Task-Level Checkpointing for Nested Fork-Join Programs using Work Stealing

AMTE – EuroPar

Lukas Reitz and Claudia Fohry
lukas.reitz@uni-kassel.de

UNI KASSEL VERSITÄT
University of Kassel, Germany
Research Group Programming Languages / Methodologies

28th August 2023

Motivation

- Problem: the increasing number of processing units in supercomputers leads to more frequent hardware failures
- Popular solutions include
  - Checkpoint/Restart
  - Application-level checkpointing
  - Algorithm-based fault tolerance, naturally fault tolerant algorithms, . . .
- We consider Asynchronous Many-Task (AMT) programs
  $\rightarrow$ **Task-level checkpointing (TC)**

AMT Programs

- Computation is divided into tasks which are processed by workers
- AMT programs differ widely in their *task models* (e.g., dynamic independent tasks, nested-fork join)
- We consider AMTs which deploy work stealing to balance the tasks between workers

Task-Level Checkpointing (TC)

- Operates in the AMT runtime system (usually transparent to the application programmer)
- Exploits the clearly defined interfaces of tasks
- Has only been studied for a few rather simple settings (e.g., dynamic independent tasks)

Contributions

- We propose a novel TC scheme for Nested Fork-Join (NFJ) programs running on clusters with multi-worker processes using work stealing
- We implement and evaluate the scheme in experiments with up to 1280 workers and find
  - a fault-tolerance overhead of up to 28.3 % and
  - negligible costs for recovery

Our Setting

- We refer to an existing TC scheme [1], called *AllFT*, which is designed for cluster AMTs supporting dynamic independent tasks that use single-worker processes
- We build our new TC scheme on AllFT and a recent cluster AMT for nested-fork join programs [2]

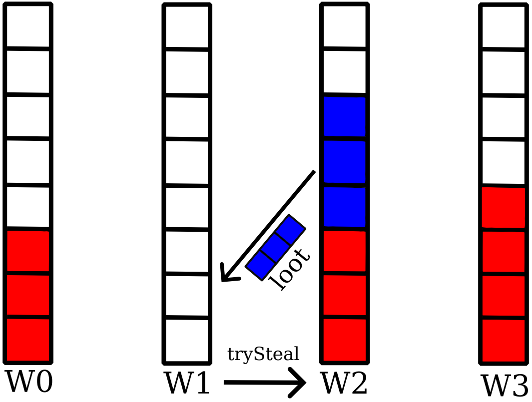
[1] Posner et al.: A Comparison of Application-level Fault Tolerance Schemes for Task Pools. Future Generation Computing Systems 105 (2019)
[2] Reitz et al.: Lifeline-based Load Balancing Schemes for Asynchronous Many-Task Runtimes in Clusters. Parallel Computing 116 (2023)

Lifeline Scheme

- The lifeline scheme[3] is a well-performing work stealing scheme
- Each worker maintains local task queue
- Workers are arranged in *lifeline graph*
- It was first implemented in X10
- Cooperative work stealing

[3] Saraswat et al., Lifeline-based Global Load Balancing, PPoPP, 2011

Cooperative Work Stealing



Example of W1 successfully stealing tasks from W2

Dynamic Independent Tasks vs Nested Fork-Join

Dynamic Independent Tasks (DIT) **Nested Fork-Join (NFJ)**

Each task may spawn child tasks

Parameter passing from parent to child

No side effects

Dynamic Independent Tasks (DIT)

- Each task yields task result
- Final result calculated by reduction
- Child-Stealing, Steal-Half policy
- Examples: GLB, Blaze-Tasks

Nested Fork-Join (NFJ)

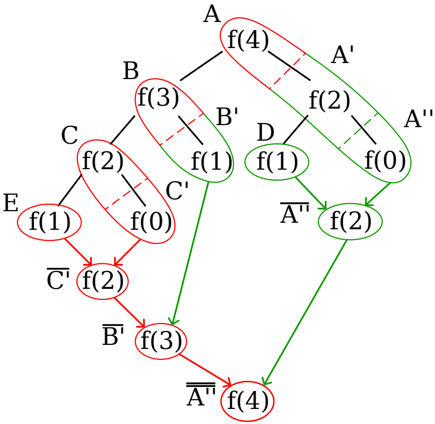
- Each task returns to parent task
- Root task yields final result
- Continuation-Stealing, Steal-1 policy
- Examples: Cilk, Satin

Example NFJ application

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

Example NFJ application

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



The AllFT scheme

AllFT encompasses:

- a checkpointing procedure which saves a worker state to a resilient store,
- a steal protocol which ensures consistency between a victim, a thief, and their respective checkpoints,
- a restore protocol which ensures that the tasks from failed workers are taken over by alive workers, and
- a selection scheme for buddy workers which are responsible for the restore of failed workers

Our TC scheme

Major changes from AllFT:

- Checkpoints include the state of an **NFJ** worker:
  - current local pool contents
  - next task (which is not always contained in the local pool)
  - task results that are yet to be incorporated into their parent frame
  - frames that are awaiting result incorporation
  - some bookkeeping information

Our TC scheme (cont.)

Major changes from AllFT:

- Checkpoints are written either at a spawn or at the end of a function
- A new frame return protocol keeps checkpoints consistent during result incorporation
- The restore protocol additionally adopts task results (in contrast to worker results in AllFT) and frames
- Buddy worker selection operates on workers instead of processes

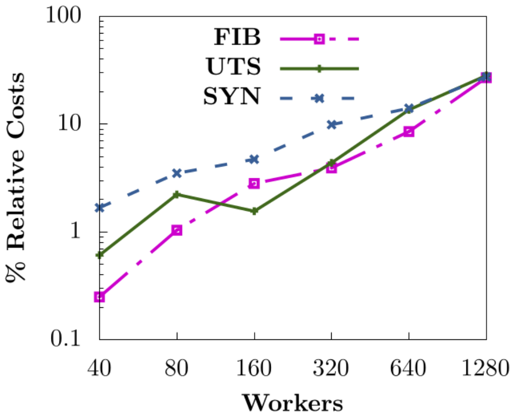
Implementation

- Our implementation is based on the APGAS for Java library
- We use Hazelcast's IMap for the resilient store
- The IMap saves the checkpoints as key-value pairs, groups them into partitions, and evenly distributes the partitions over nodes
- Checkpoints from all workers of the same process are mapped to the same partition
- Up to six simultaneous node failures can be tolerated and program abort occurs with an error message for more failures

Experimental Setting

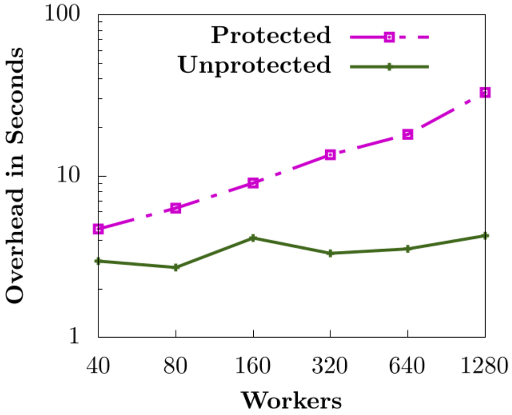
- Benchmarks:
  - naive recursive Fibonacci (FIB)
  - Unbalanced Tree Search (UTS)
  - Synthetic Benchmark (SYN)
- Goethe cluster of the University of Frankfurt
- Up to 32 nodes, totaling in 1280 workers

Results: Protection Costs



Protection costs for FIB, UTS, and SYN in failure-free execution averaged over 10 runs each

Results: Overhead of SYN



Load balancing and protection overhead of SYN with and without protection in failure-free execution averaged over 10 runs each

Results: Estimation of Recovery Costs

Average running times of UTS with executions A, B, and C in seconds

Execution	Workers	Running Time
A	640	245.23 s
B	600	277.38 s
C	640 – 80 at half the running time	291.28 s

- Executions B and C use the same average number of processing workers
- Estimation of restore overhead as the difference between running times of C and B
- Restore overhead for a crash of 80 out of 640 workers is about 5 % of the running time

Conclusions

- TC can protect NFJ programs against permanent hardware failures
- Fault-tolerance overhead in failure-free execution is lower than typical Checkpoint/Restart
- Negligible costs for recovery of single worker failures
- Future work includes the evaluation of TC in more complex benchmarks and the generalization to further task models