2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

NanoCheckpoints: A Task-based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart

Omer Subasi, Javier Arias, Osman Unsal, Jesus Labarta, Adrian Cristal
Computer Architecture
Polytechnic University of Catalonia, Barcelona Supercomputing Center
Barcelona, Spain
name.surname@bsc.es

***Abstract*—In this paper, we present NanoCheckpoints which is a lightweight software-based checkpoint/restart scheme for task-parallel HPC applications. We leverage OmpSs, a task-based OpenMP derivative programming model (PM) and its Nanos asynchronous dataflow runtime. NanoCheckpoints achieves minimal overheads by checkpointing only tasks' inputs which are available for free in the OmpSs PM. We evaluate NanoCheckpoints by both pure task-parallel shared memory benchmarks (up to 16 cores) and hybrid OmpSs+MPI applications (up to 1024 cores). The results indicate that NanoCheckpoints has on average overhead 3% for shared memory benchmarks. The dataflow semantics of Nanos, where both checkpointing and error recovery are asynchronous, allows NanoCheckpoints to scale at large core counts even when high error rates are present. For hybrid OmpSs+MPI benchmarks, NanoCheckpoints has very low overhead, on average 2%, and high scalability.**

***Keywords*-Checkpoint/restart, Task parallelism, Dataflow**

I. Introduction

As technology scaling gets closer to the limits of Moore's Law and voltage scaling, various reliability related hurdles will appear which could impact High Performance Computing (HPC). In fact, reliability has joined performance and power dissipation as a first-class system design constraint not only for HPC but for all market segments from embedded to datacenter [1]. As we get closer to the exascale timeframe we might face a reliability-wall [2], with sub 1-hour Mean Time To Failure which will hinder further advances in HPC. Moreover, it is widely believed that hardware-based fault mitigation schemes will not suffice by themselves to keep error rates below design targets. Consequently this has increased the importance of software based resilience solutions at the operating system (OS), compiler, runtime, application and algorithm levels. However, to be feasible these solutions must be efficient, scalable, flexible and easy to implement. *In this work, we show that a dataflow runtime coupled with a task-based programming model provides a suitable substrate to develop such solutions.* A dataflow runtime schedules tasks in dataflow order: a task is scheduled for execution as soon as all of its inputs are available. Preferably, a dataflow-friendly task-based programming model(PM) like OmpSs [3] would accompany such a runtime. In OmpSs, the programmer only needs to specify the inputs and outputs of a task. Its runtime Nanos [4], then automatically infers task dependencies and executes tasks in dataflow fashion.

OmpSs PM and Nanos runtime offer several advantages for reliability. In OmpSs and Nanos, programmer specifies the task inputs and outputs. As a result of this, the exact data that needs to be checkpointed, i.e. task inputs and inouts, comes for free. In addition, Nanos tasks are executed asynchronously and in parallel which makes it inherently easy to exploit these properties for fault tolerance features. Crucially, failure recovery is very efficient: Nanos tasks that do not have dependencies to a task that has failed can continue to execute, in parallel and asynchronously with the recovery of the failed task. In contrast, in coordinated or synchronous checkpointing the fault recovery process acts as a barrier both in the physical sense (even non-faulty tasks block waiting for the resolution of the fault recovery) and in the scalability sense.

In this work, we introduce NanoCheckpoints framework that leverages the above-mentioned advantages of OmpSs PM and its Nanos asynchronous dataflow runtime. NanoCheckpoints is at the runtime level, and does not require any modifications at all to application code or OS. *Checkpointing and error recovery are task-local, asynchronous, automatic and completely transparent to the user.* We use the task directionality information to develop an efficient incremental checkpoint/restart scheme that only checkpoints minimal data to recover from errors. Our results indicate that the design is scalable even for high error rates. It has 3% performance overhead for shared memory OmpSs benchmarks. Moreover we evaluate NanoCheckpoints with hybrid OmpSs+MPI applications to demonstrate its very low overhead, on average 2%, and high scalability (with 1024 cores) for larger scales. Our main contribution is the development of NanoCheckpoints, a task-local fault tolerance framework based on a dataflow runtime coupled with a task-based PM. To the best of our knowledge, this is the first work that explores the fault-tolerance advantages of a dataflow runtime in detail. In summary, our contributions are:

- A low-cost and scalable checkpoint/restart mechanism, NanoCheckpoints, with 3% performance overhead on average for shared memory benchmarks.
- Evaluation of NanoCheckpoints for hybrid OmpSs+MPI benchmarks with 2% overhead and high scalability.

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.17

II. Background and Motivation

A. Fault Types, Fault Model and Related Work

We refer to failures or errors as the manifestation of faults. Faults can be categorized as transient, intermittent and permanent. Transient faults cause a memory location to return incorrect data temporarily until the location is overwritten with correct data. Intermittent faults cause a memory location to return incorrect values from time to time. Permanent faults are consistent and due to permanent damage in a memory location such as stuck-at faults. We target all these types of faults in our fault model. Furthermore, faults can be a single bit flip or multi-bit flips in nature. Single bit faults can be corrected by state-of-art SECDED ECCs. However multi-bit faults cannot be corrected and lead to errors. These errors are expected to become more frequent in the future with the increasing likelihood of double-bit and multi-bit flips [5] for caches and memory which are typically protected through SECDED ECCs. Thus, we propose efficient and scalable resilience support to also address multi-bit errors.

System-level checkpoint/restart mechanisms such as BLCR[6] incur large memory footprint since they are not aware of the minimal application state to be checkpointed. Moreover, they require OS modification. Solutions like Revive [7] are efficient but require custom hardware support. As a software-based solution, NanoCheckpoints optimizes memory usage and is OS/hardware independent. Application-level approaches, such as chkpt[8], are more efficient than system-level ones since they are data-aware: The programmer specifies which data to be checkpointed. However, they are not generic and have to be adapted for each single application. Finally, system-wide solutions for MPI applications such as [9] are orthogonal and integrable to NanoCheckpoints.

B. OmpSs, Nanos and Motivation for leveraging them

OmpSs [3] is a high-productivity, task-based programming model derived from OpenMP. It extends OpenMP with new clauses meant to pass knowledge from the high-level application domain into lower levels such as the runtime or the architecture. The clauses which are of particular interest for our work are those for declaring task inputs and outputs. In OmpSs the programmer has to explicitly declare the tasks' inputs and outputs. The task input and output declarations are used by the Nanos [4] runtime library to build the task dependency graph. Whenever a new task is created, the dependency graph is updated based on the task's inputs and outputs. The dependency graph is subsequently used for the correct task schedule and execution by obeying the task dependencies.

The OmpSs' asynchronous dataflow execution model has advantages over the synchronous models which are based on fork-join parallelism and utilized in most contemporary runtimes such as that of vanilla OpenMP. A prior work by Amer et al. [10] reports that asynchronous execution based on the dataflow model allows more parallelism and achieves more performance than synchronous execution based on the fork-join model. Likewise, we argue that execution of asynchronous tasks of OmpSs in a dataflow manner has significant

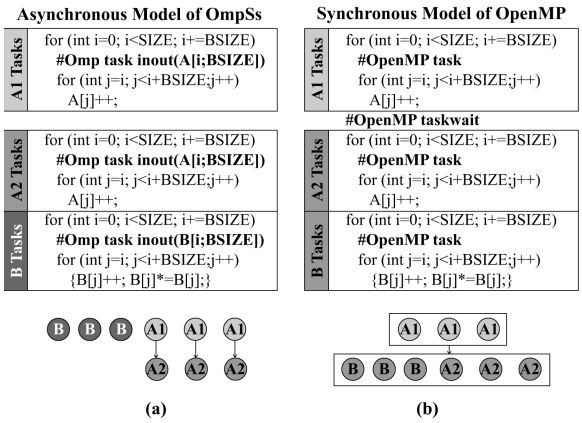

Fig. 1. Sample Code for Comparing OmpSs to OpenMP

advantages over that of the synchronous model which leads to blocking of irrelevant parts of execution on a fault occurrence. As an example in Figure 1, we have the OmpSs version on the left and the OpenMP version of the same program on the right part. In the first two loops, array A divided into chunks of BSIZE elements and each array element is incremented. In the last nested loop, array B elements are incremented and then squared. The OmpSs dataflow execution model would allow the execution of tasks which process parts of array B before finishing computation on A, since there is no data dependency between computations on A and B and the Nanos runtime is able to detect and act accordingly. On the other hand, for the OpenMP case on the right, one has to use barrier like taskwait constructs to ensure correctness since there is no automated inference of data dependencies. This prevents tasks processing parts of array B being executed before finishing those on array A when resources are available. Another advantage of OmpSs is the size of checkpoint: Information from dataflow enables the minimal checkpoint, i.e. task inputs and inouts which are provided by task annotations as seen in Figure 1 (a). In comparison, in the synchronous model, the checkpoint size is all of the application state if not all memory since its runtime does not have any information about the task states that need to be checkpointed as seen in Figure 1 (b).

III. NanoCheckpoints Design

NanoCheckpoints is a checkpoint/restart design. We checkpoint the task inputs when the task is ready to execute, which means that all its data dependencies are guaranteed to be satisfied by the runtime. This is necessary for the mechanism to be correct since if we checkpoint before the data dependencies are satisfied, we may checkpoint incorrect task input data. We checkpoint to main memory. When a task is about to finish, in case of an error, the runtime restores and restarts its execution, otherwise releases the dependencies and removes the checkpoint and related software data structures. We detect errors via capturing signals propagated by OS/hardware at the runtime. Task computations are issued within try-catch blocks and signals are treated as exceptions. After catching the exception, the runtime restores the task and reissues it

for a number of times which is provided as a configuration parameter. After these trials if the task computation does not succeed, the runtime schedules it to a new core, since most probably a permanent error is encountered in this case. If the computation does not succeed even after this migration, then the application aborts. We support global modifying task computations, such as tasks updating a shared global variable, by the privatization of the modified data (although we did not encounter any mutable global state in the applications we studied in this work).

In order to minimize the memory usage, we implement an optimization based on a concurrent hashmap. We exploit the fact that in a task parallel program, tasks are likely to share the same inputs, such as the sharing of the parts of array A by A1 and A2 tasks as in Figure 1 (a). Without this optimization, we would checkpoint the inputs of both A1 and A2 tasks. With the optimization, we checkpoint the inputs of only the task that executes first and thus decrease memory usage used for checkpointing. We implement a hashmap for maintaining the single checkpoints of task inputs. During the execution of a task parallel application, tasks insert or retrieve inputs into or from the map. Note that there is no additional synchronization needed for accessing a shared task input since by the construction of task dependencies no two tasks will be modifying a shared task input simultaneously.

IV. Experimental Evaluation

A. Methodology and Experimental Setup

We run our experiments in Marenostrum supercomputer at Barcelona Supercomputing Center. For complete system information, see [11]. We conduct experiments to assess the behaviour of NanoCheckpoints in terms of strong and weak scalability and the performance overheads with respect to fault-free executions. We obtain all results by running each single case 10 times and take the average of overheads, speedups or execution times as the final results. Our experiments are two-fold: The intra-node experiments on task-parallel shared memory benchmarks and the inter-node experiments on hybrid task-parallel MPI applications adopting the OmpSs model. Table I and II show information about shared memory and OmpSs+MPI benchmarks [12] respectively.

B. Fault Injection Mechanisms

We implement our fault injection mechanism to test the correctness of NanoCheckpoints as well as to have an injection capability to corrupt the outputs of tasks at runtime. We inject faults in the outputs of the tasks by flipping bits. We do multi-bit flips. Each flip is a random bit of a random element of the task output. We inject with different fault rates per task. We use fault rate x% to mean that the probability of a fault or faults occurring in a single task is x/100. This also corresponds to the injection of the faults approximately in x% of all tasks. We note that the fault rates in our experiments are high considering time frequency of faults in HPC systems, since the rates here are for *single* task. This effectively simulates higher fault rate for the overall system. The rationale for this is that if we achieve good performance for the upper bound case of high error rates, it would confirm our expectation that our implementations are what we call "failure scalable". Moreover, we do accelerated error injection, i.e. we simulate fault rates that are expected for future HPC systems and thus we stress test NanoCheckpoints to assess how it copes with those boosted rates. We note that since we inject at runtime, the overheads reported include also the fault injection overhead.

TABLE I
Details of Task-parallel Shared Memory Benchmarks

Sparse LU	LU decomposition Matrix size 6400x6400, block size 100x100
Cholesky	Cholesky factorization Matrix size 16384x16384, block size 512x512
FFT	Fast Fourier Transform Array size 16384x16384 (complex doubles), block size 16384x128
Perlin Noise	Noise generation to improve realism in motion pictures Array of pixels with size of 65536 (1500 iterations), block size 2048
Stream	Linear operations among arrays Array size 2048x2048 (doubles), block size 32768

TABLE II
Details of OmpSs+MPI Benchmarks

Nbody	Interaction between N bodies Array size 65536, block size depends on #nodes
Matrix Multiplication	Matrix Multiplication using CBLAS Matrix size 9216x9216 and block size 1024x1024
Ping-Pong	Computation and communication between pairs of processes Array size 65536, block size 1024
Linpack	HPL Linpack ported to OmpSs Matrix size 131072, block size 256

C. Evaluation and Discussion

Figure 2 illustrates the strong scalability of NanoCheckpoints for shared-memory benchmarks. We note that the speedups are over 1-thread case with the same fault rate. With the increasing number of threads, NanoCheckpoints scales very well except for stream. We see that it scales well up to 8 threads. The stream application does not even scale with 16 threads without NanoCheckpoints enabled. This application does not have much parallelism other than operating on different parts of arrays with different threads. It mainly consists of memory operations. This is why it does not scale (even when fault tolerance mechanisms are disabled). We purposely chose and adapted it to stress our design. Figure 3 illustrates the weak scalability of NanoCheckpoints for SparseLU benchmark. As we can see, when the input size to the benchmark is increased proportional to the number of threads, the execution time stays more or less constant in all fault rates. Our design is also weakly scaling for Cholesky, Perlin Noise and FFT but not for Stream (figures omitted for brevity). However, stream is not weakly scaling in its baseline version as well.

The checkpointing overheads of NanoCheckpoints with respect to the fault-free execution times are 0.2%, 0.3%, 7%, 0.1% and 8% for Sparse LU, Cholesky, FFT, Perlin and Stream, respectively when 16 threads are utilized. Fault-free execution time refers to the execution time of a benchmark where no injections is performed and no mechanism is enabled. As we see, the checkpointing overheads are very small with respect to total fault-free execution time. This shows that our design is highly efficient and does not degrade the performance of the applications. Our scheme improves usage of memory bandwidth due to the asynchronous and distributed

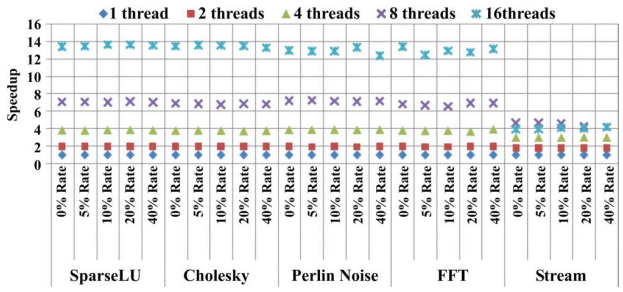

Fig. 2. Strong scalability of NanoCheckpoints in shared-memory benchmarks

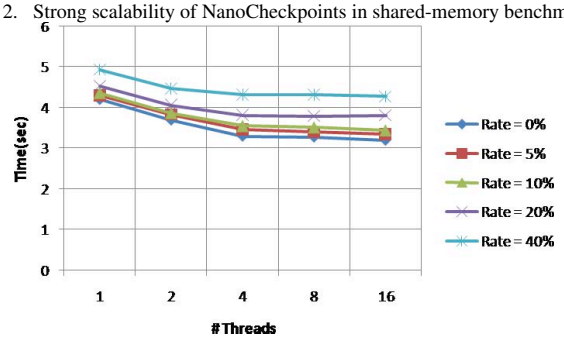

Fig. 3. Weak scalability of NanoCheckpoints: SparseLU

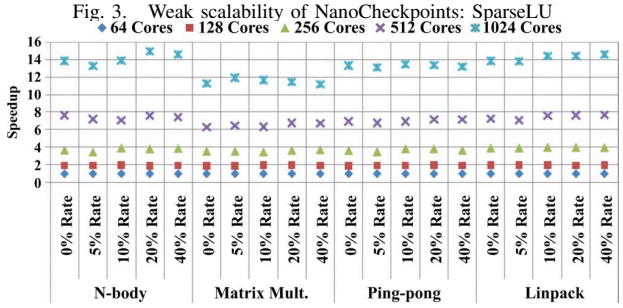

Fig. 4. Scalability of NanoCheckpoints in OmpSs+MPI benchmarks

# Threads	SparseLU	Cholesky	FFT	Perlin	Stream
1	0.985	0.988	0.933	0.988	0.770
2	0.994	0.989	0.931	0.997	0.826
4	0.998	0.989	0.928	0.998	0.808
8	0.998	0.987	0.914	0.999	0.924
16	0.998	0.989	0.930	0.999	0.922

TABLE III
EFFICIENCY OF NANOCHECKPOINTS

nature of task executions. Tasks are distributed over threads, thus checkpointing is distributed over different threads of execution which constitutes a further reason of low overhead.

We define the efficiency of a mechanism as fault-free execution time when the mechanism is disabled divided by fault-free execution time when it is enabled. Note that the closer to 1, the more efficient the mechanism is. Table III shows the efficiency of NanoCheckpoints. It is very efficient for all benchmarks except Stream on low thread counts. As stated before stream is a purposely chosen benchmark to see effects of checkpointing in an extreme case. However, as the number of threads increases, the efficiency increases close to 1, which is due to the fact that larger number of threads overlaps checkpointing and computation more effectively.

Figure 4 shows the scalability of NanoCheckpoints for hybrid task-parallel OmpSs+MPI benchmarks. The figure shows the speedups over 64 cores with the same fault rate. As seen, our mechanism is highly scalable for even much higher core counts. The checkpointing overheads in comparison to fault-free execution times when 1024 cores are utilized are 5%, 2%, 0.2% and 0.1% for N-body, Matrix Multiplication, Ping-pong and Linpack, respectively. These low overheads demonstrate that NanoCheckpoints incurs low performance overhead even for larger scales.

V. CONCLUSIONS

In this work, we introduce our fault-tolerance framework, NanoCheckpoints, based on the OmpSs PM and Nanos runtime. We discuss why OmpSs+Nanos is the right infrastructure to provide resiliency solutions. Result show that NanoCheckpoints is highly efficient and scalable. As future work we aim to support the OmpSs' heterogeneous computations.

ACKNOWLEDGMENT

This work is partially supported by the cooperation agreement between the BSC and Intel through the Intel-BSC Exascale Lab and by the Mont-Blanc project (FP7/2007-2013 under grant agreement # 288777).

REFERENCES

[1] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Lifetime reliability: Toward an architectural solution," *IEEE Micro 2005*, vol. 25, no. 3, pp. 70–80.

[2] X. Yang, Z. Wang, J. Xue, and Y. Zhou, "The reliability wall for exascale supercomputing," *IEEE Transactions on Computers 2012*, vol. 61, no. 6, pp. 767–779.

[3] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[4] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, "Support for openmp tasks in nanos v4," in *CASCON 2007*, pp. 256–259. [Online]. Available: http://dx.doi.org/10.1145/1321211.1321241

[5] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *SC 2012*, pp. 76:1–76:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389100

[6] J. Duell, P. Hargrove, and E. Roman, "Requirements for linux checkpoint/restart," http://crd-legacy.lbl.gov/~jcduell/papers/LBNL-49659.pdf, May 2002.

[7] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *ISCA 2002*, pp. 111–122. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545228

[8] V. C. Zandy and B. P. Miller, "chkpt: set of libraries and programs for user-level process checkpointing," http://pages.cs.wisc.edu/~zandy/ckpt/.

[9] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *SC 2011*, pp. 32:1–32:32. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063427

[10] A. Amer, N. Maruyama, M. Perics, K. Taura, R. Yokota, and S. Matsuoka, "Fork-join and data-driven execution models on multi-core architectures: Case study of the fmm," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer Berlin Heidelberg, 2013, vol. 7905, pp. 255–266. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38750-0_19

[11] "Marenostrum iii (2013) system architecture: http://www.bsc.es/marenostrum-support-services/mn3."

[12] B. A. Repository, "https://pm.bsc.es/projects/bar/wiki/applications."