Localized Fault Recovery for Nested Fork-Join Programs

Gokcen Kestor (Oak Ridge National Laboratory)

Sriram Krishnamoorthy (Pacific Northwest National Laboratory), Wenjing Ma (Chinese Academy of Sciences)

Introduction

- High performance computers are increasingly susceptible to errors
- Periodic checkpointing is widely used approach to fault tolerance, but
  - recovery cost can be proportional to system size
  - it introduces large performance overhead

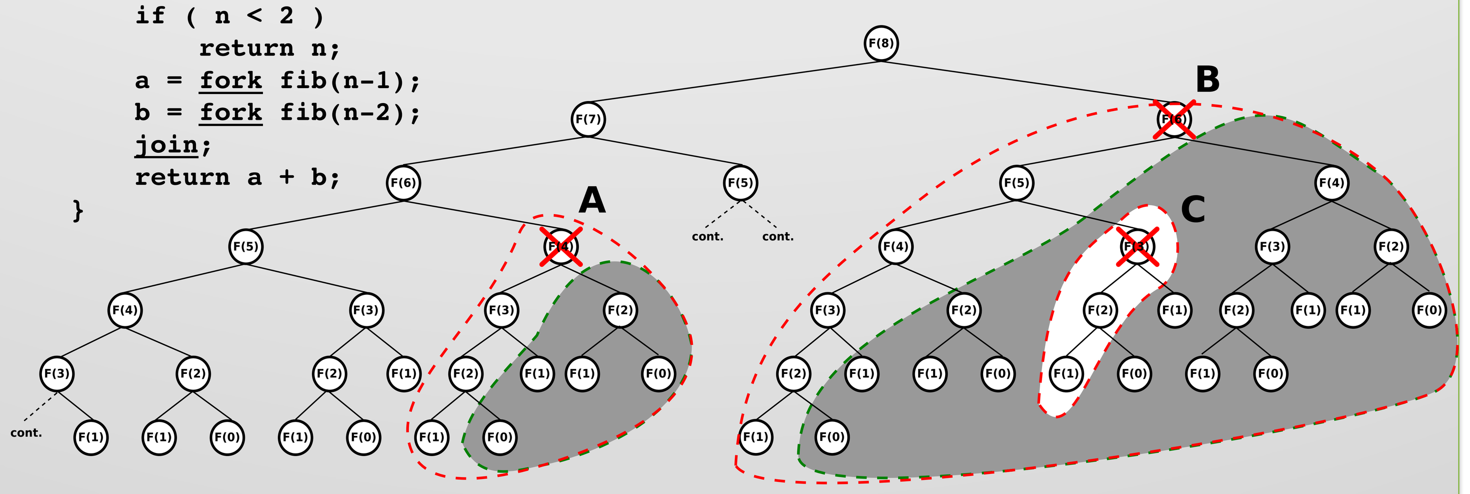
- We consider the design of fault tolerance mechanisms in the presence of fail-stop failures for
  - nested fork-join programs,
  - executed on distributed memory machines,
  - load balancing provided by work stealing

Nested fork-join models provide an opportunity to perform localized fault recovery

Problem Statement and Objectives

- Reducing the amount of re-executed work in the presence of failures
- Guaranteeing forward progress even during fault recovery
- Ensuring correct interleaving of remote operations and error notifications
- Efficiently handling nested recovery, concurrent recovery, and failure-during-recovery scenarios

```
fib(n)
{
    if ( n < 2 )
        return n;
    a = fork fib(n-1);
    b = fork fib(n-2);
    join;
    return a + b;
}
```



Our Proposal: ForkJoinFT

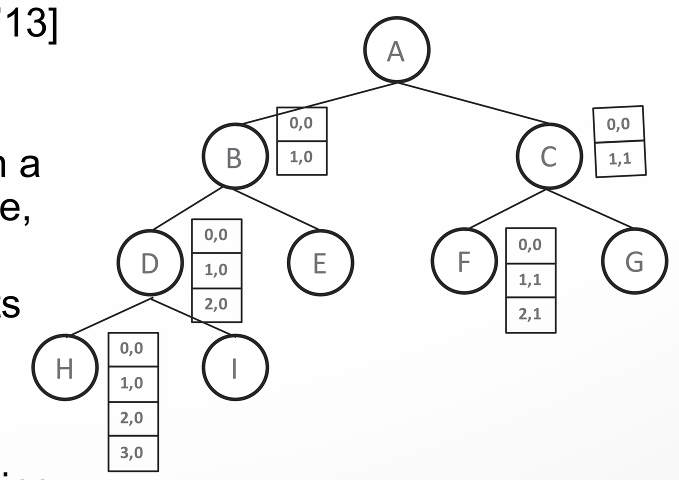
- A modified distributed-memory algorithm that incorporates efficient fault recovery

- ForkJoinFT executes ***all*** and ***only*** lost work due to a fault, it needs to:
  1. track the relationship between the subcomputations performed by different threads
  2. reconstruct the relationship among live processes that have pending interactions with the failed node
  3. re-execute all and only lost subcomputations without interfering with the normal task execution

Gokcen Kestor, Sriram Krishnamoorthy, Wenjing Ma, "Localized Fault Recovery for Nested Fork-Join Programs", *IEEE International Parallel and Distributed Processing Symposium IPDPS 2017*, pp. 397-408, May 2017, Orlando (FL).

Tracking Global Computation

- We extended steal tree algorithm [PLDI'13] to retain only the *live* portion of subcomputations:
  - each steal operation is identified with a unique ID (victim rank, working phase, level, and step)
  - at every steal operation, the thief gets its victim steal path and adds the current steal operation
  - all the preceded steals (*Stolen Step*) from a given victim in the same working phase are recorded



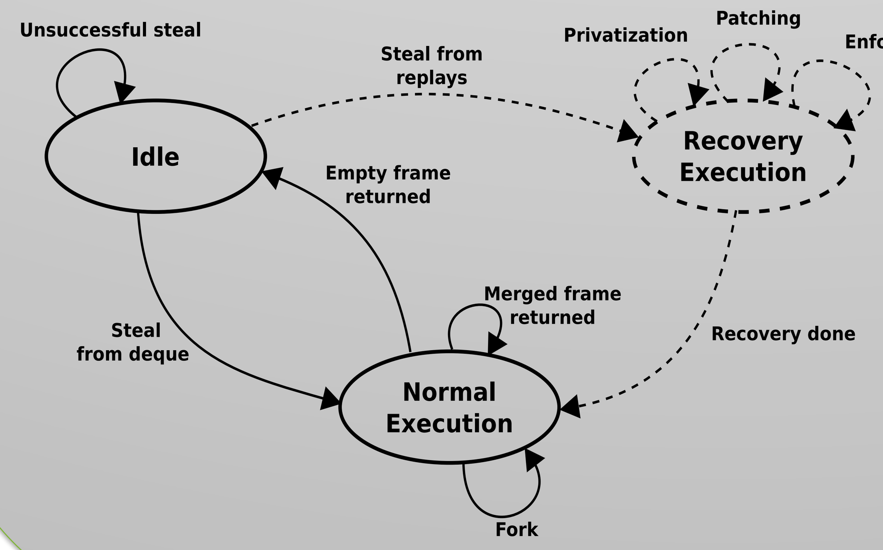
Recovering Global Computation

- Failure notifications are assumed to be sent to the server threads
- Upon a failure notification, each server thread independently initiates recovery:
  - Identifies pending subcomputations stolen by the failed worker
  - marks the victim of the failed worker as a recovery node
  - requests steal tree paths that include the failed worker from all workers
  - collects all steal tree paths and construct a replay tree
    - the root of the replay tree is the subcomputation stolen by the failed worker
    - collection is a distributed binary-tree-based reduction
  - makes the replay tree and its root task ready to be stolen

ForkJoinFT re-executes only lost subcomputations

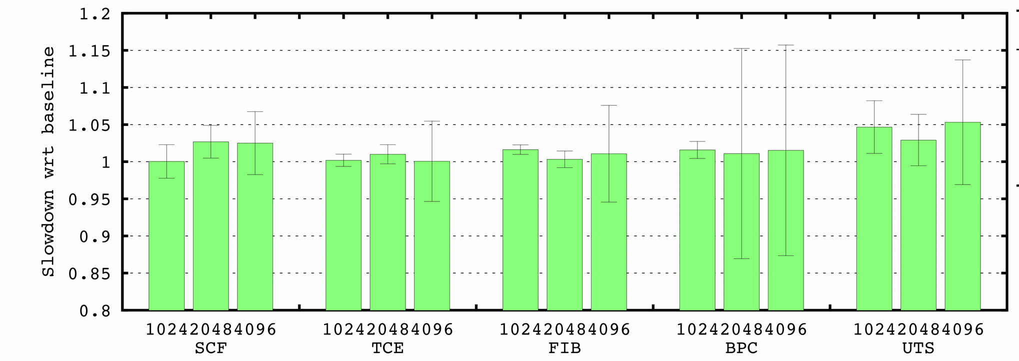
Scheduling Re-Execution

- When a thief steals work to be re-executed:
  - its victim determines the task's ***frontier***
  - task's frontier is the failed worker's list of **alive children**
  - the thief assumes **ownership** of the root task of replay tree
    - thieves of this subcomputation will return their results to new owner, rather than the failed worker



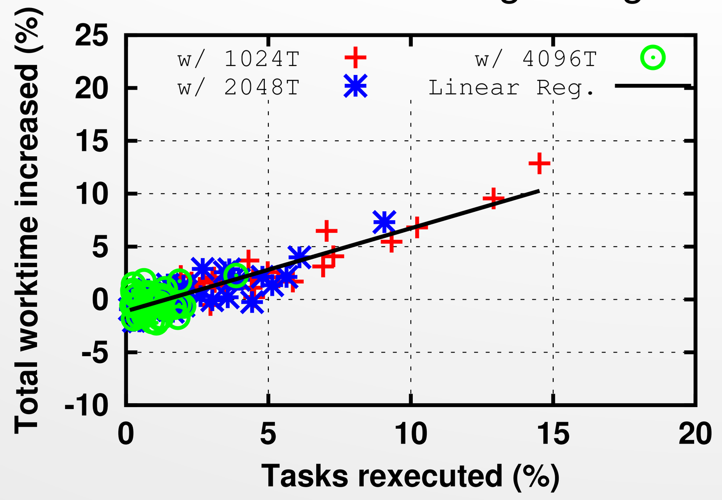
- **Privatization:** spawned task is not stolen in the failed execution, will be executed by the current worker
- **Enforced steal:** spawned task is stolen in the failed execution, will be donated to other thieves
- **Patching:** spawned task already exists in another live worker, no need to be executed

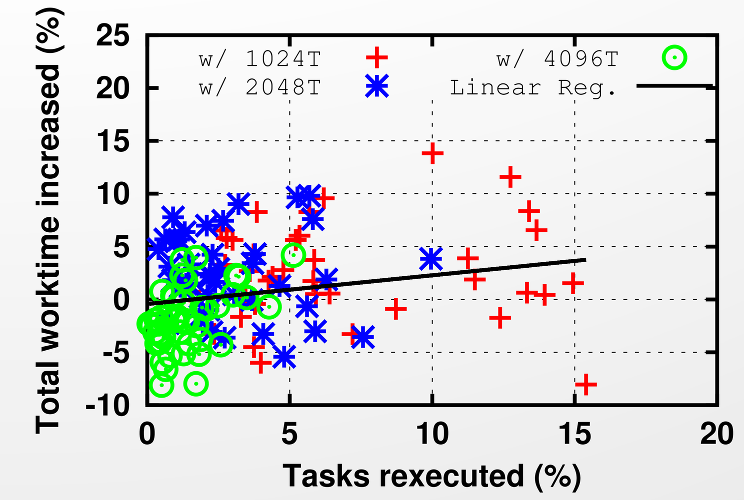
Results



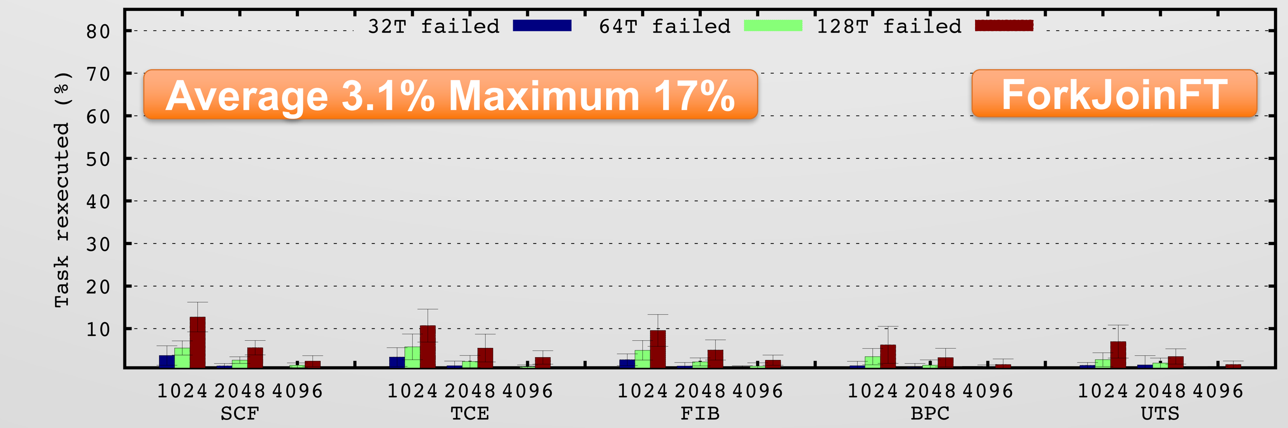
	32 nodes	64 nodes	128 nodes
SCF	2.79	2.48	2.31
TCE	2.68	2.61	2.43
FIB	0.92	0.88	0.85
BPC	42.45	43.12	43.24
UTS	2.54	2.48	2.40

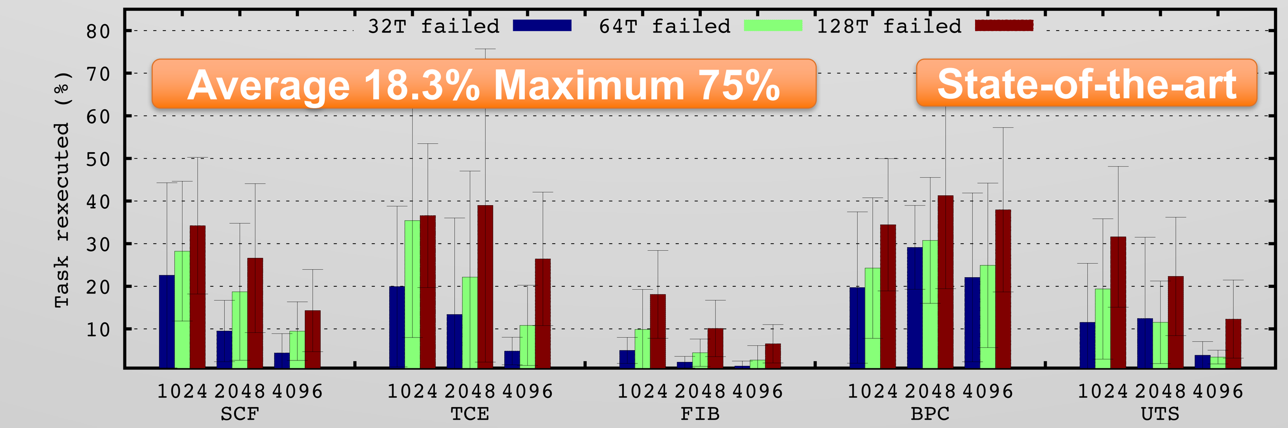
- Negligible overhead and does not increase with core count
- Space overhead per thread is generally a few KB and remains roughly constant when scaling to larger node counts





- The increase of total work time is generally less than 15%
- A regression analysis (OLS) models the relation between the number of re-executed tasks and the increase in work time reveals (sub) linear relationships





Conclusions

- We presented an approach to localized fault recovery specific to nested fork-joined programs executed on distributed-memory systems
- Our fault tolerance approach:
  - introduces negligible overhead of in the absence of faults, within the execution time variation
  - re-executes all and only lost work due to faults
  - significantly decreases the amount of work re-executed as compared to alternative strategies
  - presents a recovery overhead roughly proportional to the amount of lost work