Theft-Induced Checkpointing for Reconfigurable Dataflow Applications*

Samir Jafar†, Axel W. Krings, Thierry Gautier, and Jean-Louis Roch
Laboratoire ID-IMAG
(CNRS-INPG-INRIA-UJF – UMR 5132)
38330 Montbonnot Saint-Martin, France
{samir.jafar, axel.krings, thierry.gautier, jean-louis.roch }@imag.fr

Abstract

In this paper a new checkpoint/recovery protocol called Theft-Induced Checkpointing *is defined for dataflow computations in large heterogeneous environments. The protocol is especially useful in massively parallel multi-threaded computations as found in cluster or grid computing and utilizes the principle of work-stealing to distribute work. By basing the state of executions on a macro dataflow graph, the protocol shows extreme flexibility with respect to rollback. Specifically, it allows local rollback in dynamic heterogeneous systems, even under a different number of processors and processes. To maximize run-time efficiency, the overhead associated with checkpointing is shifted to the rollback operations whenever possible. Experimental results show the overhead induced is very small.*

1. Introduction and Background

Large parallel architectures, most notably grid and clusters, are gaining in popularity for computationally intensive applications. The computing infrastructure, consisting of a large number of computers, storage and networking devices, poses challenges in overcoming the effects of node and communication link failures. Since the computation times are often significant, effective fault-tolerance mechanisms are required to recover from faults in a fashion that avoids costly restarts.

In the absence of fault-tolerance the probability of failure, and thus the unreliability of these architectures, increases with the number of components that can fail [9]. The resulting mean time between failure (MTBF) can thereby sink below the time required by the application. As a result the execution can become infeasible. Thus, efficient mechanisms to provide fault-tolerance are not just desirable but absolutely necessary.

Recovery from faults without costly restart imply the existence of redundancy. The redundancy mechanisms must address the specific requirements associated with recovery in large heterogeneous systems. This includes taking into account a dynamic number of possibly dissimilar computational nodes. Many possible solutions based on fault-tolerance have been studied in the literature [7]. Approaches based on duplication [14] can only tolerate a fixed number of faults. More flexible approaches, e.g. log-based and checkpoint-based protocols, are based on saving the state of the processes and on constructing a consistent global state [3]. The various protocols can be compared based on three fundamental criteria. The first criterion is *coordination*, where processes coordinate each other in order to build a consistent global state at the time of checkpointing or recovery. The second is *heterogeneity*, which implies that the checkpoint state can be restored on a variety of platforms. The third criterion addresses the *scope of the recovery*, i.e. global or local recovery. If a single fault causes the roll-back of all processes in the application, one speaks of global recovery. Local recovery implies that only the roll-back of the crashed process is necessary.

Rollback-recovery methods are either *log-based*, relying on logging and replaying messages [1] , or *checkpoint-based*. Message logging is based on the fact that a process can be modelled by a sequence of interval states, each one representing a non-deterministic event [12]. Checkpoint-based methods rely on periodically saving a global state [3] of the computation to stable storage. In case of a fault, the computation is restarted from one of these previously saved states. Checkpointing-based methods differ in the way processes are coordinated and on the interpretation of a consistent global state.

Coordinated checkpointing requires the coordination of

*This work has been supported by CNRS ACI Grid-DOCG and the Region Rhône-Alpes (Ragtime project).

†The author is supported by the Damascus University

all processes for building a consistent global state before writing the checkpoints to stable storage. The disadvantage is the large latency due to coordination in order to achieve a consistent checkpoint and the need for global recovery. Its advantage is the simplified recovery without rollback propagation and minimal storage overhead, since there is only one checkpoint per process. This protocol is included in [11, 15].

Uncoordinated checkpointing assumes that each process independently saves its state and a consistent global state is achieved in the recovery phase [7]. The advantage of this method is that each process can make a checkpoint when its state is small. However, there are two main disadvantages. First, there is a possibility of rollback propagation which can result in a domino effect, i.e. rollback to the beginning of the computation. Second, the possibility of rollback propagation requires the storage of multiple checkpoints for each process.

Communication-induced checkpointing is a compromise between coordinated and uncoordinated checkpointing. To avoid a domino effect that can result from independent checkpoints of different processes, a consistent global state is achieved by forcing each process to take additional checkpoints based on some information piggybacked on the application messages [2]. The disadvantage of this approach is the need for global rollback, the possibly large number of forced checkpoints and the overhead associated with storing them.

The checkpointing tools proposed in existing systems are system-specific. Moreover, the checkpoint state is not heterogeneous, and thus not portable [4], and does not support multithreading. Portability is achieved by using portable languages like Java, but not by the checkpointing mechanism itself. Other tools, e.g. Porch [13], require recompilation to support heterogeneity at the cost of modifications of code generation with loss in optimization. The checkpointing protocol defined below will eliminate these disadvantages by introducing portability and local rollback in multithreaded environments.

2. Execution Model

At the base of the execution model is the macro dataflow model. A dataflow graph [10] allows for a natural representation of a parallel execution, and it can be exploited to achieve fault-tolerance [8]. By the principle of dataflow, tasks become ready for execution upon availability of their input data. A dataflow graph is defined as a directed graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite set of vertices and $\mathcal{E}$ is a set of edges representing precedence relations between vertices. The vertex set consists of computational tasks, as seen in the traditional context of task scheduling, and the edge set represents the data dependencies between the tasks. Within the context of this research G is a dynamic graph, i.e. it changes during runtime as the result of task creations or terminations.

2.1. Work-stealing

We adopt an online scheduling algorithm called work-stealing [5, 6] in order to distribute the workload. The principle is simple, when a process becomes idle it tries to *steal* work from another process called *victim*. The initiating process is called *thief*.

2.2. Dataflow and work-stealing in KAAPI

The target environment for multithreaded computations with dataflow synchronization between threads is the Kernel for Adaptive, Asynchronous Parallel Interface (KAAPI), implemented as a C++ library. The library is able to schedule programs at fine or medium granularity in a distributed environment. Figure 1 shows the general relationship between processors and processes in KAAPI. A processor contains one or more processes. Each process maintains its own stack.

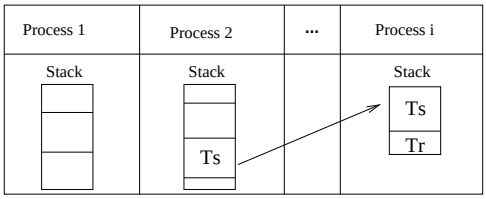

Figure 1. KAAPI processor model.

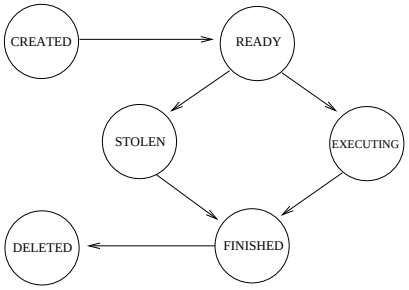

Figure 2. Life-cycle of a task in KAAPI.

The life-cycle of a task in the KAAPI execution model is depicted in Figure 2 and will be described first from a local process' and then from a thief's point of view, in the context of a task stealing.

At task creation the task enters state *created*. At this time it is pushed onto the stack. When all input data is available

the task enters state *ready*. A ready-task which is on the top of the stack can be executed, i.e. it can be popped off the stack, thereby entering state *executing*. A task in the *ready* state can also be stolen, in which case it enters the *stolen* state on the local process, which now becomes a victim. When the task is finished, either on the local process or a thief, it enters state *finished* and proceeds to state *deleted*.

If a task has been stolen, the newly created thief process utilizes the same model. In Figure 1, the theft of task T_s on Process 2 by Process i is shown, as indicated by the arrow. Whereas this example shows task stealing on the same processor, the concept applies also to stealing across processors. On the victim the stolen task is in state *stolen*. Upon theft, the stolen task enters state *created* on the thief. At this instant of time, the stolen task T_s and a task T_r charged with returning the result are the only tasks in the thief's stack, as shown in the figure. Since a stolen task by the definition of work-stealing is ready, it immediately enters state *ready*. It is popped from the stack, thereby entering state *executing*, and upon finishing, it enters state *finished*. It should be noted that the task enters this state on the thief *and* the victim. For the latter this is after receiving a corresponding message from the thief. On both processes the task proceeds to state *deleted*.

Work-stealing is the only mechanism for distributing the workload constituting the application, i.e. an idle process seeks to steal work from another process. ¿From a practical point of view the application starts with the process executing *main()*, which creates tasks. Typically some of these tasks are then stolen by idle processes, which are either local or on other processors. Thus the principle mechanisms for dispatching tasks in the distributed environment is task-stealing. The communication due to the theft is the only communication between processes. Realizing that task theft creates the only dependencies between processes is crucial to understand the checkpointing protocol to be introduced later.

It should be noted that the number of theft operations is very small in comparison to the total number of tasks executed [5, 6] and that the only nondeterministic events in the program execution are the thefts. These two properties are exploited in the checkpointing protocol defined next.

3. Theft-Induced Checkpointing

We first define the state of an execution of a parallel application using a macro dataflow graph. This graph is dynamic and can reflect changes occurring during program execution. Furthermore, it is portable, i.e. it allows the graph or portions of it to be moved during execution. Formally, at any instance of time, the macro dataflow graph G describes a platform-independent, and thus portable, consistent global state of the execution of an application.

Whereas graph G is viewed as a single dataflow graph, its implementation can in fact be distributed. Specifically, each process i contains and executes a subgraph G_i of G. Thus the state of the entire application is defined by $G = \bigcup G_i$ over all processes i. The checkpointing protocol to be presented can take advantage of this execution state formulation to allow for the rollback of only those processes that have crashed. This is due to the fact that G_i, by definition of the principle of macro dataflow, contains all information necessary to identify exactly which data is missing. ¿From a practical point of view one should note that for a process i graph G_i represents the dataflow representation of its process stack.

3.1. Definition of a checkpoint

A copy of the dataflow graph G represents a consistent global checkpoint of the application. In this research, checkpoints are with respect to a process, and consist of a copy of its local G_i, representing the stack. The checkpointing protocol must ensure that checkpoints are created in such a fashion that G is always a consistent global application state, even if only a single process is rolled back.

The checkpoint of G_i itself consists of the entries of the process stack, i.e. its tasks and their associated inputs, and not of the task execution state on the processor itself. Understanding this difference between the two concepts is crucial. Checkpointing the tasks and their inputs simply requires to store the tasks and their input data as a dataflow graph. On the other hand, checkpointing the execution of a task usually consists of storing the execution state of the processor as defined by the processor context, i.e. the processor registers such as program counters and stack pointers as well as data. In the first case, it is possible to move a task and its inputs, assuming that both are represented in a platform-independent fashion. In the latter case the fact that the process context is platform-dependent requires a homogeneous system in order to perform a restore operation or a virtualization of this state [13].

The j^{th} checkpoint of process P_i will be denoted by CP_i^j. Thus the subscript denotes the process and the superscript the instance of the checkpoint.

3.2. Checkpoint protocol definition

We will now present the checkpointing protocol called *Theft-induced checkpointing* (TIC), which was motivated by the method presented in [2]. The creation of checkpoints can be initiated by work-stealing or at specific checkpointing periods. We will first describe the protocol with respect to work-stealing, since it is the cause of the only communication (and thus dependencies) between processes. Checkpoints resulting from work-stealing are called *forced check-*

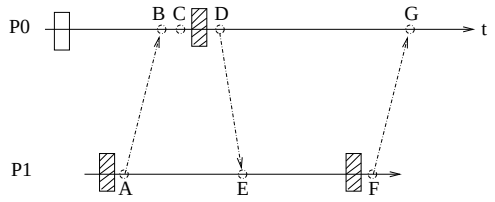

Figure 3. TIC protocol.

points. Then we will consider the periodic checkpoints, called *local checkpoints*, which are stored periodically, after expiration of the pre-defined periods τ.

3.2.1 Forced checkpoints

The TIC protocol is defined in Figure 3 with respect to events A through G for two processes P_0 and P_1. Initially P_0 is executing a task from its stack. The following sequence of events takes place:

1. A process P_1 is created on an idle resource. If it finds a process P_0 that has a potential task to be stolen, it creates a "theft" task T_t charged with stealing a task from process P_0. Before executing T_t, process P_1 checkpoints its state in CP_1^0. Event A is the execution of T_t which sends a *theft request* to P_0.

2. Event B is the receipt of the *theft request* by P_0. Between event B and C it identifies a task T_s and flags it as "stolen by P_1". Between events B and C victim P_0 is in a critical section.

3. Between event C and D it forces a checkpoint to reflect the theft. At this time P_0 becomes a victim. Event D constitutes sending T_s to P_1.

4. Event E is the receipt of the stolen task from P_0. Thief P_1 creates entries for two tasks, T_s and T_r, in its stack. Task T_r is charged with returning the results of the execution of T_s to P_0 and becomes ready when T_s finishes.

5. When P_1 finishes the execution of T_s it takes a checkpoint and executes T_r, which returns the result of T_s to P_0 in event F.

6. Event G is the receipt of the result by P_0.

3.2.2 Local checkpoints

Local checkpoints of each process i, i.e. G_i, are stored periodically, after the expiration of the pre-defined period τ. Specifically, after the expiration of τ a process receives a signal to checkpoint. The process can now take a checkpoint. However, there are two exceptions. First, if the process has a task in state *executing* it must wait until execution is finished. Second, if a process is in the critical section between events B and C in Figure 3, checkpointing must be delayed until exiting the critical section. A local checkpoint is shown in Figure 3 for process P_0 before event B.

3.2.3 TIC rollback

The objective of TIC is to allow rollback of only crashed processes. A process can be rolled back to its last checkpoint. In fact, for each process only the last checkpoint is kept. To show that one can roll back one process, while guaranteeing a consistent global state of execution, one has to consider the following two questions.

Q1 What does a process do that needs to send a message to a crashed process?

Q2 How can a process that is rolled back receive messages that it received after the last checkpoint and before the crash?

With respect to Q1, the KAAPI environment contains a process manager implemented on a reliable resource. The manager has a global view of all processes and directs the rollback of crashed processes by identifying the new process P_i' replacing the crashed P_i. An attempt to communicate with a crashed process will result in failure, indicated by an error code. The sender thus sends a message to the manager to enquire the identifier of replacement process P_i' which it uses to resend the message.

With respect to Q2, the only messages received by a process are (1) the *theft request* (event B), (2) the receipt of a stolen task (event E) and (3) the result of the stolen task (event G). We will use the events of Figure 3 in the treatment of each of the three cases.

Case (1): The loss of a *theft request* (event B) has no consequences. The thief will simply time out waiting for a response and make another request.

Case (2): If the thief crashes after receiving the stolen task (event E), but before it was able to checkpoint, it is simply rolled back as P_1' to the initial checkpoint CP_1^0 where it will re-request a task from P_0 (event A). Victim P_0, recognizing the redundant request, will change the state of T_s from *stolen* to *ready*, thus nullifying the theft, and treats the *theft request* as a new request.

Case (3): A crash of the victim after it has received the result (event G) but before it could checkpoint would stall the victim after rollback on P_0' to a state where the task is still flagged as stolen. Therefore, the manager takes the last checkpoint of the crashed P_0 and inspects it for thefts, as part of the rollback procedure. If it contains references to a thief P_1 that is already terminated, it rolls back P_0 on P_0'

using the checkpoint of P_0 together with the thief's final checkpoint containing the result. Thus, the rollback uses G_0 and G_1, which contains only T_r. If the thief is still executing, no response is necessary. The thief will request the identity of the new P_0' from the manager after the failed attempt to deliver the results to the crashed process. This occurs in event F while executing task T_r. The scenario was addressed in the context of Q1.

By addressing Q1 and Q2 we have shown that no inconsistent global state can occur as the result of rollback. However, it remains to be established why the three forced checkpoints shown (shaded) in Figure 3 are necessary for the resolutions of Q1 and Q2. Let CP_1^0 and CP_1^f denote the first and final checkpoint of a thief P_1 respectively.

The initial checkpoint CP_1^0 guarantees that there exists at least one record of a *theft request* for a thief that crashes. Thus, upon a crash, the thief is rolled back on the new process P_1'. In the worst case this is the initial checkpoint and P_1' will contact P_0 with a *theft request* with reference to its old process identifier. Without CP_1^0 any crash before a checkpoint on the thief would simply erase any reference of the theft, and would stall the victim. The final checkpoint of the thief, CP_1^f, is needed in case the victim P_0 crashes after it has received the results from the thief, but before it could checkpoint its state reflecting the result. Thus, if the victim crashes between event G and its first checkpoint after G, the actions described in the resolution of Q2 will ensure the victim can receive the result of the stolen task. It should be noted that the final checkpoint of the thief cannot be deleted until the victim has taken a checkpoint after event G, thereby checkpointing the result of the stolen task. Lastly, the forced checkpoint of the victim (between events C and D) ensures that a crash after this checkpoint does not result in the loss of the thief's computation.

The correctness of the actions associated with Q1 and Q2 was verified by enumeration over all possible failing scenarios of the victim and thief, including simultaneous faults. However, due to space limitations this enumeration could not be included in the paper.

3.2.4 Bound on rollback

Finally, one has to address the amount of work that a process can lose due to a single rollback. This is the maximal difference in time between two consecutive checkpoints. This time is defined by the checkpointing period τ and the execution time of a task, since a checkpoint of a process that is executing a task cannot be made until the task finishes execution. In the worst case, the process receives a checkpointing signal after τ and has to wait for the end of the execution of its current task before checkpointing. Thus, the time between checkpoints is bound by $\tau + \max(p_i)$ where p_i is the processing time of task T_i. But how bad can the impact of p_i be? Consider the sequential execution of a program denoted by T_1 and the execution time of the application as executed on an unbounded number of processors denoted by T_∞. In a parallel application one always assumes $T_\infty \ll T_1$. Since T_∞ is the critical path of the application any $p_i \leq T_\infty$. As a result one can assume p_i to be relatively small.

4. Experimental Results

The performance and overhead of the TIC protocol were experimentally determined for the *Quadratic Assignment Problem* (instance[1] NUGENT 22) which was parallelized in KAAPI. The experiments were conducted on the iCluster2[2]. The cluster consists of 104 nodes interconnected by a 100Mbps Ethernet network. Each node features two Itanium-2 processors (900 MHz) and 3 GB of local memory.

In order to take advantage of the distributed fashion of the checkpoint, i.e. G_i, each processor keeps a local copy of its checkpoint. To eliminate this single source of failure, it is assumed that the checkpoint of each G_i is replicated on other nodes [15]. This configuration has the advantage that one can measure the actual overhead of the checkpointing mechanism, rather than the overhead associated with a centralized checkpoint server.

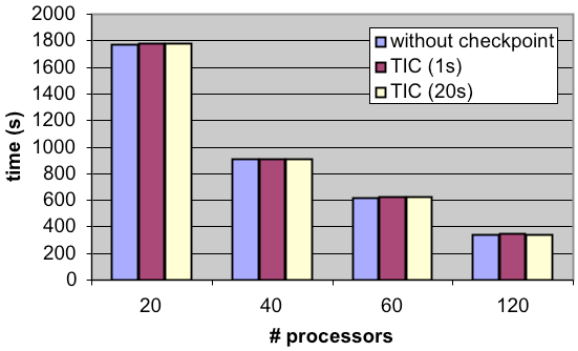

Figure 4. TIC overhead.

Figure 4 shows the overhead of checkpointing when executing the application on configurations using different numbers of processors. For each processor configuration the execution times without checkpointing and using TIC with period $\tau = 1s$ and $\tau = 20s$ are shown. As can be seen, there is very little observable overhead for each processor configuration, which shows that the overhead of TIC is negligible. However, there is small, but hardly visible overhead.

[1] see http://www.opt.math.tu-graz.ac.at/qaplib/

[2] http://www.inrialpes.fr/sed/i-cluster2

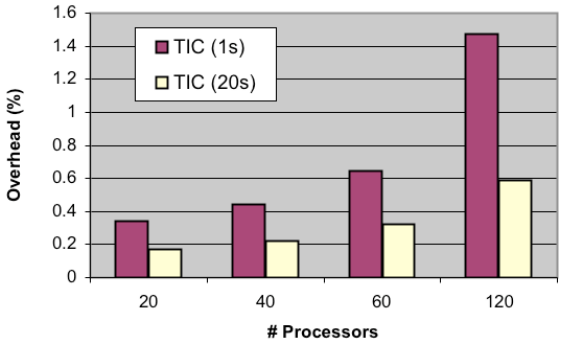

Figure 5. Relative TIC overhead.

The exact relative overhead for TIC is shown in Figure 5. In general, there is an increase in overhead with the number of processors. The reason for this is that as the number of processors increase, so does the number of forced checkpoints due to work-stealing. As expected, the overhead is smaller for larger τ. Whereas Figure 5 shows noticeable difference in overhead, one should note however that the measured overhead is less than 1.5% in any configuration. This we consider negligible in view of Figure 4.

5. Conclusions and Future Work

In order to address fault-tolerance of large parallel applications we have introduced Theft-Induced Checkpointing. The protocol has the main advantage that it only requires crashed processes to be rolled back. The state of the application is represented in a portable fashion utilizing macro dataflow graphs. This allows for a platform-independent description of the application state, which makes it possible to rollback in a dynamic environment, even when the number of processors changes. The overhead associated with checkpointing was shown to be very low, to the point of being negligible. The amount of work lost by a crashed process, determined by the spacing between checkpoints, could be controlled by the checkpointing period and the application's granularity, which affects task execution times.

References

[1] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. *TSE*, 24(2):149–159, 1998.

[2] R. Baldoni. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In *Proc. 27th Intl. Symposium on Fault-Tolerant Computing (FTCS '97)*, page 68. IEEE Computer Society, 1997.

[3] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[4] M. Litzkow et.al. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report CS-TR-97-1346, Univ. Wisconsin, Madison, 1997.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proc. ACM SIGPLAN 1998*, pages 212–223. ACM Press, 1998.

[6] F. Galilée et.al. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *PACT'98*, pages 88–95, October 1998.

[7] E.N. Mootaz et.al. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[8] A. Nguyen-Tuong et.al. Exploiting data-flow for fault-tolerance in a wide-area parallel system. In *Proc. 15 th Symposium on Reliable Distributed Systems*, pages 2–11, 1996.

[9] Trivedi K. S. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley and Sons, New York, 2001.

[10] J. Silc et.al. *Asynchrony in parallel computing: from dataflow to multithreading*, pages 1–33. Nova Science Publishers, Inc., 2001.

[11] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[12] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[13] V. Strumpen. Compiler technology for portable checkpoints. Technical Report MA-02139, MIT Laboratory for Computer Science, 1998.

[14] M. Wiesmann et.al. A systematic classification of replited database protocols based on atomic broadcast. *Proc. 3^{rd} European Research Seminar on Advances in Distributed Systems(ERSADS99)*, pages 351–360, 1999.

[15] G. Zheng, L. Shi, and L. V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.