SN Computer Science (2024) 5:320
https://doi.org/10.1007/s42979-024-02624-8

ORIGINAL RESEARCH

Task-Level Checkpointing and Localized Recovery to Tolerate Permanent Node Failures for Nested Fork–Join Programs in Clusters

Lukas Reitz[1] · **Claudia Fohry**[1]

Received: 2 November 2023 / Accepted: 10 January 2024

Abstract
Exascale supercomputers consist of millions of processing units, and this number is still growing. Therefore, hardware failures, such as permanent node failures, become increasingly frequent. They can be tolerated with system-level Checkpoint/Restart, which saves the whole application state transparently and, if needed, restarts the application from the saved state; or with application-level checkpointing, which saves only relevant data via explicit calls in the program. The former approach requires no additional programming expense, whereas the latter is more efficient and allows to continue program execution after failures on the intact resources (localized shrinking recovery). An increasingly popular programming paradigm is asynchronous many-task (AMT) programming. Here, programmers identify parallel tasks, and a runtime system assigns the tasks to worker threads. Since tasks have clearly defined interfaces, the runtime system can automatically extract and save their interface data. This approach, called task-level checkpointing (TC), combines the respective strengths of system-level and application-level checkpointing. AMTs come in many variants, and so far, TC has only been applied to a few, rather simple variants. This paper considers TC for a different AMT variant: nested fork–join (NFJ) programs that run on clusters of multicore nodes under work stealing. We present the first TC scheme for this setting. It performs a localized shrinking recovery and can handle multiple node failures. In experiments with four benchmarks, we observed execution time overheads of around 44 % at 1536 workers, and negligible recovery costs. Additionally, we developed and experimentally validated a prediction model for the running times of the scheme.

Keywords Asynchronous many-task programming · Fault tolerance · Task-level checkpointing · Work stealing

Introduction

Modern supercomputers have reached Exascale and consist of millions of processing units. For instance, the Frontier machine has over 8 million cores [23]. With an increasing component count, hardware failures become more frequent [17]. Recent studies estimate permanent node failures in supercomputers with a million cores to occur between every 5 and 53 min [4, 27].

A popular approach to tolerate hardware failures is system-level *Checkpoint/Restart (C/R)*. It transparently saves the whole application state periodically at global synchronization points and, when a hardware failure is detected, restarts the application from the last saved checkpoint [45]. System-level C/R incurs a significant running time overhead due to the synchronization and I/O to the cluster file system [32]. Another well-known approach is *application-level checkpointing*. Here, the programmer inserts function calls into the code to save only relevant data. Application-level

This is an extended version of reference [40], which built upon a preliminary paper [10].

This article is part of the topical collection "Applications and Frameworks using the Asynchronous Many Task Paradigm" guest edited by Patrick Diehl, Hartmut Kaiser, Peter Thoman, Steven R. Brandt and "Ram" Ramanujam.

✉ Lukas Reitz
lukas.reitz@uni-kassel.de

Claudia Fohry
fohry@uni-kassel.de

[1] Research Group Programming Languages/Methodologies, University of Kassel, Wilhelmshöher Allee 71-73, 34121 Kassel, Germany

Published online: 13 March 2024

checkpointing is more efficient than system-level C/R, but requires additional programming effort. Several application-level recovery schemes permit to continue running the application after failure on the intact resources, which is called *shrinking recovery*, and/or to confine the failure handling to directly affected resources, which is called *localized recovery* (e.g., [26, 35]).

While system-level and application-level checkpointing have different pros and cons, an intermediate approach, called *task-level checkpointing* (TC), promises to achieve transparency, efficiency, and a localized shrinking recovery together. Unlike the above general-purpose approaches, TC is specialized to Asynchronous Many-Task (AMT) programs. AMT is an increasingly popular programming paradigm with examples, including HPX [19], OpenMP tasks [30], Chapel [6], and Cilk [5]. AMT programs partition the computation into units, called *tasks*, and a runtime system maps the tasks to lower level resources. TC operates in the runtime system. By exploiting the clearly defined interfaces of tasks, it automatically saves task descriptors and interface data.

Current AMT environments differ widely in their *task models*, i.e., in the mechanisms for task generation and cooperation [9, 14, 49]. Examples include side effect-based task cooperation (such as in Chapel [6]), Sequential Task Flow (such as in StarPU [2]), Dynamic Independent Tasks (such as in GLB [53]), and Nested Fork–Join (such as in OpenCilk [42] and Nowa [43]). Moreover, the AMTs differ in their target architectures and in the runtime algorithms for task assignment. Thus far, TC has only been studied for a few, rather simple settings (e.g., [26, 36]).

This paper proposes a TC scheme for a new setting, namely Nested Fork–Join programs running on clusters with multi-worker processes under work stealing. To explain these terms, *Nested Fork–Join (NFJ)* programs begin the computation with a single task, which eventually returns the final result. Then, each task may spawn child tasks that return their respective results to the parent. The next term, *work stealing*, refers to *workers* denoting the compute resources that process the tasks. In our setting, workers are threads of multiple processes on different cluster nodes. Each worker maintains a *task queue*, from which it takes tasks for processing and into which it inserts newly generated tasks. If the queue is empty, the worker becomes a *thief* and tries to steal tasks from another worker, called *victim*. We consider a recent, efficient variant of work stealing in multicore clusters, called the *lifeline-pure* scheme [39].

A core challenge for our development of a TC scheme for NFJ was keeping the checkpoints consistent despite stealing-related task migration. For that, we built on a previous TC scheme for Dynamic Independent Tasks (DIT), called AllFT [35]. Major changes of AllFT were required, because: 1) NFJ differs from DIT insofar as NFJ tasks return their results to the parent, whereas DIT environments directly compute the final result by reduction from the task results. 2) AllFT refers to a help-first work-stealing policy, in which parent tasks are processed before child tasks, whereas our setting uses a work-first policy, in which the child tasks are processed first. 3) We consider multi-worker processes, whereas AllFT is restricted to single-worker processes.

Like AllFT, our scheme can handle any number of permanent worker failures, including simultaneous failures and failures during recovery. Failures never compromise a returned result, but in a few rare cases, the program aborts. To the best of our knowledge, our scheme is the first TC scheme for NFJ, and at the same time, it is the first TC scheme for multi-worker processes under work stealing.

We implemented and experimentally evaluated the scheme. In experiments on two clusters with up to 1280 and 1536 workers, respectively, we observed fault-tolerance overheads of up to 28.3 % and up to 43.98 %, respectively. Thereby the recovery costs were negligible. These observed fault-tolerance overheads are higher than those of the AllFT scheme for DIT, but well below those of C/R [32].

The remainder of this paper is organized as follows. "Background" describes NFJ, the lifeline-pure scheme, and AllFT. Then, "Task-Level Checkpointing for NFJ" presents our new TC scheme, and "Implementation" sketches its implementation. Experimental results are provided and discussed in "Experiments". Thereafter, "Prediction of Running Times" derives a formula to predict running times of our TC scheme and compares them to measured running times. The paper ends with related work and conclusions in "Related Work" and Conclusions, respectively.

Background

Nested Fork–Join Programs (NFJ)

Listing 1 depicts pseudocode of a naive recursive Fibonacci program in NFJ. The computation begins with a root task computing `f(n)` for a given `n`. Then, each task spawns two child tasks whose results are assigned to variables `a` and `b`. At the `sync` statement, the parent task waits until all previous assignments have been performed. Beyond the example, NFJ programs contain an implicit `sync` at the end of each task function. Furthermore, we assume the tasks to be free of side effects.

Listing 1: NFJ example: naive recursive Fibonacci implementation

```
f(n) {                          // 0
  if (n < 2) return 1;
  a = spawn f(n-1);             // 1
  b = spawn f(n-2);             // 2
  sync;                         // 3
  return a + b;
}
```

The execution of an NFJ program gives rise to a tree, such as the one for the Fibonacci example in Fig. 1. In the figure, rounded rectangles denote functions spawned as tasks. Numbers 0 to 3 correspond to the sequential code sections marked in Listing 1. For instance, section 0 runs from the beginning of the function until the spawn of the first child. Downward edges (solid) mark spawns, and upward edges (dotted) mark result returns at explicit or implicit sync's.

Lifeline-Pure Scheme

As noted in "Introduction", the tasks are executed by a set of worker threads from multiple processes. Each worker owns a *task queue*, in which it saves task descriptors, which are continuations. Thereby, *continuation* denotes the remaining computation of a function together with variable values. Continuations are represented by stack *frames*.

The lifeline-pure scheme [39] is an efficient work-stealing variant for, e.g., NFJ, that adapts lifeline-based global load balancing [41] to multi-worker processes. It uses *cooperative* work stealing, i.e., a thief sends a steal request to the victim, and the victim actively responds with either *loot* or a reject message. All communication is realized with active messages. Workers answer steal requests periodically after every *k* processed tasks. In our case, the loot is always a single task, namely the oldest task from the victim pool. In the lifeline-pure scheme, a thief first attempts to steal from up to *w* random victims, and then, if not successful, from up to *z* *lifeline buddies* [39]. These are preselected victims, which remember any unsuccessful *lifeline steal requests*. If they obtain tasks later, they share these tasks with the thief. After *z* unsuccessful lifeline steal requests, a thief becomes inactive. It is reactivated when a lifeline buddy later sends tasks. The program ends when the root task has finished.

The lifeline-pure scheme deploys the *work-first* policy, in which a worker encountering a spawn branches into the child task and puts a continuation of the parent task into its queue. Figure 2 illustrates work-first work stealing. The notation is the same as in Fig. 1, but the task structure differs to facilitate further discussion. Each color marks the work performed by a particular worker.

The computation starts with the green worker (called Green) processing the A function. At the first spawn, Green branches into child task B, and Brown later steals the continuation of A, which at that time encompasses A2 to A6 and is represented by A2. In general, thieves process parent frames, and victims process children, as illustrated on the right side of the figure.

The lifeline-pure scheme matches child results with their parent frames in the same way as in reference [20]: When a thief encounters a sync, it sends the frame back to the victim (or transitively to all victims), where the actual matching is accomplished with the help of a frame identifier. Note that the parent frame is sent back to the child (and not vice versa), even though the dotted arrows in Fig. 2 indicate that, logically, the result is incorporated into the parent. When a victim finishes a task whose parent is away, it locally saves the result and steals a new task.

For an example, consider Red in Fig. 2. It stole frame B from Green at B2, and was stolen from by Yellow at B3.

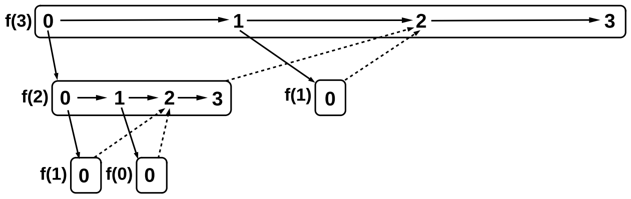


Fig. 1 Execution of a nested fork–join program

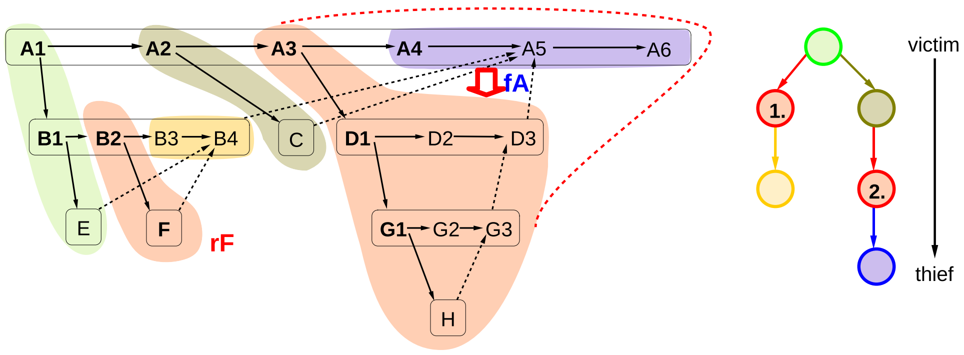


Fig. 2 Work stealing under the work-first policy

Red finished F before Yellow returned the frame. Therefore, Red kept the result (called `rF`) and stole the A frame from Brown at A3. Later, Blue stole the A frame at A4 and already returned it (called `fA`) at the `sync` opening A5. Thereby Blue returned the frame to Red, because Red was the most recent victim. The dotted red line indicates that `fA` resides at Red again. After having inserted the result from D, Red will return the frame to Brown, and later Brown will return it to Green. Green will insert the result from the B function, and afterwards, Green will continue at A6.

Now, consider the time when A has finished all sections printed in bold and is going to branch into H. At this time, Red holds `rF`, `fA`, a local pool with the continuations starting at D2 and G2, and a descriptor of H. Furthermore, Red knows the identities of all victims and thieves with still unmatched results: Green and Yellow for the B2 frame, and Brown for the A3 frame. In its entirety, this information forms Red's *state*.

Original Task-Level Checkpointing and Recovery Scheme for DIT

We adapted the AllFT scheme from Posner et al. [35] to our NFJ setting. The original AllFT scheme encompasses a checkpointing procedure, a steal protocol, a restore protocol, and a selection scheme for buddy workers:

Checkpointing is performed independently by each worker. It always writes them after having finished one task and before starting the next one, more specifically in the following situations:

- every about *R* seconds, called *regular backups*,
- during stealing at the victim and thief sides, called *steal backups*, and
- during restore, called *restore backups*.

Each backup contains the worker state and some *status information*. The latter is a compact representation of all pieces of loot that have been sent and received so far by this worker (see [33]). In the DIT context, the worker state consists of the contents of the worker's task queue and the worker's current contribution to the final result, called *worker result*. Task descriptors in the DIT setting are just task parameters. The worker result is the combined result of all tasks that have been processed by this worker so far, e.g., the sum of these results. The checkpoints are saved in a *resilient store*, for which AllFT requires that it must support the grouping of multiple access operations into transactions.

The AllFT *steal protocol* ensures consistency between a victim, a thief, and their respective backups, despite possible failures. In addition to the steal backups, the protocol involves a temporary loot saving in the resilient store. While being saved during the protocol, the loot is denoted as *open*. Each worker numbers the loot that it sent in ascending order and attaches the number to the loot, called *loot identifier*.

The *restore protocol* presupposes that all workers are informed about failures, although possibly at different times. One designated partner worker B, called *buddy*, adopts the tasks from the checkpoint of a failed worker X and makes sure that open loot sent *from* X is taken over by either the thief or by B. Open loot sent *to* X is taken care of by the victim. For that, each worker looks up whether it has been a victim of X and, if so, re-inserts any open loot into its own queue, deletes it from the resilient store, and writes a restore backup. As an example, the deletion of the open loot and the writing of the restore backup are grouped into a transaction to ensure consistency in case of an intermediate failure of X. Altogether, the restore protocol guarantees that each task is processed exactly once.

The *selection scheme for buddy workers* is based on a consecutive numbering of the workers with wraparound. The *buddy* of a failed worker is defined as the next worker alive in this *ring*. Buddy selection is still nontrivial, since AllFT allows simultaneous failures and failures during recovery. Thus, the role of buddy can move during an ongoing recovery. Details and a case-by-case analysis of correctness are given in reference [11].

Task-Level Checkpointing for NFJ

We applied the following major changes, which are further explained below:

1. We defined the state of an NFJ worker and modified the contents of checkpoints, so that they save this state in addition to the status information. Also, we clarified the times of checkpoint writing.
2. We newly added a frame return protocol and extended the restore protocol by the adoption of task results and frames (such as `rF` and `fA`).
3. We adapted the buddy worker selection scheme to multi-worker processes.

Regarding item 1, we define the *state* of a worker W to comprise the following information. Examples refer to "Lifeline-Pure Scheme" and Fig. 2:

- the current contents of W's local pool (e.g., a queue with the task descriptors of D2 and G2),
- all locally saved task results of W that are yet to be incorporated into their parent frame (e.g., `rF`),
- all frames returned to W from their thieves that are awaiting result incorporation (e.g., `fA`),

- the identities of all victims of W to which W has yet to return the respective stolen frame (e.g., Green and Brown),
- the identities of all thieves of W that have yet to return their frame to W (e.g., Yellow), and
- a task descriptor of the next task (if relevant, see below).

Like in AllFT, checkpoints are only written when the worker is outside task processing. Thus, there are three possible occasions for checkpoint writing:

A. At a spawn: Before branching into the child (e.g., before branching into H).
B. At a sync: Right before returning the frame to the child (e.g., right before returning fA).
C. At the end of a function, which gives rise to two sub-cases:

  C1. After the function had been finished and its result was incorporated into the parent frame (e.g., after H had been finished and its result was incorporated into the G frame)
  C2. After the function had been finished and its result was stored locally (e.g., after F had been finished and `rF` was stored).

The last item from the above definition of worker state is relevant in occasion A, but not in occasions B and C. In occasions B and C, the next task will either be taken from the local queue, and is then part of the state anyway, or be stolen anew, and is then contained in the checkpoint that is written during the steal protocol.

Checkpoints contain the newly defined worker state and the same status information as in AllFT (see "Original Task-Level Checkpointing and Recovery Scheme for DIT"). Checkpoints are saved in a resilient store with the same requirements as for AllFT.

The steal protocol is almost identical to that in AllFT, since the handshaking to reach consistency is independent of the contents of checkpoints. We merely added bookkeeping for the identities of victims and thieves. For completeness, we depict the protocol in Fig. 3. It shows a successful random steal. The wavy and dashed lines indicate time the worker spends on task processing and waiting, respectively. The resilient store is briefly abbreviated as *resiStore*. The figure begins at the top with a victim, who is busy processing tasks, and a thief, who has no tasks and thus attempts to steal (here: from Victim) and waits for an answer. Steal requests are active messages, whose action is written next to the wavy line. The messages are processed in parallel to the victim thread by another thread of the victim process. If the victim pool is empty, this thread immediately sends back a reject message; otherwise, it

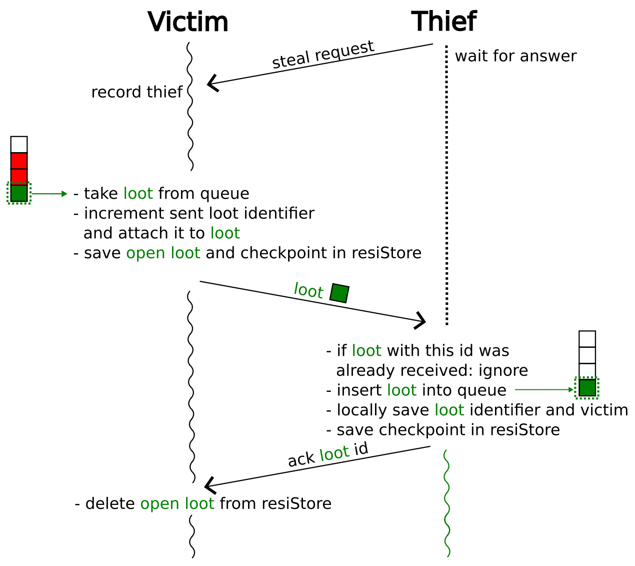

Victim
Thief
steal request
wait for answer
record thief
- take loot from queue
- increment sent loot identifier and attach it to loot
- save open loot and checkpoint in resiStore
loot
- if loot with this id was already received: ignore
- insert loot into queue
- locally save loot identifier and victim
- save checkpoint in resiStore
ack loot id
- delete open loot from resiStore

Fig. 3 Example of the steal protocol with a successful random steal

records the request. The victim occasionally interrupts its task processing to react to the recorded messages. In our example, the victim has tasks in its queue. Therefore, it takes out the oldest task, increments and attaches the loot identifier, saves the open loot and its checkpoint in the resilient store, and sends the loot to the thief. On arrival of the loot, the thief compares the loot identifier with the last received loot identifier from the same victim, and ignores the loot if the thief already received it (can happen during recovery, see below). Normally, the thief inserts the loot into its queue, locally saves the loot identifier and the victim, saves a backup in the resilient store, acknowledges the receipt, and begins to process the loot. On arrival of the acknowledgement, the victim deletes the open loot from the resilient store.

Whereas in the AllFT scheme for DIT, worker results are local and just have to be included in checkpoints, in NFJ, we must deal with consistency regarding the results. Recall that, for result matching, frames are sent back to the victims. This leads to a similar consistency problem as in stealing insofar as data (loot or a frame, respectively) are moved from one worker to another. We solve the problem with a *frame return protocol*, which closely resembles the steal protocol, except that it is initiated by the sender (there is no preceding steal request).

Figure 4 illustrates our new frame return protocol. It shows the case of two consecutive frame returns of a parent frame, first by Thief 1 and then by Thief 0. The protocol starts when Thief 1 encounters a `sync` in some task. Then, Thief 1 writes a checkpoint to the resilient store, called *frame return backup*, and additionally saves the continuation (i.e., a frame) of the corresponding task there temporarily. In analogy to open loot, the frame is denoted as *open*

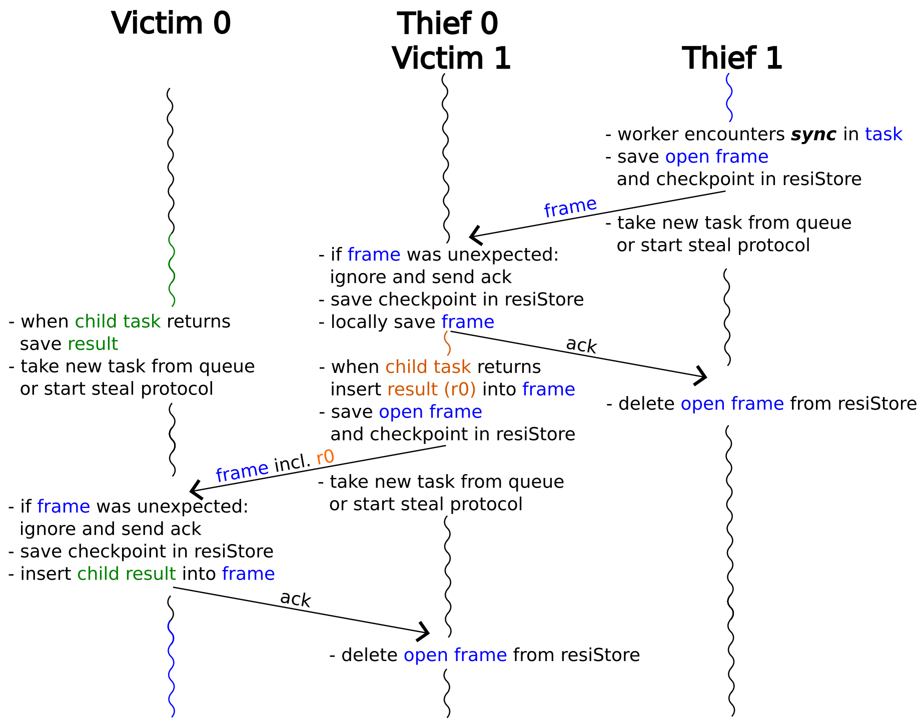

Victim 0
Thief 0
Victim 1
Thief 1
- worker encounters ***sync*** in task
- save open frame and checkpoint in resiStore
frame
- take new task from queue or start steal protocol
- if frame was unexpected: ignore and send ack
- save checkpoint in resiStore
- locally save frame
ack
- when child task returns save result
- take new task from queue or start steal protocol
- when child task returns insert result (r0) into frame
- save open frame and checkpoint in resiStore
- delete open frame from resiStore
frame incl. r0
- take new task from queue or start steal protocol
- if frame was unexpected: ignore and send ack
- save checkpoint in resiStore
- insert child result into frame
ack
- delete open frame from resiStore

Fig. 4 Example of the frame return protocol with a frame that was stolen two times: initially from Victim 0 by Thief 0, and then from Victim 1 by Thief 1

while being kept in the resilient store. Next, Thief 1 sends the frame to the last victim (here: to Victim 1) and continues task processing with a new task.

If a last victim does not expect the frame, which may happen during restore (see below), the frame is ignored. In our example, the frame is expected, and so, Victim 1 saves a checkpoint, which includes the frame, in the resilient store, and sends an acknowledgement to Thief 1. This acknowledgement would also be sent for ignored frames for which the protocol would end afterward.

The figure shows a case where Victim 1 is still processing the child task when the frame arrives. Therefore, Victim 1 saves the frame locally. When the child task eventually finishes, Victim 1 looks up the frame and inserts the child task result. Thereby it recognizes that the frame expects another result, and therefore, Victim 1 becomes Thief 0 and initiates another frame return protocol analogously.

For the second frame return, the figure shows a different case where Victim 0 has already finished the child task. Therefore, Victim 0 inserts the result right after the frame receipt. Victim 0 recognizes that the frame has all needed results, and so, Victim 0 inserts the frame at the bottom of its queue.

The *restore protocol* extends that of AllFT. Let X denote a failed worker, and B the buddy of X, respectively. Then, like in AllFT, B adopts all tasks from X's checkpoint and any open loot sent *from* X. Additionally, now, B adopts all task results and frames from X's checkpoint. B just stores them alongside its own results and frames: The results will eventually be located by their associated thieves (see below); and the frames are awaiting results from X's tasks, which have been adopted by B as well. When learning about X's failure, victims that are awaiting a frame return from X inspect the resilient store to see if the frame is open. If so, they take it, and thereby send the acknowledgement to B instead of X.

Frames sent *to* X are handled by the sender, as is open loot sent to X in AllFT. Unlike the loot, however, the frames cannot be taken back by the sender. Instead, the sender delivers them to the owner of the associated results, which is B. The sender locates B as being the next worker alive in the ring of workers. Frames to X that are finished later are handled the same way. In any case, if a frame was already received, it is ignored by the victim.

AllFT assumes all workers are equal when defining the ring. For our multi-worker processes, we number the workers in the ring blockwise: workers of process 0 get numbers $0 \dots d-1$, workers of process 1 get numbers $d \dots 2d-1$, etc. While individual worker failures can be handled, usually it is processes that fail. A process failure is handled like a simultaneous failure of multiple workers. By definition, the buddy of all of these workers is the first worker of the next process. As usual, it is responsible for the restore of all failed workers. We chose the above numbering scheme, although it may create some temporary load imbalances, since it enables many other workers to continue task processing despite the failure. Senders of loot/frames to a failed worker recognize that a whole process has failed and avoid trying every single worker in the search for the buddy.

Implementation

Our implementation is based on the APGAS for Java library [48] for programming distributed parallel applications, and on a resilient store, called IMap, from Hazelcast [16]. The APGAS library provides methods to send active messages, which are processed by threads of the Java fork–join pool [24]. Thus, they can be handled in parallel to task processing at the receiver. Further, the APGAS library provides failure notifications. More specifically, it invokes a *failure handler* on all processes that have registered for the service. We register all processes. Each of the failure handlers decides on the role of all of its workers during restore and, if needed, initiates the restore protocol.

The IMap internally saves workers and their checkpoints as key-value pairs, groups the pairs into partitions, and evenly distributes the partitions over nodes. We configured the IMap, so that the checkpoints of all workers from the same process are mapped to the same partition, to reduce network communication during restore. Moreover, the IMap supports transactions, as it was required in Original Task-Level Checkpointing and Recovery Schemefor DIT. The IMap works with at most six replicas of each partition, and we configured it with one replica. If more cluster nodes than the number of replicas fail simultaneously or in close succession, a checkpoint may be irrecoverably lost and the program aborts. This, as well as the loss of *all* workers are the only occasions in which our TC scheme aborts. AllFT, in contrast, also aborts after failure of worker 0, due to differences in termination detection between DIT and NFJ.

The source code of our implementation is available online [38].

Experiments

Experimental Setting

We evaluated our scheme with four benchmarks:

- FIB: The Fibonacci benchmark resembles that from "Background" for $n = 67$, except that we spawn a task for only one of the two recursive function calls.
- UTS: The Unbalanced Tree Search benchmark generates an irregular tree, using some statistical method that allows to control the number of child nodes. The tree is not stored, but its number of nodes is counted on-the-fly. We spawn a task for each tree node. We used a geometric distribution with expected value $b = 4$ for the distribution of the number of child nodes, initial seed $s = 19$ for a pseudorandom number generator, and tree height $d = 19$.
- NQ: The N-Queens benchmark counts the number of valid placements of n queens on an $n \times n$ chessboard, such that no two queens can attack each other. The computation begins with one task which operates on the first column of the chessboard. Each task loops through the fields of its column and attempts to place a queen on it. If successful, it spawns a task for the next column. We used $n = 18$.
- SYN: The synthetic benchmark recursively spawns tasks, so that they form a perfect w-ary task tree. Each task runs a dummy computation with configurable duration and then reports back to its parent [36]. We configured $m = 10^6$ tasks per worker with total duration $T_{\text{calc}} = 100$ s, and $v = 20\,\%$ load variance between workers.

We used the existing FIB, UTS, and SYN implementations from reference [39], and implemented NQ from scratch.

FIB, UTS, and NQ allow setting a sequential cut-off (C), which is a problem size threshold at which the `spawn` keyword is ignored: for FIB, it refers to calls with $n < C$, for UTS, it refers to all tree nodes with a depth greater than C, and for NQ, it refers to tasks with at most C unplaced queens. Like in reference [39], we set $C = 30$ for FIB, $C = 13$ for UTS, and we found $C = 6$ to perform well for NQ. Recall that parameter k denotes the number of tasks that a worker processes before answering steal requests. Like in reference [39], we set $k = 10$ for FIB, $k = 16$ for UTS, and $k = 1$ for NQ and SYN. We set $R = 10$ s, like in reference [35].

Experiments were conducted on two clusters:

Goethe: We used a partition of the Goethe cluster of the University of Frankfurt [50], which consists of homogeneous Infiniband-connected nodes, each with two 20-core Intel Xeon Skylake Gold 6148 CPUs and 192 GB of main memory. We assigned one process with 40 workers to each cluster node.

Lichtenberg: We used a partition of the Lichtenberg cluster of the Technical University of Darmstadt [51], which consists of homogeneous Infiniband-connected nodes, each with two 48-core Intel Xeon Platinum 9242 CPUs and 384 GB of main memory. We assigned one process with 96 workers to each cluster node.

On both clusters, we used Java version 19.0.2 and the APGAS for Java library from [18].

Failure-Free Runs

Figure 5 depicts the performance of failure-free FIB, UTS, NQ, and SYN executions with and without protection on

Table 1 Average running times in seconds with and without failure protection on Goethe

Benchmark	Protection	Workers					
		40	80	160	320	640	1280
FIB	Protected	3532.39	1798.49	907.52	464.96	244.30	144.31
	Unprotected	3523.55	1780.05	882.66	447.40	225.07	113.65
UTS	Protected	2628.18	1372.52	707.35	370.94	200.20	114.25
	Unprotected	2612.21	1342.14	696.51	355.36	176.22	89.06
NQ	Protected	2492.58	1300.79	666.83	347.50	184.49	107.01
	Unprotected	2477.38	1284.32	652.76	333.88	167.67	85.38
SYN	Protected	104.70	106.32	109.05	113.55	118.11	132.93
	Unprotected	102.97	102.71	104.13	103.32	103.54	104.27

Table 2 Average running times in seconds with and without failure protection on Lichtenberg

Benchmark	Protection	Workers				
		96	192	384	768	1536
FIB	Protected	1957.49	987.96	534.15	285.02	179.67
	Unprotected	1938.16	959.80	497.94	248.28	126.05
UTS	Protected	1446.45	719.23	369.81	205.24	126.50
	Unprotected	1422.35	699.10	343.46	176.73	89.6
NQ	Protected	1419.61	750.29	399.22	236.11	139.64
	Unprotected	1405.37	713.30	368.20	202.63	98.74
SYN	Protected	105.40	110.27	116.74	122.86	150.98
	Unprotected	103.00	103.60	103.93	104.18	104.86

Goethe and Lichtenberg. Corresponding raw data are given in Tables 1 and 2. All values are averages over 10 runs.

Figures 5a–c depict running times. They employ strong scaling to give an impression of magnitudes. In all cases, the costs for protection increase with the number of workers. On Goethe at 1280 workers, the difference in running times of protected vs unprotected runs is 144.3 s vs 113.7 s (FIB), 114.3 s vs 89.1 s (UTS), and 107.01 s vs 85.38 s (NQ). On Lichtenberg at 1536 workers, it is 179.67 s vs 126.05 s (FIB), 126.5 s vs 89.6 s (UTS), and 139.64 s vs 98.74 s (NQ).

Figure 5d depicts the difference between measured running times and T_{calc}, called *overall runtime overhead*, which includes both the costs of load balancing and the costs of protection. One can see that this overhead increases with the number of workers up to 32.9 % on Goethe and up to 50.98 % on Lichtenberg. For comparison, the pure work stealing overhead in unprotected runs is up to 4.27 % on Goethe and up to 5.45 % on Lichtenberg.

The results from Fig. 5 are summarized in Fig. 6. The figure shows the protection overhead, which is the quotient of the running times of protected runs over unprotected runs minus one as percentage. As can be seen in the figure, the protection overhead curves are similar for the four benchmarks, with a maximum of 28.3 % at 1280 workers on Goethe, and 43.98 % at 1536 workers on Lichtenberg.

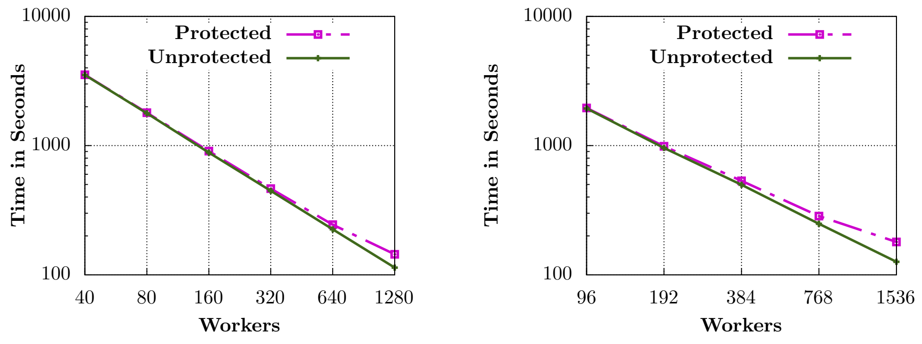
These numbers are roughly in line with previously reported protection overheads for AllFT. To see this, observe that our protection overheads can be expected to be roughly twice of those of AllFT, since each AllFT steal gives rise to the execution of a steal protocol, whereas each of our steals gives rise to the execution of both a steal and a frame return protocol. Thus, the number of checkpoint writings, which are the most expensive operations, approximately doubles.

The most recent performance evaluation of AllFT is given in [36]. This reference only reports protection overheads for SYN with up to 144 workers on a different cluster. Additionally, it reports running times for UTS with up to 640 workers on Goethe. We used the raw data of these running times to calculate the protection overhead of AllFT at 640 workers on Goethe. It is about 5.5 %. Comparing this number with the protection overhead of our new scheme at 640 workers on Goethe, which is 13.6 %, our expectation of a factor of two is approximately met. The remaining difference can be attributed to a different design of the data structures (e.g., larger task descriptors in NFJ vs DIT), and to a slightly less sophisticated implementation.

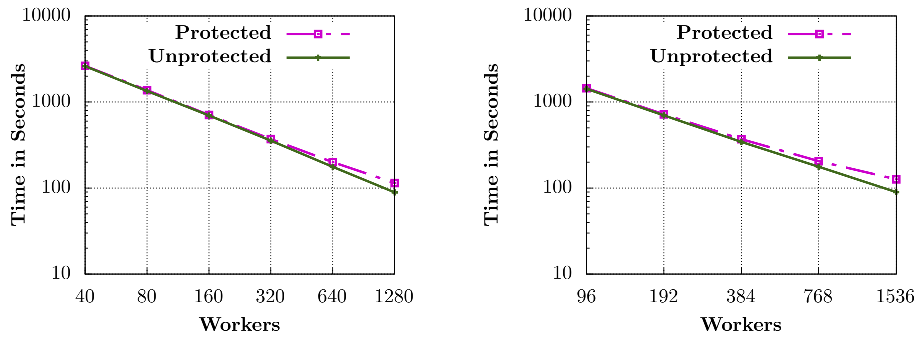
Estimation of Restore Overhead

In a second group of experiments, we estimated the restore overhead after worker failures. This overhead includes the times for: failure detection, execution of the restore protocol, and reprocessing of the lost tasks. It does not include the increase in running time that is due to our usage of a shrinking recovery, i.e., to the use of less resources after the failure.

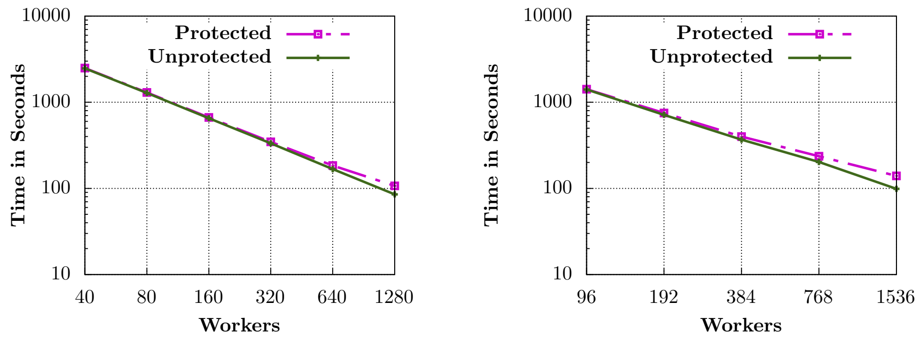
Fig. 5 Performance in failure-free runs on Goethe (left) and Lichtenberg (right)



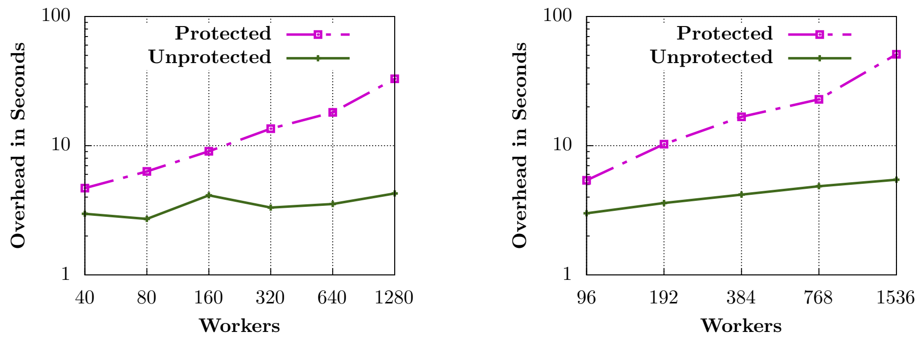
(a) Running times of FIB



(b) Running times of UTS



(c) Running times of NQ



(d) Overall runtime overhead of SYN

We estimated the overhead with the help of a methodology from reference [35]. For that, we measured the running times of three program executions:

(A) with 640 workers,
(B) with 600 workers, and

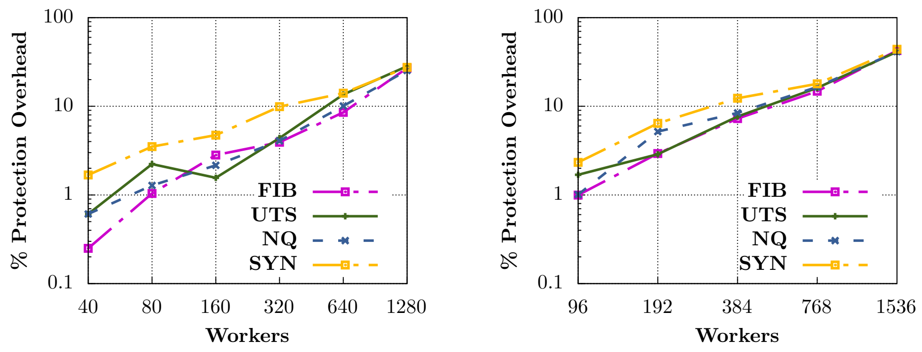


Fig. 6 Protection overheads of the benchmarks in failure-free execution on Goethe (left) and Lichtenberg (right)

(C) with 640 workers, of which we crashed 80 workers (2 processes) after half of the expected running time by calling `System.exit()`. Thereby, the expected running time was taken from execution (A) (245.23 s).

Because executions (B) and (C) use the same average number of workers, we roughly estimated the restore overhead as the difference between the running times of executions (C) (291.28 s) and (B) (277.38 s). All numbers are averages over 50 runs of each execution.

As can be seen from the numbers, the restore overhead is about 5 % of the running time. Since this overhead refers to the failure of 80 workers, the overhead of a single-worker failure is negligible.

Correctness

To verify that our TC scheme can handle any number of worker failures, regardless of their timing, we injected process failures by calling `System.exit()` in particularly difficult situations. The test cases were selected by picking the hardest cases from a list of test cases for AllFT in reference [33], and adding new, similar cases to test the frame return protocol. In the following list of test cases, X denotes a random worker that was crashed so as to fulfill the test case. In addition, we always crashed all workers of X's process, and in some cases also the original buddy Y of X. The X worker was crashed:

1. after X had written its first regular backup,
2. after X had sent the acknowledgement during the steal protocol,
3. after X had sent loot, but before the acknowledgement arrived,
4. after X had written the victim side steal backup, but before X sent the loot,
5. after X had saved open loot to the IMap, but before X saved its checkpoint,
6. after X had sent loot, and we delayed the thief side steal protocol until recovery (for all failed workers) was complete,
7. after X had saved an open frame to the IMap, but before X saved its checkpoint,
8. after X had written the victim side frame return backup, but before X sent the frame return message,
9. after X had written the victim side frame return backup, but before X sent the acknowledgement,
10. after X had sent a frame return message, and we delayed the execution of the victim side frame return protocol until recovery (for all failed workers) was complete,
11. at a random time, and
12. at a random time, and we additionally crashed Y at the beginning of the restore protocol.

In an additional test case, we crashed 90 % of all processes to test the abort of the computation with an error message.

We run our correctness tests on Goethe with 640 workers using SYN. Each of the above tests was repeated 25 times. Our observation of the program log files showed that all program executions behaved correctly.

Prediction of Running Times

This section derives a formula to predict the running times of runs with multiple worker failures. Afterwards, we validate the formula by comparing its estimates with measured running times on Lichtenberg. For generality, the formula refers to worker failures, but it can be easily extended to process failures by multiplying the number of failures with the number of workers per process.

The formula is derived with the same methodology as in reference [36]. Generally speaking, the formula takes into account three causes of running time increase: protection costs (as in "Failure-Free Runs"), restore overhead (as in "Estimation of Restore Overhead"), and the reduction in the number of available resources due to shrinking recovery (not previously considered). We assume that the failures are uniformly distributed, i.e., a failure occurs with equal probability at any time during the program execution. The following notation is used:

- p: number of workers,
- $x \ll p$: number of worker failures,
- j: number of steals per worker per second,
- r: number of regular backups per worker per second,
- $T^N O(p)$: average running time of unprotected executions, and
- $T^F T_x(p)$: average running time of protected executions with $x \geq 0$ worker failures.

Let us start with the protection costs. They include the time for regular backup writing, and the time spent in the steal and frame return protocols. The time to write a regular backup is about constant (for a given benchmark), and we denote it by c_0. We calculate the time spent in the protocols per steal. Per steal, four backups are written, namely two in the steal protocol and two in the frame return protocol, and several other actions are performed in the two protocols. Overall, the protection costs per steal are about constant, and we denote them by c_1. Taking the steal and regular backup rates into account, we get an estimate for the running time of failure-free, protected executions

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

Next, we will estimate the restore overhead. It is composed of 1) the costs to run the restore protocol itself, and 2) the costs to re-execute all the tasks that any of the failed workers had processed since its last checkpoint. The protocol costs are about constant, and we denote them by c_2 for each of the x failures. For the estimation of re-execution costs, we first determine the length of the backup interval, which is the average distance between successive backups. Since a worker writes four backups per steal (if we offset the thief and victim sides), plus the regular backups, it writes $4j + r$ backups per second. Reversely, the length of the backup interval is $b = 1/(4j + r)$. On average, a failure occurs at half of this interval. Thus, tasks with duration about $b/2$ need to be reprocessed per failed worker. This reprocessing is shared among the available workers, which are $p - i$ workers after the ith failure. Consequently, the overall restore overhead is

$$\sum_{i=1}^{x} \frac{b}{2(p-i)} + c_2 x.$$

Finally, we will estimate the impact of resource reduction. Due to the uniform distribution of failures, on average, failures occur at half of the running time. Thus about $x/2$ of the overall computing power is lost due to the shrinking recovery. This aspect leads to a proportional increase of the running time to

$$\frac{p}{p-(x/2)}\, T_0^{FT}(p).$$

Putting the above three causes of running time increase together, we obtain our formula

$$T_x^{FT}(p) = \frac{p}{p-(x/2)}\,(1 + c_0 r + c_1 j)\, T^{NO}(p) + \sum_{i=1}^{x} \frac{b}{2(p-i)} + c_2 x. \tag{1}$$

To complete it, we need values for j, r, and c_0 to c_2. These values are benchmark-specific, since, for instance, the costs for backup writing depend on the size of task descriptors.

We experimentally determined the values for the SYN benchmark on Lichtenberg; see Table 3. For this, we modified the code to log operation durations, run the benchmark 25 times with the parameters from "Experiments", and injected a single-worker failure into each run at a random time. The runs used only 192 workers due to time and cluster compute time quota restraints. First, we determined j by summing up the number of steals of each worker (10,163 total steals) and dividing the sum by the number of workers and the average running time (112.11 s). The runs used $R = 10$ s, and thus, $r = 1/R = 0.1$. For c_0, we measured the duration of writing all regular backups and divided the sum by the total number of regular backups and workers. For c_1, similarly, we measured the duration of the bookkeeping and IMap accesses during all steals and frame returns on the victim and thief sides, summed them up, and divided the sum by the number of steals and workers. For c_2, we measured the duration of all failure handler executions at each worker, summed them up, and divided the sum by the number of failures and the average number of workers alive.

While the above constants were determined by injecting single worker failures, we tested the accuracy of our formula by injecting multiple failures. Thus, we compared the predictions of the formula (including the constants) with measured running times of SYN executions on Lichtenberg, into which we injected up to 32 worker failures at random times. Figure 7 shows the predicted and measured running times with 192 workers and the same parameter values as in "Experiments". The measured running times are averages over 25 runs. We observe that the predictions

Table 3 Constant values averaged over 25 runs

Constant	j	r	c_0	c_1	c_2
Value	0.47	0.1	5 ms	68 ms	521 ms

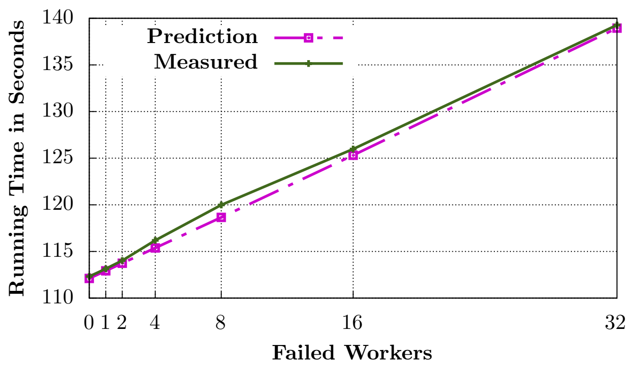


Fig. 7 Predicted and measured running times for up to 32 failures out of 192 workers on Lichtenberg

are close to the measurements, with a maximum difference of 1.11 % percent for 8 worker failures.

Related Work

There is a large body of work on AMTs (surveyed in [9, 14, 49]) and on AMT scheduling (e.g., [5, 25, 52]). The lifeline scheme was proposed in [41], where it referred to single-threaded processes. Later, it was extended to a hybrid scheme, in which the workers of each process balance their load via work sharing [8]. Reitz and Fohry [39] compared the hybrid scheme with the simpler lifeline-pure scheme introduced in that reference, and observed that the lifeline-pure scheme usually performs better. Its efficiency can be further improved through locality- and load-aware victim selection [39].

Outside the AMT area, fault-tolerance tools for parallel programs include BLCR [15] and DMTCP [1] for system-level C/R, and SCR [29] and FTI [3] for application-level checkpointing [45]. In addition to application-level checkpointing, other application-level techniques are Algorithm-Based Fault Tolerance (ABFT) and naturally fault-tolerant algorithms [44]. All of these techniques protect programs against permanent node failures. Other resilience techniques handle silent data corruptions (SDC), such as bit flips. A prominent technique to protect programs against SDC is replication [46].

Much of the research on AMT resilience refers to SDC, e.g., [13, 22, 31]. AMT research to handle permanent node failures can be classified according to the use of checkpointing or other techniques. Additionally, it refers to different task models. Most of this research refers to the DIT model. In addition to AllFT, other schemes have been studied, which, e.g., use incremental checkpointing [35], do not rely on a resilient store [11], or combine fault tolerance with elasticity [34]. Incremental checkpointing could be applied to our NFJ scheme to further reduce the protection overhead. For the hybrid lifeline scheme mentioned above, a fault-tolerance approach has been sketched in reference [37]. Another recent checkpointing technique for StarPU concentrates on checkpointing the data that are communicated between tasks [26]; whereas the tasks themselves are known from the beginning and can be easily re-run. Somewhat related, Ma and Krishnamoorthy [28] consider tasks with side effects. They log memory accesses of tasks and use this information to restore data and to identify tasks to be re-executed after failure.

A resilience technique for NFJ programs, which is not based on checkpointing, leverages the natural task duplication during work stealing. Victims re-initiate execution of their stolen tasks in case the thief dies, which is sometimes called supervision [20]. Supervision also works for DIT, where it was compared to TC in [36]. In this reference, supervision was observed to have less overhead than TC in failure-free runs, and TC to have lower recovery costs. The fault tolerance overheads for SYN with up to 144 workers were less than 1 % for both methods in failure-free runs.

Localized recovery has been previously deployed for both AMT (e.g., [20, 21, 26]) and other parallel programs (e.g., [12]). It must be supported by a programming environment such as User-Level Failure Mitigation (ULFM [27]). An alternative to shrinking recovery is the usage of spare processes [27].

Theoretical studies on fault tolerance are common. For instance, models have been studied for determining the optimal checkpointing interval and for comparing fault-tolerance approaches [4, 7, 47]. We based our running time estimation on a prior one for the AllFT scheme [36].

Conclusions

This paper has shown that TC can protect NFJ programs against permanent node failures. We presented the first TC scheme for NFJ, which is also the first TC scheme for multi-worker processes under work stealing.

We evaluated the scheme in experiments with four benchmarks and up to 1536 workers, and observed protection overheads of up to 43.98 % and negligible recovery costs. The protection overheads are higher than those of TC for DIT, but lower than typical C/R overheads. The higher costs than in DIT are mostly due to the need for an additional frame return protocol. Nevertheless, we expect that they can be reduced in future work, e.g., through low-level optimizations, and by combining backups from different protocols into one.

Further, we proposed a formula for predicting running times in case of multiple worker failures and experimentally validated it. We observed an error of up to 1.11 % compared to measured running times for up to 32 worker failures.

With our work, we were able to transfer a previous TC scheme from DIT to NFJ. The success of this transfer is promising, since it suggests that future work may possibly be able to generalize TC to further task models. Additionally, future work should consider incremental checkpointing and more complicated benchmarks.

Acknowledgements The authors gratefully acknowledge the computing time provided to them on the Goethe-HLR cluster at the Frankfurt Center for Scientific Computing and on the high-performance computer Lichtenberg at the NHR Centers NHR4CES at TU Darmstadt. The Lichtenberg cluster is funded by the Federal Ministry of Education and Research, and the state governments participating on the basis of the resolutions of the GWK for national high-performance computing at universities.

Author Contributions All listed authors have contributed to this research.

Funding Open Access funding enabled and organized by Projekt DEAL. This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Project no. 512078735.

Data availability statement The source code of this paper is available online: https://zenodo.org/doi/10.5281/zenodo.10055194.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Research involving human and/or animals This research does not contain any studies with human participants or animals performed by any of the authors.

Informed consent Informed consent was obtained from all individual participants included in this research.

References

1. Ansel J, Arya K, Cooperman G. DMTCP: transparent checkpointing for cluster computations and the desktop. In: Proceedings international parallel and distributed processing symposium (IPDPS). IEEE. 2009. pp. 1–12. https://doi.org/10.1109/ipdps.2009.5161063.
2. Augonnet C, Thibault S, Namyst R, et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr Comput Pract Exp. 2011;23:187–98. https://doi.org/10.1002/cpe.1631.
3. Bautista-Gomez L, Tsuboi S, Komatitsch D, et al. FTI: High performance fault tolerance interface for hybrid systems. In: Proceedings international conference for high performance computing, networking, storage and analysis (SC). ACM. 2011. pp. 1–32. https://doi.org/10.1145/2063384.2063427.
4. Benoit A, Herault T, Fèvre VL, et al. Replication is more efficient than you think. In: Proceedings international conference for high performance computing, networking, storage and analysis (SC). ACM. 2019. pp. 1–14. https://doi.org/10.1145/3295500.3356171.
5. Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. J ACM. 1999;46(5):720–48. https://doi.org/10.1145/324133.324234.
6. Chamberlain BL, Callahan D, Zima HP. Parallel programmability and the Chapel language. Int J High Perform Comput Appl. 2007;21(3):91–312. https://doi.org/10.1177/1094342007078442.
7. Daly JT. A higher order estimate of the optimum checkpoint interval for restart dumps. Future Gener Comput Syst. 2006;22(3):303–12. https://doi.org/10.1016/j.future.2004.11.016.
8. Finnerty P, Kamada T, Ohta C. Self-adjusting task granularity for global load balancer library on clusters of many-core processors. In: Proceedings international workshop on programming models and applications for multicores and manycores (PMAM). ACM. 2020. pp. 1–10. https://doi.org/10.1145/3380536.3380539.
9. Fohry C. An overview of task-based parallel programming models. In: Tutorial at European network on high-performance embedded architecture and compilation conference (HiPEAC). 2020. https://doi.org/10.5281/zenodo.8425959.
10. Fohry C. Checkpointing and localized recovery for nested fork-join programs. In: International symposium on checkpointing for supercomputing (SuperCheck). 2021. arXiv:2102.12941.
11. Fohry C, Bungart M, Plock P. Fault tolerance for lifeline-based global load balancing. J Softw Eng Appl. 2017;10(13):925–58. https://doi.org/10.4236/jsea.2017.1013053.
12. Gamell M, Teranishi K, Heroux MA, et al. Local recovery and failure masking for stencil-based applications at extreme scales. In: Proceedings international conference for high performance computing, networking, storage and analysis (SC). 2015. pp. 70:1–70:12. https://doi.org/10.1145/2807591.2807672.
13. Gupta N, Mayo JR, Lemoine AS, et al. Towards distributed software resilience in asynchronous many-task programming models. In: Workshop on fault tolerance for HPC at eXtreme Scale (FTXS). 2020. pp. 11–20. https://doi.org/10.1109/FTXS51974.2020.00007.
14. Gurhem J, Petiton SG. A current task-based programming paradigms analysis. In: Proceedings computational science (ICCS). Springer; 2020. pp. 203–16. https://doi.org/10.1007/978-3-030-50426-7_16.
15. Hargrove PH, Duell JC. Berkeley lab checkpoint/restart (BLCR) for linux clusters. J Phys Conf Ser. 2006;46:494–9. https://doi.org/10.1088/1742-6596/46/1/067.
16. Hazelcast. The leading open source in-memory data grid. 2023. http://hazelcast.org.
17. Herault T, Robert Y. Fault-tolerance techniques for high-performance computing. Berlin: Springer; 2015. https://doi.org/10.1007/978-3-319-20943-2.
18. IBM. The APGAS library for fault-tolerant distributed programming in Java 8. 2023. https://github.com/x10-lang/apgas.
19. Kaiser H, Heller T, Adelstein-Lelbach B, et al. HPX: a task based programming model in a global address space. In: Proceedings international conference on partitioned global address space programming models (PGAS). ACM. 2014. pp. 1–11. https://doi.org/10.1145/2676870.2676883.

20. Kestor G, Krishnamoorthy S, Ma W. Localized fault recovery for nested fork-join programs. In: Proceedings international symposium on parallel and distributed processing (IPDPS). IEEE. 2017. pp. 397–408. https://doi.org/10.1109/ipdps.2017.75.
21. Kolla H, Mayo JR, Teranishi K, et al. Improving scalability of silent-error resilience for message-passing solvers via local recovery and asynchrony. In: Proceedings Workshop on fault tolerance for HPC at eXtreme Scale (FTXS). 2020. pp. 1–10. https://doi.org/10.1109/FTXS51974.2020.00006.
22. Kurt MC, Krishnamoorthy S, Agrawal K, et al. Fault-tolerant dynamic task graph scheduling. In: Proceedings international conference for high performance computing, networking, storage and analysis (SC). ACM. 2014. pp. 719–30. https://doi.org/10.1109/SC.2014.64
23. Laboratory ORN. Frontier. 2023. https://www.olcf.ornl.gov/frontier.
24. Lea D. A Java fork/join framework. In: Proceedings of the conference on java grande. ACM. 2000. pp. 36–43. https://doi.org/10.1145/337449.337465.
25. Lifflander J, Slattengren NL, Pébaÿ PP, et al. Optimizing distributed load balancing for workloads with time-varying imbalance. In: Proceedings IEEE international conference on cluster computing (CLUSTER). 2021. pp. 238–48. https://doi.org/10.1109/Cluster48925.2021.00039.
26. Lion R, Thibault S. From tasks graphs to asynchronous distributed checkpointing with local restart. In: Proceedings international conference on high performance computing, networking, storage and analysis (SC) workshops (FTXS). ACM. 2020. pp. 31–40. https://doi.org/10.1109/FTXS51974.2020.00009.
27. Losada N, González P, Martìn MJ, et al. Fault tolerance of MPI applications in exascale systems: the ULFM solution. Future Gener Comput Syst. 2020;106:467–81. https://doi.org/10.1016/j.future.2020.01.026.
28. Ma W, Krishnamoorthy S. Data-driven fault tolerance for work stealing computations. In: Proceedings international conference on supercomputing (ICS). ACM. 2012. pp. 79–90. https://doi.org/10.1145/2304576.2304589.
29. Moody A, Bronevetsky G, Mohror K, et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings international conference for high performance computing, networking, storage and analysis (SC). ACM. 2010. pp. 1–11. https://doi.org/10.1109/SC.2010.18.
30. OpenMP Architecture Review Board. OpenMP application programming interface (version 5.2). 2021. https://www.openmp.org.
31. Paul SR, Hayashi A, Slattengren N, et al. Enabling resilience in asynchronous many-task programming models. In: Proceedings Euro-par: parallel processing. Springer. pp. 346–60. https://doi.org/10.1007/978-3-030-29400-7_25.
32. Posner J. System-level vs. application-level checkpointing. In: Proceedings international conference on cluster computing (CLUSTER), extended abstract. IEEE. 2020. pp. 404–5. https://doi.org/10.1109/CLUSTER49012.2020.00051.
33. Posner J, Fohry C. A Java task pool framework providing fault-tolerant global load balancing. Special Issue Int J Netw Comput. 2018;8(1):2–31. https://doi.org/10.15803/ijnc.8.1_2.
34. Posner J, Fohry C. Transparent resource elasticity for task-based cluster environments with work stealing. In: Proceedings international conference on parallel processing (ICPP) workshops (P2S2). ACM. 2021. https://doi.org/10.1145/3458744.3473361.
35. Posner J, Reitz L, Fohry C. A comparison of application-level fault tolerance schemes for task pools. Future Gener Comput Syst. 2019;105:119–34. https://doi.org/10.1016/j.future.2019.11.031.
36. Posner J, Reitz L, Fohry C. Task-level resilience: checkpointing vs. supervision. Special Issue Int J Netw Comput. 2022;12(1):47–72. https://doi.org/10.15803/ijnc.12.1_47.
37. Reitz L. Task-level checkpointing for nested fork-join programs. In: Proceedings international parallel and distributed processing symposium (IPDPS), Ph.D. forum, extended abstract. IEEE. 2021. https://doi.org/10.1109/IPDPSW52791.2021.00160.
38. Reitz L. Implementations of our nested fork-join AMTs with- and without task-level checkpointing. 2023. https://zenodo.org/doi/10.5281/zenodo.10055194.
39. Reitz L, Fohry C. Lifeline-based load balancing schemes for asynchronous many-task runtimes in clusters. Special Issue J Parallel Comput. 2023. https://doi.org/10.1016/j.parco.2023.103020.
40. Reitz L, Fohry C. Task-level checkpointing for nested fork-join programs using work stealing. In: Workshop on asynchronous many-task systems for exascale (AMTE). Springer; 2023 **(to appear)**.
41. Saraswat VA, Kambadur P, Kodali S, et al. Lifeline-based global load balancing. In: Proceedings SIGPLAN symposium on principles and practice of parallel programming (PPoPP). ACM. 2011. pp. 201–11. https://doi.org/10.1145/1941553.1941582.
42. Schardl TB, Lee ITA. OpenCilk: A modular and extensible software infrastructure for fast task-parallel code. In: Proceedings of the 28th SIGPLAN annual symposium on principles and practice of parallel programming. ACM. 2023. pp. 189–203. https://doi.org/10.1145/3572848.3577509.
43. Schmaus F, Pfeiffer N, Schroder-Preikschat W, et al. Nowa: a wait-free continuation-stealing concurrency platform. In: International parallel and distributed processing symposium (IPDPS). 2021. pp. 360–371. https://doi.org/10.1109/IPDPS49936.2021.00044.
44. Semmoud A, Hakem M, Benmammar B. A survey of load balancing in distributed systems. Int J High Perform Comput Netw. 2019;15:233. https://doi.org/10.1504/IJHPCN.2019.106124.
45. Shahzad F, Wittmann M, Kreutzer M, et al. A survey of checkpoint/restart techniques on distributed memory systems. Parallel Process Lett. 2013;23(4):1340011–30. https://doi.org/10.1142/s0129626413400112.
46. Subasi O, Yalcin G, Zyulkyarov F, et al. Designing and modelling selective replication for fault-tolerant HPC applications. In: International symposium on cluster, cloud and grid computing (CCGRID). 2017. pp. 452–7. https://doi.org/10.1109/CCGRID.2017.40
47. Subasi O, Martsinkevich T, Zyulkyarov F, et al. Unified fault-tolerance framework for hybrid task-parallel message-passing applications. Int J High Perform Comput Appl. 2018;32(5):641–57. https://doi.org/10.1177/1094342016669416.
48. Tardieu O. The APGAS library: resilient parallel and distributed programming in Java 8. In: Proceedings SIGPLAN workshop on X10. ACM. 2015. pp. 25–26. https://doi.org/10.1145/2771774.2771780.
49. Thoman P, Dichev K, Heller T, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. J Supercomput. 2018;74(4):1422–34. https://doi.org/10.1007/s11227-018-2238-4.
50. TOP500.org. Goethe-HLR of the University of Frankfurt. 2023. https://www.top500.org/system/179588.
51. TOP500.org. Lichtenberg II (phase 1) of the Technical University of Darmstadt. 2023b. https://www.top500.org/system/179857.
52. Yang J, He Q. Scheduling parallel computations by work stealing: a survey. Int J Parallel Programm. 2018;46(2):173–97. https://doi.org/10.1145/324133.324234.
53. Zhang W, Tardieu O, Grove D, et al. GLB: Lifeline-based global load balancing library in X10. In: Proceedings workshop on parallel programming for analytics applications (PPAA). ACM. 2014. pp. 31–40. https://doi.org/10.1145/2567634.2567639.