CCK: An Improved Coordinated Checkpoint/Rollback Protocol for Dataflow Applications in KAAPI

Xavier Besseron, Samir Jafar*, Thierry Gautier and Jean-Louis Roch

Projet MOAIS(CNRS/INPG/INRIA/UJF) – Laboratoire ID-IMAG(UMR 5132)
Monbonnot ZIRST/51 avenue Jean Kuntzmann – 38330 Monbonnot – FRANCE
Email:(Xavier.Besseron, Samir.Jafar, Thierry.Gautier, Jean-Louis.Roch)@imag.fr

Keywords. Parallel Application, Dataflow Graph, Checkpoint/Recovery.

Abstract

Fault tolerance protocols play an important role in today long runtime scientific parallel applications because the probability of failure may be important due to the number of unreliable components involved during simulation. In this paper we present our approach and preliminary results about a new checkpoint/recovery protocol based on a coordinated scheme. This protocol is highly coupled to the availability of an abstract representation of the execution.

1 Introduction

Since few years, fault-tolerance has been studied in the context of scalable parallel applications which allow to make simulation of complex phenomena using large scale cluster [10, 3]. Due to the number of unreliable components involved during the computation, the apparition of faults is not an exceptional event: the system or the middleware should provide fault-tolerance protocols so as to mask failures. The subject has been well studied in the context of distributed systems and distributed middlewares [4, 6]. The renewed interest is that optimizing performance becomes a major objective. Recent propositions study the runtime behavior of applications in order to specialize or extend published protocols [9, 3].

The idea behind this research direction wants to automatically adapt a fault-tolerance protocol to the minimal requirements of an application about dependability features. This paper is in this context: the specialization of fault-tolerance protocol is done at the level of an abstract representation of the execution which permits important optimizations at runtime. We based our work in the framework of KAAPI [10, 9], where the abstract representation of execution was firstly designed to be able to plug scheduling algorithms independently of applications. In [10] it was shown that this abstract representation is well suited for defining the checkpoint of local process. In the context of this paper, this abstract representation is used to specialize a fault-tolerance protocol for long runtime iterative simulation.

Coordinated checkpoint/rollback protocols are promising for large scale parallel applications because they do not add extra overhead on communication and current experiments demonstrate their availability to scale up to thousands of processors [6, 3], including the global synchronization. In case of fault, all the processors restart from their most recent checkpoints, even those which did not fail. The two challenging problems about performances of coordinated checkpoint/rollback protocols are:

1. How to speed up restart of processors after the occurrence of a fault?
2. How to reduce the amount of lost computation time in case of fault?

In [6, 3] the solution to solve (1) is: each processor keep a local copy of its checkpoint and send an other copy to either a stable storage [3] or either to a fixed number of neighbor processors [6]. Within this approach, all processors except the failed processor, restart from their local copy of the most recent checkpoint.

Our contribution is mainly to propose a solution for (2). Thanks to the abstract representation of execution of any KAAPI's applications, it is possible to compute the strictly required set of computation to resend messages to the failed processor. Moreover by adapting the local scheduling of tasks we present an optimization that may improve this required computation without impacting the parallel performance of the execution.

The outline of the paper is the following. The next section deals with related works. Section three presents our improved coordinated checkpoint/rollback protocol for KAAPI applications. It begins with an overview of

*This author is supported by a grant of the Syrian Government

the abstract representation in KAAPI, the definition of the local state of process. Then we presents the protocol and an analysis of its complexity is sketched. The next section presents preliminary experimentations on virtual reality applications to simulate clothes using a physical model. The conclusion ends the paper.

2 Related works

In this paper we deal with long runtime of parallel applications with high ratio communication versus the computation. Such kind of applications appear during iterative simulation of physical phenomena: for instance molecular dynamics [12], virtual reality [17] or domain decomposition applications. Fault-tolerance protocols have been classified in three categories: those based on duplication to introduce redundancy of computations; protocols based on event logging and protocols based on checkpoint/rollback approach. Because protocols based on duplication only tolerate a fixed number of faults and may consume a lot of resources they are not selected [4, 6, 3] when the criteria is to minimize the completion time. Log-based protocols assume that the state of the system evolves according to non-deterministic events (message reception in this case). These events are logged in order to rollback from a previous saved checkpoint [4]. But experimentations [3] have shown an important overhead for communication intensive applications. Checkpoint/rollback protocols periodically save the state of the local process of the applications and have few overhead with respect to the communication. They come in three forms. Uncoordinated protocols make no assumption about the coherency of the global state captured and may be impacted by the domino effect: in the worst case, the application is required to be rollback at the beginning [16]. Coordinated protocols are based on global synchronization to ensure that the set of local checkpoints forms a global coherent state. Communication-Induced checkpointing protocols [1] are a mix between coordinated and uncoordinated protocols where forced checkpoints are computed on receiving same messages.

3 Improved coordinated checkpoint/rollback protocol

The idea of the CCK[1] protocol is to build after occurrence of fault, the computation of every processors that is strictly required to resend messages to the failed processor. Thanks to KAAPI, the computation to re-execute is fewer than in classical coordinated protocols [4, 6, 3]. This section presents how to be able to reduce the number of instructions to be re-executed in the context of KAAPI environment. We first describes the execution model and the abstract representation of execution in KAAPI. Then we deals with the improved coordinated protocol.

[1]Coordinated Checkpointing in Kaapi

3.1 Execution model and abstract representation

In KAAPI an API, for instance Athapascan [5, 7], allows to define the tasks and the data in global memory that task produces or consumes. A task is closed to a function call with addition to the mode of access to the data in the global memory (essentially read or write access). At the difference of MPI, the communication is not explicit in KAAPI but deduce from the mapping of tasks among the processors and the mode access to the global memory. The reader interested in the API of KAAPI may refer to [5, 7]. A runtime interpretation of some instructions of the API allows to unfold the data flow graph of the (future) execution of the program. Then KAAPI schedules and executed them on processors.

At the base of the execution model is the macro dataflow model. A dataflow graph [8] allows for a natural representation of a parallel execution, and it can be exploited to achieve fault-tolerance [14, 10]. 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, *i.e.* an edge $t_i \rightarrow t_j$ means that task t_i produces data that task t_j consumes. 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.

This data flow graph is called the **abstract representation** of the application because this representation is casually connected to the (execution of the) application: any new execution of instruction of the API is reported by the creation of new vertices in the data flow graph; and any modification in the data flow graph is traduced to a modification in the execution of the application. For instance, the scheduling algorithms that are used to compute which tasks are affected by which processes: the data flow graph is distributed among the processes and the execution of the application reflect this by having (generally) speedup in comparison to the sequential execution.

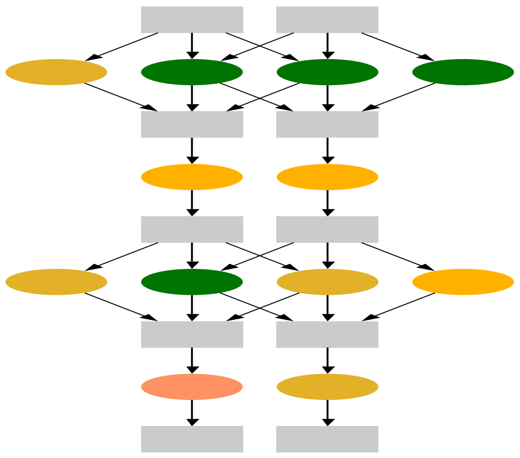

Figure 1: Graphical representation of the abstract representation of execution as data flow graph. Boxes represent global data. Ellipses represent tasks and edges represent data that are produced or consumed by task.

During the distribution of the data flow graph, communications are generated: task that produces data to task mapped into different process is executed at runtime by a inter-process communication.

The KAAPI has some assumptions and properties about the way programs are scheduled and executed for which presentation is outside the scope of this paper.

Proposition 1 *Abstract of* KAAPI *properties.*

1. *The sequential execution of* KAAPI *is a valid execution order.*
2. *Any schedule that respects the data flow constraints between tasks is a valid schedule.*
3. *If two tasks have accesses to the same data in global memory, then they share a same parent task.*

3.2 Definition of a checkpoint

A copy of the abstract representation 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 a data flow graph. The checkpointing protocol must ensure that checkpoints are created in such a manner that G is always a consistent global application state, even if only a single process is rolled back.

Definition 1 *The* ***checkpoint*** *of G_i itself consists of the entries of the data flow graph, 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 [18].

3.3 Coordinated protocol definition

The checkpointing protocol called *Coordinated Checkpointing in* KAAPI *(CCK)*, was motivated by the method presented in [11]. The protocol is coordinated by one process at specific checkpoint periods. The algorithm is sketches in figure 2. P_i denotes the process with index i. The process with index 0 is assumed to be the coordinator.

```
1. Synchronisation of all processes
    1.1 $\forall i \in \{0, ..., N\}$ , send message START to $P_i$
    1.2 Wait $N$ receptions of message ACK

2. Store local checkpoint

3. Continuation of all processes
    1.1 $\forall i \in \{0, ..., N\}$ , send message CONT to $P_i$
```

Figure 2: CCK protocol definition (coordinator code)

The role of the synchronization step is to flush in-transit message in communication channel between processes. Note that the protocol assumes that the messages are FIFO ordered between pair of processes. Upon reception of messages, all processes react using the rules defined in figure 3.

If an error occurs during the execution of the checkpoint phase of the protocol, then the phase is aborted and the failed process is restarted.

3.4 Restarting failed process

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_{new} replacing the crashed P_{failed}.

```
1. If message received is START then
    1.1. reset all counters for the CCK protocol
    1.2. stop local computation
    1.3. $\forall\, i \in \{0, ..., N\}$ , send message FLUSH

2. If message received is CONT then
    2.1. continue local stopped computation

3. If message received is FLUSH then
    3.1 if number of such message is $N + 1$ then send
    to coordinator message ACK

4. If message received is ACK then
    4.1 if number of such message is $N + 1$ then
    wakeup coordinator
```

Figure 3: Reactions to incoming message

In case of failure of process P_{failed}, a fault tolerance detector[2] informs the manager to restart the process. The newly created process P_{new} starts its computation from the previous checkpoint of process P_{failed}. All other process will restart the strictly required set of computation that will generates communication to destination P_{failed} in order to re-send lost messages.

Definition 2 *The **strictly required set of computation** for a process P_i with respect to a process P_k is the minimal set of tasks stored in the previous checkpoint of P_i which are executed on P_i and which produce data that will be directly or not directly send to P_k.*

This operation is to compute the set of tasks that produce a data that will be send to process P_{failed} by analyzing the data flow graph stored in the previous checkpoint of each process P_i. The demonstration that all lost messages is re-send is based on the properties of the KAAPI execution model and the way the work is distributed among the processes. The coordination flushes all in-transit messages which imply that the set of local checkpoints is a global coherent state of the execution. If P_{failed} should have received a message from P_i, then it imply that P_i has task that will produces a data consumed by task in P_{failed}, this is a consequence of the KAAPI properties presented in section 3.1.

Thus each local checkpoint of process P_i (a data flow graph with its inputs) represents *all* the future of execution of P_i, including the tasks that produce data send to P_{failed}. An analysis of the data flow dependencies between tasks allows to compute the strictly required set of computation necessary to re-send data to P_{new} in place of P_{failed}.

[2]We assumed that such detector is part of the fault-tolerance framework, its presentation is outside the scope of this paper.

Figure 4 sketches the restart of a failed process P_{failed}.

```
1. Read the previous checkpoint of $P_{failed}$

2. Analyze the data flow graph of the checkpoint in
order to compute the list of processes for which $P_{failed}$
has to receive data

3. Send message RESTART to all these process

4. Restart local computation
```

Figure 4: Pseudo code of a process P_{new} that restart process P_{failed}.

On reception of the message RESTART, the process P_i compute the strictly required set of computation necessary to re-send data to P_{failed} and updated to send to P_{new}.

3.5 Complexity analysis

In this section we analyse the complexity of the execution with a fault in comparison to the complexity of traditional coordinated checkpoint protocol [4, 6, 3] that restart all processes when one is faulty.

The worst case of our protocol is the case where the strictly required set of computation of P_i with respect to P_{failed} contains all executed tasks on P_i. If it is true for all processes P_i, then the complexity of our protocol is the complexity of the traditional protocols plus the complexity to do the analysis of the data flow graph in order to compute the strictly required set of computation. This latter complexity is linear with respect to the number of tasks in the data flow graph to analyze.

Nevertheless, for some class of parallel applications, the complexity of our algorithm is lesser than traditional coordinated protocols on two points:

1. The number of processes involved in the restart of P_{failed} is less that the total number of processes that have to restart for classical protocol. Moreover, this number may be a constant.

2. The number of tasks in the *strictly required set of computation* is generally less than the executed tasks.

The point 1 is due to the fact that the knowledge of the data flow graph permits to the protocol to know the communication between processes. The point 2 is due to the nature of the dependencies on some application, especially in the case of numerical scalable simulation application that exhibit good locality of (remote) data accesses. Moreover, the number of tasks in the strictly required set of computation may be decreased if the scheduling of tasks try to execute first the tasks that generate communications: at the period of the checkpointing operation, such tasks will be not contained into the checkpoint and thus they will do not be re-executed again.

4 Case of study

In this paper, we present some results about of a cloth simulation [19, 17] written on top of KAAPI using the Athapascan API. This application is called SAPPPE. It is a cloth simulation [2] that provides a 3D and realistic modelling of dressed humans in real time. SAPPPE is based on a physical model: a cloth is represented as a triangular mesh of particles linked up by springs emulate the material properties. The mesh topology describes how particles interact and exert forces on each other.

The experiments were conducted on a cluster of Grid5000[3]. The cluster consists of 32 nodes interconnected by a Fast Ethernet network. Each node features two AMD Opteron processors (2.0 GHz) and 2 GB of local memory.

4.1 Presentation of the parallelisation

The loop iteration of each simulation is composed of two main parts: (1) Computation of forces that act on each particle or atom; (2) Computation of each particle or atom states (acceleration, velocity, position) by integrating the dynamic equations of the system.

To design parallel algorithms for such simulations, computations are partitioned in a set of tasks. The technique is based on a particle decomposition [19]. It consists in splitting the set of particles in several subsets using algorithm of library such as Scotch [15] or Metis [13]. The interactions between particles leads to many data dependencies between tasks. The parallelisation is achieve by unrolling the main loop over few iterations (typically 2 or 3) and by computing a schedule of the tasks generated. The same schedule will be used to the tasks of the next iteration.

[3]http://www.grid5000.fr/

4.2 Cost of the CCK protocol

The experiment reported in figure 5 presents the times of using the CCK protocol when the number of processors is increasing. The deployment architecture uses only one checkpoint server. The overhead of the experiment with 2000 tasks with respect to the experiment with 200 tasks is due to the bottleneck on the checkpoint server. In fact, for the experiment with 2000 tasks, checkpoint file of each process is about 300KBytes, for 40 processors, the checkpoint server receives about 12MBytes of data.

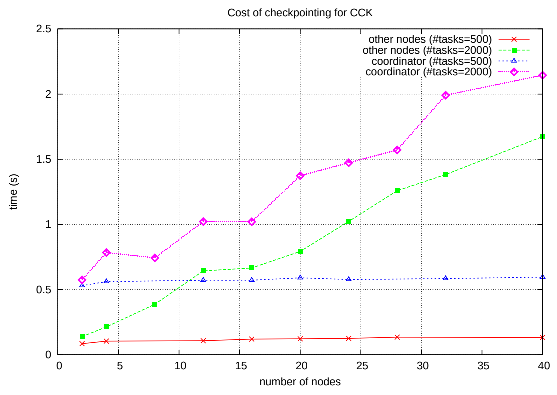

Figure 5: Times of the CCK protocol with respect to the number of processes. Two experiments with 500 and 2000 tasks per process.

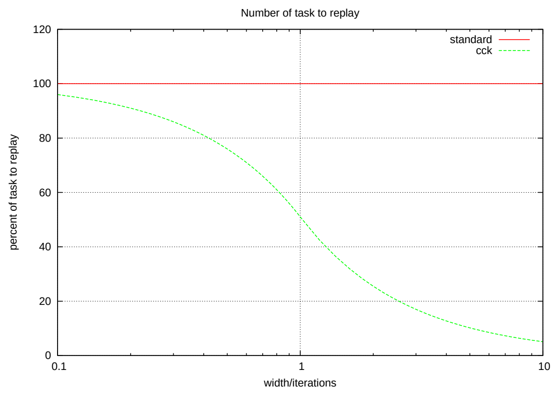

Figure 6: Average number of tasks to re-execute on a process that communicates to process failed.

4.3 Average number of re-executed tasks

The figure 6 reports the number of tasks that is re-executed using our protocol with a classical coordinated protocol. The measures reported for alive process that should redo message to the faulty process. The X-axe is the iteration step when a checkpoint is done stored. When the fault appears after the checkpoint, the number of tasks to re-executed is very small because most of the communication tasks are in checkpoint due to the schedule.

5 Conclusion

In this paper we have presented a new coordinated checkpoint protocol for fault tolerance mechanism for parallel iterative simulation. The originality of our work comes from the abstract representation provided by the KAAPI library for any parallel execution of applications. The main contribution is to show how to improve classical coordinated checkpoint protocol by using a better knowledge of the application and especially about the dependencies between processes due to communication. We have shown that 1/ the number of processes that require to redo computation may be small; 2/ the restart delay for all involved processes is generally less than what is required by classical protocol. Ongoing work is to experiment the protocol on a real grid.

References

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

[2] D. Baraff and A. Witkin. Large steps in cloth simulation. In *Computer Graphics Proceedings, Annual Conference Series*, pages 43–54. SIGGRAPH, 1998.

[3] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. In *In proceedings of The 2003 IEEE International Conference on Cluster Computing*, Honk Hong,China, 2003.

[4] E. N. Mootaz Elnozahy, L. Alvisi, Y.-M. Wang, and Johnson D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[5] G. Cavalheiro M. Doreille F. Galilée, J.-L. Roch. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[6] L. V. Kalé G. Zheng, L. Shi. 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.

[7] R. Revire J. L. Roch, T. Gautier. Athapascan: Api for asynchronous parallel programming. Technical Report RT-0276, `www-id.imag.fr/software/ath1`, Projet APACHE, INRIA, February 2003.

[8] T. UNGERER J. Silc, B. ROBIC. *Asynchrony in parallel computing: from dataflow to multithreading*, pages 1–33. Nova Science Publishers, Inc., 2001.

[9] S. Jafar, T. Gautier, A. Krings, and J-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In *Proceedings of (LNCS) EuroPar'05*, Lisboa, Portugal, August 2005.

[10] S. Jafar, A. Krings, T. Gautier, and J-L. Roch. Theft-induced checkpointing for reconfigurable dataflow applications. In *Proceedings of the IEEE Electro/Information Technology Conference EIT2005*, Lincoln, Nebraska,U.S.A., May 2005.

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

[12] Laxmikant Kal, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Artiomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. Namd2: greater scalability for parallel molecular dynamics. *J. Comput. Phys.*, 151(1):283–312, 1999.

[13] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. Technical report, 1997.

[14] A. Nguyen-Tuong, A. S. Grimshaw, and M. Hyett. Exploiting data-flow for fault-tolerance in a wide-area parallel system. In *Proceedings 15 th Symposium on Reliable Distributed Sysstesm*, pages 2–11, 1996.

[15] F. Pellegrini and J. Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical Report 1038-96, 1996.

[16] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, 1975.

[17] R. Revire, F. Zara, and T. Gautier. Efficient and easy parallel implementation of large numerical simulation. In Springer, editor, *Proceedings of ParSim03 of EuroPVM/MPI03*, pages 663–666, Venice, Italy, 2003.

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

[19] F. Zara, F. Faure, and J-M. Vincent. Physical cloth simulation on a pc cluster. In X. Pueyo D. Bartz and E. Reinhard, editors, *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany, September 2002.