Optimised recovery with a coordinated checkpoint/rollback protocol for domain decomposition applications

Xavier Besseron and Thierry Gautier

MOAIS Project
Laboratoire d'Informatique de Grenoble
ENSIMAG - Antenne de Montbonnot
ZIRST 51, avenue Jean Kuntzmann
38330 Montbonnot Saint Martin, France
(xavier.besseron, thierry.gautier)@imag.fr

Abstract. Fault-tolerance protocols play an important role in today long runtime scientific parallel applications. The probability of a failure may be important due to the number of unreliable components involved during an execution. In this paper we present our approach and preliminary results about a new checkpoint/rollback protocol based on a coordinated scheme. One feature of this protocol is that fault recovery only requires a partial restart of other processes thanks to the availability of an abstract representation of the execution. Simulations on a domain decomposition application show that the amount of computations required to restart and the number of involved processes are reduced compared to the classical global rollback protocol.

Key words: grid, fault-tolerance, parallel computing, data flow graph

1 Introduction

Since few years, fault-tolerance has been studied in the context of high-performance parallel applications that makes use of large scale clusters or grids (i.e. simulation of complex phenomena) [1–6]. Due to the number of unreliable components involved during the computation, the apparition of faults is not an exceptional event [7, 8]: the system or the middleware should provide fault-tolerance protocols in order to mask failures. Moreover, some applications require an important computation time to complete (like a week running on a thousand processors [9]). Exclusive reservation of computing resources during such a period conflicts with reservation policies aiming at fairness between users on short periods. In this case, fault-tolerance allows to split a large computation and run it during many shorter separated reservations [10, 11].

This subject has been well studied in the context of distributed systems and distributed middlewares [1, 2, 12, 13]. Optimising performance on large scale architectures becomes a major objective. Recent propositions study the applications runtime behaviour in order to specialise or extend published protocols [5, 6, 14, 15]. This is the context of our paper.

In our document the specialisation of fault-tolerance protocol is done using an abstract representation of the execution offering important optimisations at runtime. We implemented this work in the framework of KAAPI [4, 14, 16], where the abstract representation of execution was firstly designed to plug scheduling algorithms independently of applications. In [4, 6], it was shown that this abstract representation is well suited for defining the local process checkpoint. In this paper, this abstract representation is used to specialise a fault-tolerance protocol for long runtime intensive iterative simulation where the communications versus computing ratio is high.

Experiments carried out in [2, 5] show that coordinated checkpoint/rollback protocols are efficient up to thousands of processors. In case of fault, all the processors restart from their most recent checkpoint, even those which did not failed. The two challenging problems about performances of coordinated checkpoint/rollback protocols are:

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

In [2, 5] the solution to solve (1) is: each process keeps a local copy of its checkpoint and sends another copy to either a stable storage [5] or a fixed number of neighbour processes [2]. Within this approach, all processes except the failed process, 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 applications, it is possible to compute the strictly required computation set which is the computation task set that a processor have to re-execute to resend lost messages to the failed processor. This optimisation reduces the amount of computation required to restart the application. Furthermore, if the task set is parallel enough, it can be scheduled over all the processors to speed up the restart.

The outline of the paper is the following. The next section deals with related works. Section three presents the improved rollback of our coordinated checkpoint/rollback protocol. It begins with an overview of the abstract representation in KAAPI and the process state definition. Then we present the recovery step and an analysis of its complexity is sketched. The next section presents a study case on a domain decomposition application and simulations of its restart. The conclusion ends the paper.

2 Related works

In this paper we deal with long runtime of parallel applications with a high ratio communication versus computation. Such kind of applications appear during iterative simulation of physical phenomena: for instance molecular dynamics [17], virtual reality [18]. Parallelisation of such applications uses domain decomposition method: the simulation domain is splitted into smaller subdomains. During an iteration, each processor communicates with its neighbours according to subdomain relationship.

Fault-tolerance protocols have been classified in three categories [1]: those based on duplication to introduce redundancy of computations [12, 19]; protocols based on event logging [20] and protocols based on checkpoint/rollback approach [1, 21].

Protocols based on duplication only tolerate a fixed number of faults and may consume lots of resources [19]. Since the main criteria for the considered applications is the performance, and moreover, an interruption during the computation can be tolerated, they are not selected.

Log-based protocols assume that the state of the system evolves according to non-deterministic events. These events are logged in order to rollback from a previous saved checkpoint [1]. In our case, non-deterministic events are communications between subdomains which represent a large amount of data. So these protocols are not selected, they require too many resources (memory, bandwidth) [3].

Checkpoint/rollback protocols periodically save the local process state of the applications and have few overhead with respect to the communications. They come in three forms depending on the way they build a coherent global state for the application restart [1]. Uncoordinated protocols make no assumption about the coherency of the global state captured and may be impacted by the domino effect: in worst case, the application is required to rollback at the beginning [22]. Coordinated protocols are based on global synchronisation to ensure that the set of local checkpoints forms a coherent global state [21]. Communication-induced checkpointing protocols [23] are a mix between coordinated and uncoordinated protocols where forced checkpoints are computed on reception of some messages.

Coordinated checkpoint/rollback protocols have the advantage of having a low overhead towards application communications [2, 5]. However, they produce a large communication volume due to the checkpoints size which are sent simultaneously to the checkpoints servers. This can be amortised by choosing a suitable checkpoint period [3] or using incremental checkpoints [24].

3 Improved coordinated checkpoint/rollback protocol

The idea of the Coordinated Checkpointing in Kaapi (CCK) protocol is to build after fault occurrence, the computations of every processes that are strictly required to resend messages to the failed processor. Thanks to Kaapi, the amount of computation to re-execute is less than in classical and improved coordinated protocols [1, 2, 5] for which all the processors restart from their last checkpoint.

This section presents how to reduce the number of instruction to re-execute using the execution abstract representation provided by Kaapi. We first describe the execution model and the abstract representation of Kaapi. Then we deal with the optimised recovery.

3.1 Execution model and abstract representation

KAAPI[1] [16] is a middleware that allows to execute distributed and/or parallel applications. It offers a high level parallel programming model. The programmer writes his program describing potential parallelism independently of the target architecture, using for example the ATHAPASCAN [25, 26] programming interface.

With ATHAPASCAN, the parallelism is defined with two simple concepts: *shared data* and *tasks*. A shared data is a data in global memory that a task can produce or consume. A task is an indivisible instruction set that declares an access mode to a shared data (read or write). With this description, KAAPI can execute the application according to the *precedence constraints* which are dynamically detected.

The set {tasks, shared data, precedence constraints} builds the data flow graph representing the application execution [26]. A data flow graph is defined as a directed graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite vertex set (tasks and shared data) and $\mathcal{E}$ is an edge set (precedence constraints) between vertices. This data flow graph is called the *abstract representation* of the application. This representation is causally connected to the (execution of the) application: any new execution of an API instruction is reported by the creation of new vertices in the data flow graph; and any modification in the data flow graph is rendered in a modification in the application execution. For instance, the data flow graph is distributed among the processes and the application execution reflects this by having (generally) speedup in comparison to the sequential execution.

For the application aimed in this paper, we use the following approach, called *static scheduling* [11], to execute the data flow graph. First, a pluggable library like SCOTCH [27] or METIS [28] partitions the data flow graph of one iteration in N data flow subgraphs where N is the wanted processor number to run on. For each subgraph, the static scheduling KAAPI engine automatically generates the tasks for the required communications. Then data flow subgraphs are distributed over all the processors and they execute their subgraph iteratively. If no modification of the data flow graph occurs between iterations then subgraphs are reused without recomputing them.

3.2 Definition of a checkpoint

The application state is represented by the state of all its processes and by the state of communication channels. Because the communication channels' state is not available, the principle of coordinated protocols is the synchronise all the processes and to flush all in-transit messages in order to checkpoint the application. Under this condition, the application state is made of the union of all the process local states [21].

The process state can be save using its abstract representation as a data flow graph G_i (which is composed of the graph and its input data). Moreover, this state is independent of the computer executing the process (hardware, operating system) if it is saved between the execution of two tasks.

[1] `http://kaapi.gforge.inria.fr`

Definition 1. *The checkpoint G_i of process P_i is composed of its data flow graph, i.e. its tasks and their associated inputs. It does not depend on the task execution state on the processor itself.*

Finally, a coherent global state G of the application is the set of all the local checkpoints G_i which are saved during the same coordination step.

The checkpointing step of CCK protocol implemented in KAAPI is based on the classical coordinated checkpointing protocol presented in [21] and on optimisations proposed in [29]. It is fully detailled in [30, 31].

3.3 Recovery after failures

When one or many processes fail during the computation, the role of a checkpoint/rollback protocol is to restart the application in a state that could happen in a normal execution (i.e. without failure). At the failure time, the application is composed of two kind of processes: failed processes and non-failed processes. The last checkpoint of all processes is available and all these checkpoints form a coherent global state of the application before the failure. Furthermore, the current state of the non-failed processes is known.

In the case of the classical rollback protocol [21], all processes would restart from their last checkpoint (failed processors are replaced using spare processors). However, all computations performed on all the processes since the last checkpoint step are lost. This waste can be important specially when a large processor number is used.

The CCK rollback protocol try to reduce this waste. The substituting processes that replace failed processes have to restart from the last checkpoint because the failure made failed processes loose their current state. As for the non-failed processes, they keep their ongoing computation. Because the global state made of the states of substituting processes and non-failed processes is not coherent, the computation can't continue from this state. Analysing the execution abstract representation as a data flow graph allow us to identify, among the last checkpoint of non-failed processes, the *strictly required computation set* that need to be re-executed so as to guarantee that this global state is coherent.

Definition 2. *The* ***strictly required computation set*** *for a process P_i with respect to a process P_k is the minimal task set stored in the previous checkpoint of P_i which have already been executed on P_i and which produce, directly or indirectly, a data that will be send to P_k.*

The distributed algorithm that determines the strictly required computation set to re-execute is detailed in [31]. This algorithm computes the task set which produces data that will be send to failed processes by analysing the data flow graph stored in the previous checkpoint of each process. The demonstration that all lost messages is re-send is based on the properties of KAAPI execution model and data flow graph's. The coordination flushes all in-transit messages which imply that the set of local checkpoints is a coherent global state; so if a failed process P_{failed} should have received a message from process P_i, then there is a task in P_i that will produce the data consumed by task in P_{failed}.

3.4 Complexity analysis

In this section we analyse the execution complexity with a fault in comparison to the complexity of classical coordinated checkpoint protocol [21] that restart all processes when one is faulty.

The worst case for our protocol is the case where the strictly required computation set 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 our protocol's complexity is the traditional protocols' complexity plus the complexity to analyse the data flow graph in order to compute the strictly required computation set. This latter complexity corresponds to the computation of transitive closures on the graphs, which is linear with respect to the task number in the data flow graph to analyse because they are acyclic and directed [32].

Nevertheless, for the considered class of parallel applications, our algorithm's complexity is lesser than the classical coordinated protocol on two points:

1. The number of involved processes in the restart of P_{failed} is less that the total number of processes that have to restart for the classical protocol. Moreover, this number may be a constant.
2. The task number in the strictly required computation set 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 know the communications between processes. The point 2 is due to the nature of the dependencies on some applications, especially in domain decomposition applications that exhibit good locality of (remote) data accesses because most of the computations use data from the process itself, only a few computations require data from other processes. These processes are bordering processes (according to subdomain relationship) and are in constant number.

4 Simulations

In this section we present simulations of the recovery step of CCK after one process failed. We consider an application that solves the Poisson problem on a large domain to study gains with a large processor number. The application uses the Jacobi method on a three dimensional domain. The size of the domain is $2,048^3$ (64 GB) split in 64^3 subdomains of 32 KB size. For each computation iteration, a subdomain update corresponds to one computation task. The execution of this task requires the neighbour subdomains. On a reference computer (Bi-Opteron 2 GHz CPU with a 2 GB memory), the execution of one computation task lasts 10 ms.

4.1 Checkpoint period influence

For this simulation, the 64^3 submains are distributed on 1,024 processors, so there are 256 subdomains (64 MB) for each process. In this case, the execution of one iteration (i.e. the update of all the subdomains) last about 2.5 seconds.

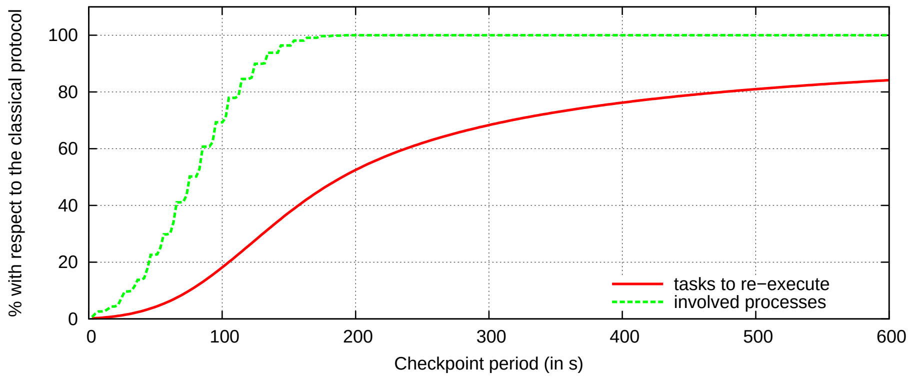

Fig. 1. Proportion in the worst case of tasks to re-execute and of involved processes for CCK restart with respect to the classical protocol

The figure 1 shows the proportion, with respect to the classical protocol, of tasks to re-execute and of involved processes for the CCK restart in relation to the checkpoint period. The curve shows the worst case values, i.e. when the failure happens just before the next checkpoint. With a 60-second period, less than 30 % of the processes are involved and only 6 % of the tasks have to be re-executed with respect to the classical protocol.

In order to reduce the restart time, the task set to re-execute can be distributed on all the processors. In this case, the estimated restart time is 3.6 seconds for CCK instead of 60 seconds for the global restart of the classical protocol. To this time, we have to add the time to identify and to distribute the strictly required task set. These will be evaluated in future experimentations.

4.2 Processor number influence

The two next simulations show the processor number influence on the CCK restart. The figure 2 shows the proportion of tasks to re-execute in comparison with the global restart for many checkpoint periods. On the figure 3 is the number of involved processes. For the scenario application run on 8,192 processors, a 10-seconds checkpoint period gives less than 10 % of tasks to re-execute and less than 2,500 involved processes (over 8,192).

Between two checkpoints, the amount of computation and the iteration number increase proportionally with the processor number. When the processor number increases, the proportion of tasks to re-execute and the number of involved processes also increase because the application graph is bigger and holds more dependencies. To preserve the protocol performances, it is required to decrease the checkpoint period when the processor number increases. Moreover, it guarantee that in case of failure, the lost computation will not be too big [3].

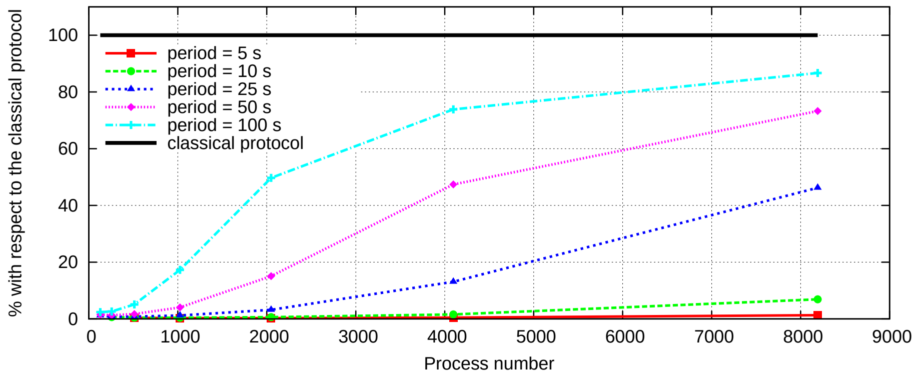

Fig. 2. Proportion of tasks to re-execute with CCK restart with respect to the classical protocol for many checkpoint periods in relation to the process number

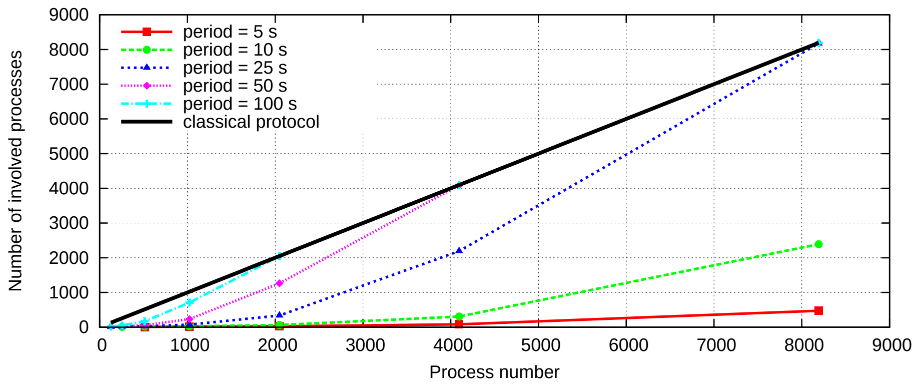

Fig. 3. Number of involved processes with CCK restart with respect to the classical protocol for many checkpoint periods in relation to the process number

5 Conclusion

In this paper we presented the CCK protocol, an improved coordinated checkpoint/rollback protocol for parallel applications. Our work originality comes from the abstract representation provided by the KAAPI library for any applications' parallel execution. 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 communications. We improved the application restart after failure: 1/ the number of processes involved in the restart is smaller; 2/ the restart time for this partial restart is shorter. This work is still in progress, additional evaluations and experiments at grid scale are planned. The final purpose is to provide a framework that adapts dynamically to available resources [11] using the CCK fault-tolerance protocol.

References

1. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3) (2002) 375–408
2. Zheng, G., Shi, L., Kalé, L.V.: 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)
3. Elnozahy, E.N., Plank, J.S.: Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. IEEE Transactions on Dependable and Secure Computing 1(2) (April-June 2004) 97–108
4. Jafar, S., Krings, A.W., Gautier, T., Roch, J.L.: Theft-induced checkpointing for reconfigurable dataflow applications. In IEEE, ed.: IEEE Electro/Information Technology Conference, (EIT 2005), Lincoln, Nebraska (May 2005) This paper received the EIT'05 Best Paper Award.
5. Bouteiller, A., Lemarinier, P., Krawezik, G., Cappello, F.: Coordinated checkpoint versus message log for fault tolerant mpi. In: Proceedings of The 2003 IEEE International Conference on Cluster Computing, Honk Hong,China (2003)
6. Jafar, S., Krings, A., Gautier, T.: Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing. IEEE Transactions on Dependable and Secure Computing (TDSC), in print (2008)
7. Xie, M., Dai, Y.S., Poh, K.L.: Reliability of Grid Computing Systems. In: Computing System Reliability. Springer US (2004) 179–205
8. Neokleous, K., Dikaiakos, M., Fragopoulou, P., Markatos, E.: Grid reliability: A study of failures on the egee infrastructure. In Gorlatch, S., Bubak, M., Priol, T., eds.: Proceedings of the CoreGRID Integration Workshop 2006. (October 2006) 165–176
9. Anstreicher, K.M., Brixius, N.W., Goux, J.P., Linderoth, J.: Solving large quadratic assignment problems on computational grids. Technical report, Iowa City, Iowa 52242 (2000)
10. Wang, Y.M., Huang, Y., Vo, K.P., Chung, P.Y., Kintala, C.: Checkpointing and its applications. Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on (27-30 Jun 1995) 22–31
11. Jafar, S., Pigeon, L., Gautier, T., Roch, J.L.: Self-adaptation of parallel applications in heterogeneous and dynamic architectures. In IEEE, ed.: ICTTA'06 IEEE Conference on Information and Communication Technologies: from Theory to Applications, Damascus, Syria (April 2006) 3347–3352
12. Avizienis, A.: Fault-tolerant systems. IEEE Trans. Computers 25(12) (1976) 1304–1312
13. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fédak, G., Germain, C., Hérault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Néri, V., Selikhov, A.: Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In: SuperComputing, Baltimore, USA (2002)
14. Jafar, S., Gautier, T., Krings, A.W., Roch, J.L.: A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In: Euro-Par. (2005) 675–684
15. Baude, F., Caromel, D., Delbé, C., Henrio, L.: A hybrid message logging-cic protocol for constrained checkpointability. In: Euro-Par. (2005) 644–653
16. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO '07:

Proceedings of the 2007 international workshop on Parallel symbolic computation. (2007) 15–23

17. Kal, L., Skeel, R., Bhandarkar, M., Brunner, R., Gursoy, A., Krawetz, N., Phillips, J., Shinozaki, A., Varadarajan, K., Schulten, K.: Namd2: greater scalability for parallel molecular dynamics. J. Comput. Phys. 151(1) (1999) 283–312
18. Revire, R., Zara, F., Gautier, T.: Efficient and easy parallel implementation of large numerical simulation. In Springer, ed.: Proceedings of ParSim03 of EuroPVM/MPI03, Venice, Italy (2003) 663–666
19. Wiesmann, M., Pedone, F., Schiper, A.: A systematic classification of replicated database protocols based on atomic broadcast. In: Proceedings of the 3rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal (1999)
20. Alvisi, L., Marzullo, K.: Message logging: Pessimistic, optimistic, causal, and optimal. IEEE Transactions on Software Engineering 24(2) (1998) 149–159
21. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1) (1985) 63–75
22. Randell, B.: System structure for software fault tolerance. In: Proceedings of the international conference on Reliable software. (1975) 437–449
23. Baldoni, R.: A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In: Proc. of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97), IEEE Computer Society (1997) 68
24. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent Checkpointing under Unix. In: Proceedings of USENIX Winter1995 Technical Conference, New Orleans, Louisiana/U.S.A. (January 1995) 213–224
25. Galilée, F., Roch, J.L., Cavalheiro, G., Doreille, M.: Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, ed.: Pact'98, Paris, France (October 1998) 88–95
26. Roch, J.L., Gautier, T., Revire, R.: Athapascan: Api for asynchronous parallel programming. Technical Report RT-0276, Projet APACHE, INRIA (February 2003)
27. Pellegrini, F., Roman, J.: Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical Report 1038-96, LaBRI, Université Bordeaux I (1996)
28. Karypis, G., Aggarwal, R., Kumar, V., Shekhar, S.: Multilevel hypergraph partitioning: Application in vlsi domain. In: DAC '97: Proceedings of the 34th annual conference on Design automation, New York, NY, USA, ACM Press (1997) 526–529
29. Koo, R., Toueg, S.: Checkpointing and rollback-recovery for distributed systems. IEEE Trans. Softw. Eng. 13(1) (1987) 23–31
30. Besseron, X., Jafar, S., Gautier, T., Roch, J.L.: Cck: An improved coordinated checkpoint/rollback protocol for dataflow applications in kaapi. In IEEE, ed.: ICTTA'06 IEEE Conference on Information and Communication Technologies: from Theory to Applications, Damascus, Syria (April 2006) 3353–3358
31. Besseron, X., Pigeon, L., Gautier, T., Jafar, S.: Un protocole de sauvegarde / reprise coordonné pour les applications à flot de données reconfigurables. Technique et Science Informatiques - numéro spécial RenPar'17 27 (2008)
32. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press (September 2001)