jobs, we aborted the job and re-queued it. Note that we continued TC/ST jobs on less than p workers, but restarted unprotected jobs on p workers. Two job sets on different supercomputers were considered:

- **Mira:** This computer (ranked 22 by top500.org in November 2019) has a total of $49,152$ nodes [42]. We extracted 30 job sets, namely one per day during June 2019, from the official published log data [2]. Each job set holds $66 \ldots 277$ jobs with running times of $37 \ldots 86$ s, and $p = 512 \ldots 49,152$.

- **Exa:** This hypothetical exascale computer has 1 million nodes. We generated 30 job sets, each with approximately 1250 jobs, with running times of $50 \ldots 10,000$ s (for a total of $p \cdot 24$ h) and $p = 500 \ldots 500,000$.

Figures 8 and 9 depict the simulated makespans, averaged over all respective job sets and 1,000 runs of each. Both figures indicate a clear difference between protected and unprotected jobs. For example, in the Mira experiment with a worker MTBF of 0.125 years, TC reduces the

[2]This data was generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357 [43].

makespan *by* as much as 98.46%, and ST reduces the makespan by 98.45%. The effect is smaller for a long MTBF, but at a worker MTBF of 64 years, the reduction is still 12.01% for TC and 12.44% for ST. Similarly, in the Exa experiment with a worker MTBF of 1 year, TC reduces the makespan by 97.20%, and ST reduces the makespan by 97.15%.

The difference between TC and ST is rather small. As detailed in the upper right corners, ST is slightly better for *worker MTBF* > 0.25 years in Mira and for > 4 years in Exa.

Overall, in both simulations ST caused slightly less overhead than TC, except for a low worker MTBF. This outcome agrees with that of Section 6.1. Our most striking result was a huge difference between protected and unprotected jobs, strongly indicating that job protection by an intermediate-level resilience method pays off.

7 Related Work

Although substantial research on fault tolerance has been conducted in recent years [2, 3, 44, 45], resilient programs are not yet state of the art. In a recent survey among participants of the US Exascale Computing Project, only 2% of the respondents reported on current fault-tolerant applications, whereas 67% were planning for such applications [46].

Regarding fail-stop failures, checkpoint/restart is the prevailing approach [47, 48, 5]. This method is available in a traditional variant, which writes data to a shared file system, and newer variants such as uncoordinated, in-memory, and multi-level checkpointing. The traditional variant is realized by system-level libraries such as BLCR [49] and DMTCP [50], and the newer ones by application-level libraries such as FTI [51] and SCR [52]. Application-level libraries provide users with control over aspects such as data selection [53]. All variants restart the application after failures, delays may be avoided by allocating spare nodes at job submission [54].

Other application-level approaches include naturally fault-tolerant algorithms [55] and algorithm-based fault tolerance (ABFT) [7, 56]. Checkpoint/restart and ABFT can be combined, such that different program sections are protected differently [57]. Chung *et al.* defined transaction-based containment domains, which encapsulate failures and recoveries in hierarchically arranged program segments [58]. Some program-level approaches rely on resilient arrays, e.g., the resilience scheme of NWChem [59]. These approaches require failure notification, as offered by programming systems such as ULFM for MPI [60] and Resilient X10 [61]. Other resilience support of programming systems includes in-memory checkpointing [62] and replication [63].

Previous intermediate-level resilience techniques were mostly aimed at static tasks, e.g., in MapReduce [64], hierarchical master/worker patterns [65], and the A* algorithm [66]. A well-known example is the lineage technique of Spark [67].

Resilience for dynamic tasks has received rather little attention to date, and the research conducted is spread across different topics. To note some examples, Kurt *et al.* [68] consider soft errors that are discovered late during the execution of a task graph, and minimize the number of task re-executions. Cao *et al.* [69] detect silent data corruptions in task graphs and reduce task re-executions with the help of checkpointing. Subasi *et al.* [70] handle silent errors with a combination of task-/system-level checkpointing and message logging. Ma and Krishnamoorthy [71] consider fail-stop failures for tasks with side effects, and suggest a technique to avoid updating the same data twice.

Prior to ST_{orig}, the Cilk-NOW system used supervision for dynamic tasks, but did not integrate healthy subtasks [20]. Later, Satin improved on it by integrating finished subtasks and aborting the others [21]. Moreover, checkpointing was added to Satin [22]. In contrast to that work, ST_{orig} integrates *all* subtasks that are available on healthy nodes [23].

Among the precursors of TC (e.g., [72, 24]) is an X10 scheme that explicitly saves data in other workers' main memories instead of using a resilient store [18]. This scheme was also combined with dynamic resource management [33]. Moreover, variants of TC, such as incremental checkpointing [19], have been studied. Recently, a TC-like algorithm for NFJ has been sketched [73].

Besides GLB, TC and ST should be applicable to other DIT systems. Examples include the YewPar parallel tree search framework [15, 16] and the Blaze-Tasks framework [74].

Theoretical analyses are common in fault tolerance research. For example, these have been used to determine optimal checkpoint intervals [75, 3], and to evaluate combinations of checkpoint/restart with process replication [76]. Similar to our work, Subasi *et al.* [70] derive formulas for the overall running time of their combined scheme, and use these for a comparison with checkpoint/restart. Makespan analyses of job sets have to date focused on resilient scheduling heuristics for parallel jobs [40], checkpointing strategies for optimized system I/O [77], and the impact of malleability on job scheduling [78].

8 Conclusions

In this paper, we have compared two resilience approaches for AMT runtime systems. Although these approaches (TC and ST) were previously known, they had thus far been targeted at different task models, namely DIT and NFJ programs, respectively.

We first transferred ST to the DIT setting, and then compared TC with ST. Our redesign includes a novel fork-join structure between victims and thieves, and the definition of appropriate history information for help-first scheduling. The comparison involved experiments, predictions, and simulations. We conducted experiments with up to 640 processes and five benchmarks, using the GLB library. Thereafter, we determined conditions under which TC/ST are superior in single application runs. For this purpose, we derived running time formulas depending on MTBF, number of workers, and steal rate. Finally, we simulated the execution of job sets on a real and a hypothetical supercomputer and evaluated the makespans.

The three investigations consistently support the same conclusions: program protection at the intermediate level of an AMT runtime system pays off. Moreover, the choice between TC and ST is secondary. We consistently observed ST as being superior in typical current settings, but TC takes over on large machines and for frequent errors.

Future work should address limitations of the current study. Most importantly, it should consider advanced task models such as those based on data flow and data exchange through side effects, as well as advanced features of AMT runtime systems such as a hierarchical structuring of workers. The advanced settings differ from DIT in at least two fundamental aspects: 1) the consideration of locality in task placement and victim selection, and 2) the possibility of tasks causing side effects. The first aspect is orthogonal to the design of TC and ST, and so we expect both approaches to be transferable. The second aspect will require additional methods, as task descriptors no longer cover the entire functionality of a task, and update operations must not be re-executed.

Future research may also extend our experiments to larger benchmarks, other DIT systems, and more workers. Finally, TC/ST should be integrated with the handling of other failure types such as silent errors.

References

[1] Jonas Posner, Lukas Reitz, and Claudia Fohry. Checkpointing vs. supervision resilience approaches for dynamic independent tasks. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS) Workshops (APDCM)*. IEEE, 2021. doi:10.1109/IPDPSW52791.2021.00089.

[2] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *The Int. Journal of High Performance Computing Applications (IJHPCA)*, 28(2):129–173, 2014. doi:10.1177/1094342014522573.

[3] Thomas Herault and Yves Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing.* Springer, 2015. doi:10.1007/978-3-319-20943-2.

[4] Al Geist. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum*, 10:2–3, 2016. URL: `https://spectrum.ieee.org/computing/hardware/how-to-kill-a-supercomputer-dirty-power-cosmic-rays-and-bad-solder`.

[5] Faisal Shahzad, Markus Wittmann, Moritz Kreutzer, Thomas Zeise, Georg Hager, and Gerhard Wellein. A survey of checkpoint/restart techniques on distributed memory systems. *Parallel Processing Letters (PPL)*, 23(4):1340011–1340030, 2013. `doi:10.1142/s0129626413400112`.

[6] Marcos Maroñas, Sergi Mateo, Kai Keller, Leonardo Bautista-Gomez, Eduard Ayguadé, and Vicenç Beltran. Extending the OpenCHK model with advanced checkpoint features. *Future Generation Computer Systems (FGCS)*, 112:738–750, 2020. `doi:10.1016/j.future.2020.06.003`.

[7] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing (JPDC)*, 69(4):410–416, 2009. `doi:10.1016/j.jpdc.2008.12.002`.

[8] OpenMP Architecture Review Board. OpenMP API 5.1 Specification, 2020. URL: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf`.

[9] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A task based programming model in a global address space. In *Proc. Int. Conf. on Partitioned Global Address Space Programming Models (PGAS)*, pages 1–11. ACM, 2014. `doi:10.1145/2676870.2676883`.

[10] Bardford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *The Int. Journal of High Performance Computing Applications (IJHPCA)*, 21(3):91–312, 2007. `doi:10.1177/1094342007078442`.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998. `doi:10.1145/277650.277725`.

[12] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat, and Mikio Takeuchi. GLB: Lifeline-based global load balancing library in X10. In *Proc. Workshop on Parallel Programming for Analytics Applications (PPAA)*, pages 31–40. ACM, 2014. `doi:10.1145/2567634.2567639`.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. ACM, 2012. `doi:10.1109/SC.2012.71`.

[14] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[15] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. Implementing YewPar: A framework for parallel tree search. In *Proc. Euro-Par Parallel Processing*, pages 184–196. Springer, 2019. `doi:10.1007/978-3-030-29400-7_14`.

[16] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. YewPar: Skeletons for exact combinatorial search. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 292–307, 2020. `doi:10.1145/3332466.3374537`.

[17] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–12. IEEE, 2009. `doi:10.1109/ipdps.2009.5161079`.

[18] Claudia Fohry, Marco Bungart, and Paul Plock. Fault tolerance for lifeline-based global load balancing. *Journal of Software Engineering and Applications (JSEA)*, 10(13):925–958, 2017. `doi:10.4236/jsea.2017.1013053`.

[19] Jonas Posner, Lukas Reitz, and Claudia Fohry. A comparison of application-level fault tolerance schemes for task pools. *Future Generation Computer Systems (FGCS)*, 105:119–134, 2020. doi:10.1016/j.future.2019.11.031.

[20] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proc. Annual Conf. on USENIX*, pages 1–10, 1997. doi:10.5555/1268680.1268690.

[21] G. Wrzesińska, R.V. van Nieuwpoort, J. Maassen, and H.E. Bal. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, page 13.1. IEEE, 2005. doi:10.1109/ipdps.2005.224.

[22] G. Wrzesińska, A.M. Oprescu, T. Kielmann, and H. Bal. Persistent fault-tolerance for divide-and-conquer applications on the grid. In *Proc. Euro-Par Parallel Processing*, volume 4641, pages 425–436, 2007. doi:10.1007/978-3-540-74466-5_46.

[23] Gokcen Kestor, Sriram Krishnamoorthy, and Wenjing Ma. Localized fault recovery for nested fork-join programs. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 397–408. IEEE, 2017. doi:10.1109/ipdps.2017.75.

[24] Jonas Posner and Claudia Fohry. A Java task pool framework providing fault-tolerant global load balancing. *Int. Journal of Networking and Computing (IJNC)*, 8(1):2–31, 2018. doi:10.15803/ijnc.8.1_2.

[25] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Proc. Int. Conf. on Partitioned Global Address Space Programming Models (PGAS)*. ACM, 2011.

[26] Jonathan Lifflander, Sriram Krishnamoorthy, and V. Laxmikant Kale. Steal tree: low-overhead tracing of work stealing schedulers. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 507–518. ACM, 2013. doi:10.1145/2491956.2462193.

[27] Olivier Tardieu. The APGAS library: resilient parallel and distributed programming in Java 8. In *Proc. SIGPLAN Workshop on X10*, pages 25–26. ACM, 2015. doi:10.1145/2771774.2771780.

[28] Vijay Saraswat, George Almasi, Ganesh Bikshandi, et al. The asynchronous partitioned global address space model. In *Proc. SIGPLAN Workshop on Advances in Message Passing*. ACM, 2010.

[29] Jonas Posner. PLM-APGAS-Examples, 2020. URL: https://github.com/posnerj/PLM-APGAS-Applications.

[30] Jonas Posner, Lukas Reitz, and Claudia Fohry. Artefact: Checkpointing vs. Supervision Resilience Approaches for Dynamic Tasks, 2020. URL: http://doi.org/10.5281/zenodo.3941784.

[31] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *Proc. SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 201–212. ACM, 2011. doi:10.1145/1941553.1941582.

[32] Lukas Reitz. Design and evaluation of a work stealing-based fault tolerance scheme for task pools. Mastersthesis, University of Kassel, 2019.

[33] Marco Bungart and Claudia Fohry. A malleable and fault-tolerant task pool framework for X10. In *Proc. Int. Conf. on Cluster Computing, Workshop on Fault Tolerant Systems*, pages 749–757. IEEE, 2017. doi:10.1109/CLUSTER.2017.27.

[34] Hazelcast. The leading open source in-memory data grid, 2020. URL: http://hazelcast.org.

[35] Competence Center for High Performance Computing in Hessen (HKHLR). Linux cluster kassel, 2021. URL: https://www.hkhlr.de/en/clusters/linux-cluster-kassel.

[36] TOP500.org. Goethe-hlr, 2018. URL: https://www.top500.org/system/179588.

[37] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 235–250. Springer, 2006. doi:10.1007/978-3-540-72521-3_18.

[38] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35, 1977. doi:10.2307/3033543.

[39] Evgeni J. Gik. *Schach und Mathematik*. Thun, 1 edition, 1987.

[40] Anne Benoit, Valentin Le Fèvre, Padma Raghavan, Yves Robert, and Hongyang Sun. Design and comparison of resilient scheduling heuristics for parallel jobs. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–12. IEEE, 2020. doi:10.1109/IPDPSW50202.2020.00099.

[41] Valentin Le Fèvre. Source code of job simulator, 2020. URL: http://www.github.com/vlefevre/job-scheduling.

[42] TOP500.org. Mira - BlueGene/Q, Power BQC 16C 1.60GHz, 2020. URL: https://www.top500.org/system/177718.

[43] Argonne Leadership Computing Facility. Mira log traces, 2020. URL: https://reports.alcf.anl.gov/data/mira.html.

[44] Saurabh Hukerikar and Christian Engelmann. Resilience design patterns: A structured approach to resilience at extreme scale. *Supercomputing Frontiers and Innovations (JSFI)*, 4(3):4–42, 2017. doi:10.14529/jsfi170301.

[45] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations (JSFI)*, 1(1):5–28, 2014. doi:10.14529/jsfi140101.

[46] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. A survey of MPI usage in the US Exascale Computing Project. *Concurrency and Computation: Practice and Experience (CCPE)*, 32(3), 2020. doi:10.1002/cpe.4851.

[47] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013. doi:10.1007/s11227-013-0884-0.

[48] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *Computing Surveys (CSUR)*, 34(3):375–408, 2002. doi:10.1145/568522.568525.

[49] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conf. Series*, 46:494–499, 2006. doi:10.1088/1742-6596/46/1/067.

[50] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–12. IEEE, 2009. doi:10.1109/ipdps.2009.5161063.

[51] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–32. ACM, 2011. doi:10.1145/2063384.2063427.

[52] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. ACM, 2010. doi:10.1109/SC.2010.18.

[53] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. Experimental evaluation of application-level checkpointing for OpenMP programs. In *Proc. Int. Conf. on Supercomputing (ICS)*, pages 2–13. ACM, 2006. doi:10.1145/1183401.1183405.

[54] Atsushi Hori, Kazumi Yoshinaga, Thomas Herault, Aurélien Bouteiller, George Bosilca, and Yutaka Ishikawa. Overhead of using spare nodes. *The Int. Journal of High Performance Computing Applications (IJHPCA)*, 34(2):208–226, 2020. doi:10.1177/1094342020901885.

[55] Christian Engelmann and Al Geist. Super-scalable algorithms for computing on 100,000 processors. In *Computational Science*, pages 313–321. Springer, 2005. doi:10.1007/11428831_39.

[56] Nawab Ali, Sriram Krishnamoorthy, Mahantesh Halappanavar, and Jeff Daily. Multi-fault tolerance for cartesian data distributions. *Int. Journal of Parallel Programming (JPDC)*, 41(3):469–493, 2012. doi:10.1007/s10766-012-0218-5.

[57] George Bosilca, Aurélien Bouteiller, Thomas Herault, Yves Robert, and Jack Dongarra. Composing resilience techniques: ABFT, periodic and incremental checkpointing. *Int. Journal of Networking and Computing (IJNC)*, 5(1):2–25, 2015. doi:10.15803/ijnc.5.1_2.

[58] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. ACM, 2012. doi:10.1109/SC.2012.36.

[59] Hubertus J. J. van Dam, Abhinav Vishnu, and Wibe A. de Jong. Designing a scalable fault tolerance model for high performance computational chemistry: A case study with coupled cluster perturbative triples. *Journal of Chemical Theory and Computation (JCTCCE)*, 7(1):66–75, 2010. doi:10.1021/ct100439u.

[60] Nuria Losada, Patricia González, Marìa J. Martìn, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Generation Computer Systems (FGCS)*, 106:467–481, 2020. doi:https://doi.org/10.1016/j.future.2020.01.026.

[61] David Grove, Sara S. Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Olivier Tardieu. Failure recovery in resilient X10. *Transactions on Programming Languages and Systems (TOPLAS)*, 41(3):1–40, 2019. doi:10.1145/3332372.

[62] Gengbin Zheng, Lixia Shi, and L.V. Kale. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Proc. Int. Conference on Cluster Computing*, pages 93–103. IEEE, 2004. doi:10.1109/CLUSTR.2004.1392606.

[63] Sri Raj Paul, Akihiro Hayashi, Nicole Slattengren, Hemanth Kolla, Matthew Whitlock, Seonmyeong Bak, Keita Teranishi, Jackson Mayo, and Vivek Sarkar. Enabling resilience in asynchronous many-task programming models. In *Proc. Euro-Par: Parallel Processing*, pages 346–360. Springer, 2019. doi:10.1007/978-3-030-29400-7_25.

[64] Bunjamin Memishi, Shadi Ibrahim, María S. Pérez, and Gabriel Antoniu. Fault tolerance in MapReduce: A survey. In *Computer Communications and Networks*, pages 205–240. Springer, 2016. doi:10.1007/978-3-319-44881-7_11.

[65] Ahcene Bendjoudi, Nouredine Melab, and El-Ghazali Talbi. FTH-B&B: A fault-tolerant hierarchical branch and bound for large scale unreliable environments. *Transactions on Computers*, 63(9):2302–2315, 2014. doi:10.1109/tc.2013.40.

[66] Upama Kabir and Dhrubajyoti Goswami. Identifying patterns towards algorithm based fault tolerance. In *Proc. Int. Conf. on High Performance Computing & Simulation (HPCS)*, pages 508–516. IEEE, 2015. doi:10.1109/hpcsim.2015.7237083.

[67] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A unified engine for big data processing. *Communications of the ACM (CACM)*, 59(11):56–65, 2016. doi:10.1145/2934664.

[68] Mehmet Can Kurt, Sriram Krishnamoorthy, Kunal Agrawal, and Gagan Agrawal. Fault-tolerant dynamic task graph scheduling. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 719–730. ACM, 2014. doi:10.1109/SC.2014.64.

[69] Chongxiao Cao, Thomas Herault, George Bosilca, and Jack Dongarra. Design for a soft error resilient dynamic task-based runtime. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 765–774. IEEE, 2015. doi:10.1109/ipdps.2015.81.

[70] Omer Subasi, Tatiana Martsinkevich, Ferad Zyulkyarov, Osman Unsal, Jesus Labarta, and Franck Cappello. Unified fault-tolerance framework for hybrid task-parallel message-passing applications. *The Int. Journal of High Performance Computing Applications (IJHPCA)*, 32(5):641–657, 2018. doi:10.1177/1094342016669416.

[71] Wenjing Ma and Sriram Krishnamoorthy. Data-driven fault tolerance for work stealing computations. In *Proc. Int. Conf. on Supercomputing (ICS)*, pages 79–90. ACM, 2012. doi:10.1145/2304576.2304589.

[72] Claudia Fohry, Jonas Posner, and Lukas Reitz. A selective and incremental backup scheme for task pools. In *Proc. Int. Conf. on High Performance Computing & Simulation (HPCS)*, pages 621–628. IEEE, 2018. doi:10.1109/HPCS.2018.00103.

[73] Claudia Fohry. Checkpointing and localized recovery for nested fork-join programs. In *Int. Symp. on Checkpointing for Supercomputing (SuperCheck)*, 2021. URL: https://arxiv.org/abs/2102.12941.

[74] Peter Pirkelbauer, Amalee Wilson, Christina Peterson, and Damian Dechev. Blaze-Tasks: A framework for computing parallel reductions over tasks. *ACM Trans. on Architecture and Code Optimization (TACO)*, 15(4):66:1–66:25, 2019. doi:10.1145/3293448.

[75] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems (FGCS)*, 22(3):303–312, 2006. doi:10.1016/j.future.2004.11.016.

[76] Anne Benoit, Thomas Herault, Valentin Le Fèvre, and Yves Robert. Replication is more efficient than you think. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–14. ACM, 2019. doi:10.1145/3295500.3356171.

[77] Thomas Herault, Yves Robert, Aurélien Bouteiller, Dorian Arnold, Kurt Ferreira, George Bosilca, and Jack Dongarra. Checkpointing strategies for shared high-performance computing platforms. *Int. Journal of Networking and Computing (IJNC)*, 9(1):28–52, 2019. doi:10.15803/ijnc.9.1_28.

[78] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kale. A batch system with efficient adaptive scheduling for malleable and evolving applications. In *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 429–438. IEEE, 2015. `doi:10.1109/IPDPS.2015.34`.