Task-Level Checkpointing for Nested Fork-Join Programs

Lukas Reitz
Advisor: Prof. Dr. Claudia Fohry
University of Kassel, PLM, Germany
{lukas.reitz | fohry}@uni-kassel.de

Abstract— Fault tolerance is often realized through checkpointing and, upon failure, continuing the application from a previously saved checkpoint. Checkpointing on task-level can be efficient, because only task descriptors have to be saved.

Nested fork-join programs are an important type of task-based parallel programs, where tasks may spawn multiple child tasks and wait for their results. Recently, a task-level checkpointing scheme for nested fork-join programs has been proposed. However, it has originally been designed for programs that restrict the number of worker threads per process to one.

This work extends the scheme to multiple workers threads per process. We describe the required algorithmic changes, including a backup writing scheme that saves valid checkpoints efficiently.

Keywords— Fault Tolerance, Resilience, Work Stealing, Task-based Parallel Programming

1 Introduction

Checkpointing is an established technique for tolerating fail-stop failures in distributed applications [1]. Recently, task-level checkpointing has received some attention [2, 3]. It achieves efficiency through saving only task descriptors to a resilient store.

We consider the nested fork-join (NFJ) task model, where tasks are usually represented by stack frames. Tasks may spawn several child tasks, wait for their children's results, and return a result to their parent. The NFJ model is suitable for, e.g., divide-and-conquer algorithms. NFJ programs are often implemented with work stealing, where processes, called thieves, steal tasks from other processes, called victims. An example of an NFJ programming environment is Cilk [4].

Recently, a task-level checkpointing scheme for the NFJ model has been sketched [5]. It saves checkpoints at regular time intervals, in the event of work stealing, and during recovery. Upon failure, it recovers locally, confining the failure handling to a small subset of the processes. Moreover, it deploys shrinking recovery, where the program execution continues on a reduced set of processes. The original scheme has been designed for programs that restrict the number of worker threads per process to one. On computing nodes equipped with several processing units, however, it is often beneficial to run a small number of processes with multiple worker threads each.

This paper extends the original scheme to support such cases. We assume that each worker thread maintains an own task queue. Moreover, the workers of a process use *shared data*, for instance intermediate results. Checkpoints are saved per process instead of per worker, as this considerably reduces the overall number of checkpoints. The original scheme defines the contents of checkpoints for single worker processes. We modify their definition to include the task queues of all workers as well as the shared data.

The process of constructing a valid checkpoint and writing it to a resilient store, henceforth called *backup writing*, requires consideration of all work stealing-related operations being performed during this process. In the remainder of this paper, we propose a way to realize backup writing without stalling the workers. For that, we copy the task queues independently and delay some operations to achieve valid checkpoints.

2 Checkpointing Algorithm

We refer to the following work stealing scheme, which resembles the one by Finnerty et al. [6]: Each worker repeatedly processes up to n tasks from its queue, and then answers steal requests, i.e., each worker alternates between task processing and communication phases. When all task queues of a process are empty, the last worker whose queue became empty steals work from another process.

Workers proceed in a work-first manner: When a task spawns another task, the worker branches into the child task and puts the continuation of the parent task into its queue. When a previously stolen task needs the result of a remote child, a so-called *frame return* operation is executed. Therein, the parent frame is sent

back to the victim, where the child result is filled in. If it has not been computed yet, the frame is saved in the victim's shared data instead, and the result is filled in later. After filling in the result, the frame is queued for execution on the victim. Besides the frames, the shared data area contains local results awaiting their frame and other data (see Reference [5]).

A naive approach to backup writing could synchronize and pause all workers of a process from the beginning until the end of backup writing. We devised the following more efficient alternative, in which the workers first construct a checkpoint via independent contributions, and then one of them writes it to a resilient store. This way, workers need not pause.

Backup writing of a process is initiated by an arbitrary worker, e.g., when the time interval for the backup writing is over. Each worker checks in its communication phase, whether a backup writing has begun, and if so, it copies its task queue. The last worker additionally writes the task queue copies and the shared data to a resilient store.

Recall that tasks are represented by stack frames and that a frame may either be located in a task queue or in the shared data area. One operation that may move a frame from the shared data area into a task queue is result return, where a task result is filled into the frame.

Regarding the checkpoint, this movement can lead to frame loss: If, for example, right after a worker saved its task queue, it processes a frame that returns a result to its parent frame, the parent frame may be moved from the shared data area to this worker's task queue. Then, the frame is not contained in the checkpoint, as neither the previously saved task queues nor the shared data contain it. Similarly, operations that move a frame from a task queue to the shared data area may cause the constructed checkpoint to contain the frame twice.

Therefore, during the whole process of backup writing, no operations that move frames between different parts of the shared state may be executed. To allow workers to continue task processing when they encounter such operations, we log them and execute them after backup writing. No waiting occurs as long as each worker has enough tasks in its queue to process during backup writing.

In the original checkpointing scheme, both the victim and the thief save checkpoints in the event of work stealing. The victim saves a checkpoint before sending the tasks, and the thief saves a checkpoint after receiving the tasks. If the victim writes a backup in the way described above, the sending of the tasks will be delayed until all workers have reached their communication phases and copied their task queues. This delay causes the thief to remain idle for a longer period of time than in the original scheme. To counter this, we adopt incremental steal backups [7], which save the number of tasks that were stolen since the last saved checkpoint, instead of the task queue contents. This way, the victim does not need to write a full backup, but instead it just has to update this number in the latest checkpoint, which is just an integer, before sending the tasks to the thief. Therefore, the delay is avoided, because the number can be updated without the participation of all workers.

3 Conclusions

This work extended the NFJ checkpointing scheme [5] to support multiple worker threads per process. We defined the checkpoint contents and sketched how checkpoints can be constructed without stalling the workers' task processing. We expect the extended scheme to perform similar to the original one. Future work should implement and experimentally evaluate it.

References

[1] T. Herault and Y. Robert, Eds., *Fault-Tolerance Techniques for High-Performance Computing.* Springer, 2015.

[2] R. Lion and S. Thibault, "From tasks graphs to asynchronous distributed checkpointing with local restart," in *FTXS*, 2020.

[3] J. Posner, L. Reitz, and C. Fohry, "A comparison of application-level fault tolerance schemes for task pools," *FGCS*, 2020.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *PLDI*, 1998.

[5] C. Fohry, "Checkpointing and localized recovery for nested fork-join programs," in *SuperCheck*, 2021.

[6] P. Finnerty, T. Kamada, and C. Ohta, "Self-adjusting task granularity for global load balancer library on clusters of many-core processors," in *PMAM*, 2020.

[7] C. Fohry, J. Posner, and L. Reitz, "A selective and incremental backup scheme for task pools," in *HPCS*, 2018.