A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing[⋆]

Samir JAFAR[†], Thierry GAUTIER[†], Axel KRINGS[‡], Jean-Louis ROCH[†]

†Laboratoire ID - IMAG, Pre-project MOAIS (CNRS-INRIA, INPG-UJF)
51, Avenue Jean Kuntzmann, 38330 Montbonnot St. Martin, France
‡ Computer Science Dept., University of Idaho, Moscow, ID 83844-1010, USA
{Samir.Jafar,Thierry.Gautier,Jean-Louis.Roch}@imag.fr,
krings@cs.uidaho.edu

Abstract. This paper presents a new checkpoint/recovery method for dataflow computations using work-stealing in heterogeneous environments as found in grid or cluster computing. Basing the state of the computation on a dynamic macro dataflow graph, it is shown that the mechanisms provide effective checkpointing for multithreaded applications in heterogeneous environments. Two methods, *Systematic Event Logging* and *Theft-Induced Checkpointing*, are presented that are efficient and extremely flexible under the system-state model, allowing for recovery on different platforms under different number of processors. A formal analysis of the overhead induced by both methods is presented, followed by an experimental evaluation in a large cluster. It is shown that both methods have very small overhead and that trade-offs between checkpointing and recovery cost can be controlled.

1 Introduction and Background

Grid and cluster architectures are gaining in popularity for scientific computing applications. The distributed computations, as well as their underlying infrastructure consisting of a large number of computers, storage and networking devices, pose 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.

Fault-tolerance is an effective method to address the possibility of faults in large systems. This is especially important in the case of grids and clusters since in the absence of fault-tolerance the probability of failure, and thus the unreliability of such architecture, increase with the number of components that can fail [21]. Recovery from faults imply the existence of redundancy, e.g. time, information, or spatial redundancy. In the case of large heterogeneous environments, redundancy mechanisms must address the specific requirements associated with recovery mechanisms, taking into account a dynamic number of possibly dissimilar computational nodes.

Many possible solutions based on fault-tolerance have been studied in the literature. Approaches based on duplication [15] can only tolerate a fixed number of faults.

[⋆] This work is supported by the CNRS, ACI-GRID DOC-G and Région Rhône-Alpes project RAGTIME.

All other protocols are based on saving the state of the processes and on constructing a consistent global state [11], i.e. log-based and checkpoint-based protocols [5]. 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, e.g. node architecture or operating system. In the contrary, one speaks of homogeneity. The last 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.

We focus on roll-back strategies under consideration of crash faults and present two major mechanisms: *log-based* and *checkpoint-based* rollback-recovery.

1.1 Log-based protocols

Message logging [12] is based on the fact that a process can be modelled by a sequence of interval states, each one representing a non-deterministic event [16]. Under the hypothesis that each non-deterministic event can be identified, their logging allows a crashed process to be recovered by (1) restoring it to the initial state and (2) replaying messages to it in the same order they were delivered before the crash. To avoid a roll-back to the initial state of a process and to limit the amount of messages that need to be replayed, each process periodically saves its local state. Examples of systems based on this method include MPICH-V2 [4], and FTL-Charm++ [18]. For applications with extensive inter-process communication, log-based protocols are burdened with the possibly large overhead, with respect to space and time, induced by the logging of messages.

1.2 Checkpoint-based protocols

Checkpointing methods are based on periodically saving a global state [11] 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 all processes for building a consistent global state before writing the checkpoints to stable storage. The disadvantage of coordinated checkpointing is the large latency due to coordination in order to achieve a consistent checkpoint. Its advantage is the simplified recovery without roll-back propagation and minimal storage overhead, since there is only one checkpoint per process. This protocol is included in [6,19].

Uncoordinated checkpointing assumes that each process independently saves its state and a consistent global state is achieved in the recovery phase [5]. 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 possibly large number of forced checkpoints and the overhead associated with storing them.

There are only few approached supporting portability, multi-threading, local recovery and cost models [4,13,20]. However, portability of existing checkpointing tools is achieved by using portable languages like Java or by re-compilation to support heterogeneity [20], but not by the checkpointing mechanism itself.

2 Dataflow Work-Stealing for Grid Computations

Dataflow graphs [9] allow for a natural representation of a parallel execution, and they can be exploited to achieve fault-tolerance [1]. At runtime, ready-to-execute instructions are executed depending on the availability of data. Formally, a dataflow graph is 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 dataflow graph, generated at runtime, as described in [7].

We adopt an efficient online scheduling algorithm called work-stealing [14]. The principle is simple, when a processor becomes idle it tries to *steal* work from other processors. In Cilk [14], a theoretical upper bound on the makespan is given for the case of *multithreaded computation*. This result was extended in [7] to our dynamic dataflow graph, and in [3] to consider heterogeneous systems.

2.1 Dataflow and Work-Stealing in KAAPI

The Kernel for Adaptive, Asynchronous Parallel Interface (KAAPI) used in this research is a C++ library that allows to program and execute multithreaded computations with dataflow synchronization between threads. The library is able to schedule programs at fine or medium granularity in a distributed environment.

In the KAAPI execution model a multi-processor system is viewed as a collection of so-called *K*-processors, which can be thought of as kernel threads. A process may consist of several *K*-processors. A *K*-processor in turn executes so-called *K*-threads, which can be thought of as application-level user threads. On a *K*-processor only one *K*-thread is active at a given time. The thread of control is a sequence of non-interruptible tasks. A *K*-processor becomes idle if there are no ready-tasks, i.e. either all tasks have finished execution or they are waiting for data as the result of synchronization. Under the work-stealing strategy, an idle *K*-processor tries to steal a task of a *K*-thread from a randomly selected *K*-processor called *victim*.

2.2 KAAPI Model Analysis

The KAAPI cost model will be the frame of reference for the analysis in Section 4. The time of a sequential execution of a program is denoted by T_1. It is the total time to execute all the operations in the computation on a single processor, with no scheduling overhead. Furthermore, let T_∞ be the execution time of the application as executed on an unbounded number of processors. Thus T_∞ represents the execution time associated with the critical-path.

For the execution of a KAAPI program on p identical physical processors, the corresponding execution time T_p is affected by T_1, T_∞ as well as the overhead associated with loading and managing the data-structures and scheduling using work stealing. We will adopt the simplified model of Cilk-5 [14], which utilizes Graham's bound [8], and is also valid for KAAPI. Then, T_p is bounded by (see Equation 2 in [14]):

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + c_\infty T_\infty \tag{1}$$

The constant c_∞ defines a bound on the overhead associated with the critical-path. In the remainder of the text, we assume that each physical processor executes only one *K*-processor.

3 Checkpoint/Recovery Model

Before describing the two fault-tolerance mechanisms we have integrated into KAAPI we need to define the *state of an execution*. This definition is perhaps the most important difference between this work and the related works (see Section 1) and is the basis for allowing checkpointing in a heterogeneous environment with the flexibility of recovery on any type or number of processors.

3.1 Definition of Execution State

We use a macro dataflow graph to define the state of the application's execution. The graph is a representation of the computational tasks to be carried out along with the associated data, which constitute the inputs and outputs. The dataflow is dynamic, changing during execution of the program, e.g. at the invocation of a task, and it is platform-independent. As a result the graph or portions of it can be moved across platforms during execution. Formally, at any instance of time t, 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 virtual dataflow graph, its implementation is in fact distributed. Specifically, each process i contains and executes a subgraph G_i of G. Within this representation lies the flexibility of restarting individual processes: in the case of a single fault, one does not have to perform a global roll-back. 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. Note that this also includes the information associated with dependencies between G_i and G_j, $i \neq j$.

The instant of time at which a checkpoint can be taken is either before or after the execution of an application task. The checkpoint itself is a snapshot of G, which consists of tasks, specifically their function IDs, and their associated inputs. It does not consist of the task execution context itself. Understanding this difference between the two concepts is crucial. *Checkpointing a task* and its inputs simply requires to store the task's function ID and its input data. *Checkpointing the execution of a task* usually consists of storing the context of the processor, i.e. processor registers (such as program counters and stack pointers) and 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.

The checkpointed macro dataflow graph contains only the future of the execution, i.e. the tasks to be carried out and the necessary data. Certain temporary data associated with the execution of the task are not necessary to the future of the execution and are not checkpointed. The result is a reduction of the checkpoint size.

3.2 Systematic Event Logging

Systematic Event Logging (SEL) is derived from a log-based method [12]. Only the state-change events, i.e. additions and deletions of nodes in the macro dataflow graph, are logged. A recovery consists of simply loading and rebuilding subgraph G_i associated with the failed process i from the respective checkpoint file. The advantage of this approach is that during recovery it allows the re-execution of single tasks, which is interesting for applications requiring the certification of computations and results [10].

In the implementation of SEL, the events that trigger the change of the state of the macro dataflow graph are either the creation or deletion of tasks or the data dependencies they produce. Recall that tasks and data dependencies constitute the two principal components of the graph. These events, together with a uniquely assigned identifier allowing their association with the node in the graph, are stored in stable storage.

3.3 Theft-Induced Checkpointing

Theft-induced checkpointing (TIC) is based on the method presented in [2]. The creation of checkpoints can be initiated (1) at specific checkpointing periods or (2) by the theft of a task. In the first case, checkpoints of the macro dataflow graph G, i.e. G_i on process i, are stored periodically[1] at pre-defined periods τ. In the second case, the state of the macro dataflow graph is checkpointed as the result of communication between processes. In the presence of work stealing, each theft will cause such communication, thus resulting in a so-called *forced* checkpoint. The communication due to work stealing accounts for the only communication of the application. The checkpoint is generated at the time of a theft operation. A recovery consists of loading G_i from the checkpoint file related to the crashed process.

Recall that in KAAPI a process executes on a collection of *K*-processors, which in turn execute a certain number of *K*-threads. In the implementation of TIC in KAAPI the checkpoint of a process is implemented by checkpointing its associated *K*-processor. Each *K*-processor generates incremental checkpoints for each associated *K*-thread. At the expiration of period τ, each process checkpoints its state represented by G_i. In case of a task theft, only the *K*-processor from which the task was stolen forces a checkpoint.

[1] Recall that checkpointing is performed at the task level. This should not be confused with preemptive periodic scheduling, where the context of the preempted tasks are stored.

4 Model Analysis

In the analysis of the overhead associated with SEL and TIC we differentiate between executions without and with faults. Furthermore, we assume that $\frac{T_1}{p} \gg c_\infty T_\infty$, which will be referred to as the *parallel slackness assumption* [14]. In the presence of work-stealing this leads to a linear speedup of $T_p \approx \frac{T_1}{p}$.

4.1 Analysis of Fault-free Execution

If we add a checkpointing mechanism, it is of special interest to analyze its overhead associated with fault-free execution, since the occurrence of faults is considered to be the rare exception rather than the norm.

Analysis of $\boldsymbol{SEL}$: In SEL a log is initiated for each node created. Thus the overhead associated with logging depends on dataflow graph G. Let T_P^{SEL} denote the execution of a KAAPI program on p processors under consideration of logging overhead. Then,

$$T_P^{SEL} \leq \frac{T_1^{SEL}}{p} + c_\infty T_\infty^{SEL}. \tag{2}$$

T_∞^{SEL} denotes the critical-path under SEL, where $T_\infty \approx T_\infty^{SEL}$. Furthermore, T_1^{SEL} denotes the time of a sequential execution of a program under consideration of the overhead induced by logging, i.e. $T_1^{SEL} = T_1 +$ *logging overhead*. This overhead is a function of two parameters. First, it depends on the size of G. Specifically, it depends on the number of tasks and data dependencies, as well as the size of the latter. Second, it depends on the time of an elementary access to stable storage, denoted by t_s. Therefore,

$$T_1^{SEL} = T_1 + f_{overhead}^{SEL}(|G|, t_s). \tag{3}$$

The real measure of SEL overhead is thus $T_1^{SEL} - T_1$, which in turn allows the derivation of the overhead in the parallel execution, i.e. $T_P^{SEL} - T_p$.

Analysis of $\boldsymbol{TIC}$: In theft-induced checkpointing, a checkpoint is performed periodically for each process, as dictated by period τ, and as the result of task stealing. Let T_P^{TIC} denote the execution of a KAAPI program on p processors under TIC. Thus,

$$T_P^{TIC} \leq T_p + \max_{i=1,\ldots,p}\{CheckpointOverhead_i\}, \tag{4}$$

where $CheckpointOverhead_i$ denotes the total checkpointing overhead on processor i. This overhead depends on the total number of checkpoints taken on processor i and the overhead of a single checkpoint. The maximal number of checkpoints performed by a processor is $[T_P^{TIC}/\tau + N_{theft}]$, where T_P^{TIC}/τ indicates the number of checkpoints due to period τ and N_{theft} is the maximal number of thefts performed by any processor.

The overhead of a single checkpoint in TIC is different from that in SEL, since now the checkpoint constitutes the collection of tasks in G_i, rather than a single task.

The number of tasks in G_i has an upper bound of N_∞, which denotes the maximum number of tasks in a path of G [14]. The checkpoint overhead is thus bound by $\max_{i=1,\dots,p}\{CheckpointOverhead_i\} \leq [T_P^{TIC}/\tau + N_{theft}]\ f_{overhead}^{TIC}(N_\infty, t_s)$. Note that function $f_{overhead}^{TIC}()$, which indicates the overhead associated with a single checkpoint, depends only on G, or more preceisly N_∞ and t_s. Thus, the checkpointing overhead is

$$T_P^{TIC} \leq T_p + [T_P^{TIC}/\tau + N_{theft}]\ f_{overhead}^{TIC}(N_\infty, t_s). \tag{5}$$

Under the parallel slackness assumption, it is important to note that the number of thefts, N_{theft}, resulting in forced checkpoints is bound and small for many applications [14,17]. Then, by selecting an appropriate τ the number of local checkpoints can be adjusted in order to obtain $T_P^{TIC} \approx T_p$.

4.2 Analysis of Executions Containing Faults

The overhead associated with fault-free execution is the penalty one pays for having a recovery mechanism. It remains to be shown how much overhead is associated with recovery as the result of a fault and how much execution time can be lost under different strategies.

The overhead associated with recovery is due to loading and rebuilding G. This can be effectively achieved by loading G_i of the affected processes. The time depends on the size of G_i and is dominated by the size of the data representing the task inputs. Thus, the time of recovery of a single process i, denoted by $t^i_{recovery}$, depends only on the size of its associated subgraph G_i. Therefore, $t^i_{recovery}$ is of the order of the size of the subgraph, i.e. $t^i_{recovery} = O(|G_i|)$. Note that for a global recovery, as the result of the failure of the entire application, this translates to $max(t^i_{recovery})$ and not to $\sum t^i_{recovery}$.

The advantage of SEL is that, due to its fine granularity, the maximum amount of execution time lost is that of a single task. Furthermore, the rollback only requires the recovery overhead of a single task. However, this comes at the cost of higher logging overhead, as was addressed in Equation 3.

For TIC the maximum amount of lost execution time is generally higher than for SEL and is bound by period τ. The recovery overhead depends on the size of the graph that need to be loaded and rebuilt. However, note that by appropriately selecting τ one can exercise control over the recovery overhead. The trade-off between lower checkpointing overhead and slower recovery will need to be determined by the application.

It should be noted that we do not consider the time lost due to fault-detection. Whereas the fault-detection time is an important issue, its impact is not the subject of this research; any detection mechanism may be used.

5 Experimental Results

The performance and overhead of the SEL and TIC mechanisms were experimentally determined for the *Quadratic Assignment Problem* (instance[2] NUGENT 22) which was

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

parallelized in KAAPI. The experiments were conducted on the iCluster2[3]. 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 [6]. This configuration has two advantages. First, it reflects the theoretical assumptions of the previous section and second, the actual overhead of the checkpointing mechanism is measured, rather than the overhead associated with a centralized checkpoint server.

Fig. 1. Impact of threshold of parallelism

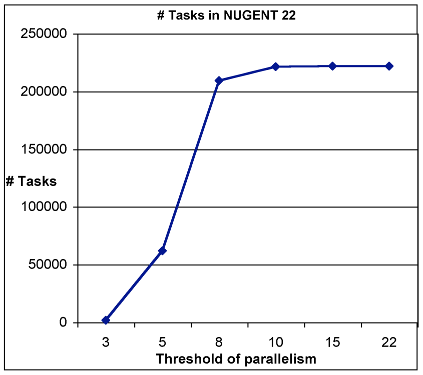
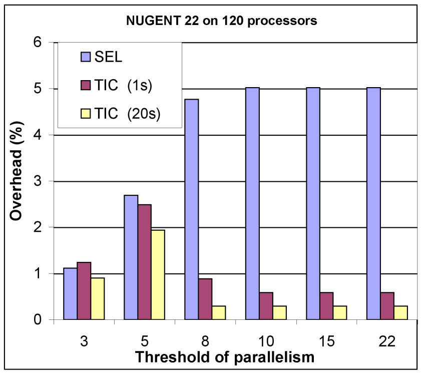


Fig. 2. Checkpoint overhead



The application recursively generates tasks and the degree of parallelism can be adjusted. After a given depth of recursion no more tasks are generated. The sequential execution time without KAAPI was 34,695 seconds. With KAAPI, at fine grain (threshold ≥ 10), the execution on a single processor generated 225,195 tasks and ran in 34,845 seconds. The impact of the degree of parallelism can be seen in Figure 1 and 2. The number of parallel tasks generated for different thresholds of parallelism is shown in Figure 1. The degree of parallelism increases drastically for threshold 5 and approaches its maximum at threshold 10.

The number of tasks has direct implications on the cost of the checkpointing mechanism. Figure 2 shows that the cost of SEL is very susceptible to the total number of tasks, as predicted by Equation (2) and (3), which showed the overhead as a function of the number of tasks.

Figure 2 also shows the impact of parallelism on the overhead of TIC for period τ equal to 1 and 20 seconds. As shown in Equation 5, the overhead is dependent on the critical-path, i.e. T_∞, and N_∞. As parallelism increases, and thus both T_∞ and N_∞ drastically decrease, the checkpointing overhead is reduced. As predicted, the longer period results in lower cost.

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

Figure 3 demonstrates that the checkpoint mechanisms as well as the application are scalable. As the number of processors increase, the different protocols show little change in cost. The impact of the number of faults on the cost of recovery for SEL can

Fig. 3. Scalability of the checkpointing

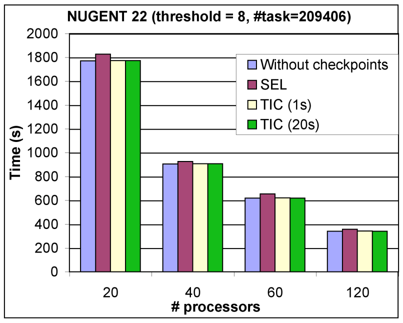
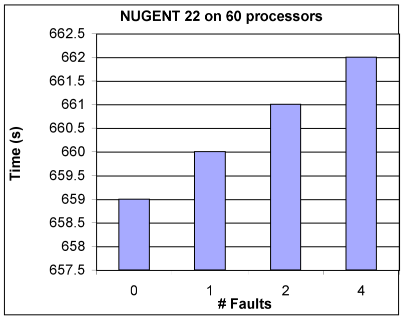


Fig. 4. Recovery cost for SEL



be seen in Figure 4. In fact, the overhead due to restart increases linearly. The recovery times are derived from two measures. The first is the computation time averaging 0.25s per computational task. The second is the overhead due to loading the checkpoint file, averaging 7 MBytes, for rebuilding each G_i. Since single process roll-back was hardly measurable, the experiment shows faults and restarts of all processors.

6 Conclusions

Two portable fault-tolerance mechanisms, systematic event logging and theft-induced checkpointing, have been introduced for heterogeneous multithreaded applications. The flexibility of macro dataflow graphs has been exploited to allow for a platform-independent description of the application state. This description resulted in flexible, portable, recovery strategies. Systematic event logging allowed for rollback at lowest level of granularity, with a maximal computation loss of one task. However, its overhead was sensitive to the size of the application graph, i.e. the number of tasks. Theft-induced checkpointing has lower overhead, related to work-stealing, which was shown bound to the critical-path. The experimental results demonstrated low overhead of both approaches and confirmed the theoretical analysis.

References

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

3. M. Bender and M. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk, 2002.
4. A. Bouteiller, F. Cappello, T. Hérault, P. Lemarinier, G. Krawezik, and F. Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. In *SuperComputing*, Phoenix, USA, 2003.
5. 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.
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. F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *PACT'98*, pages 88–95, Paris, France, October 1998.
8. Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.
9. T. Ungerer J. Silc, B. Robic. *Asynchrony in parallel computing: from dataflow to multithreading*, pages 1–33. Nova Science Publishers, Inc., 2001.
10. S. Jafar, S. Varrette, and J.-L. Roch. Using data-flow analysis for resilience and result checking in peer-to-peer computations. In *IEEE DEXA'2004*, Zaragoza, Spain, August 2004.
11. L. Lamport K. M. Chandy. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
12. K. Marzullo L. Alvisi. Message logging: Pessimistic, optimistic, causal and optimal. *TSE*, 24(2):149–159, 1998. Transactions on Software Engineering.
13. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report CS-TR-97-1346, Univ. Wisconsin, Madison, 1997.
14. K. H. Randall M. Frigo, C. E. Leiserson. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223. ACM Press, 1998.
15. A .Schipper M.Wiesmann, F. Pedonne. A systematic classification of replited database protocols based on atomic broadcast. *In Proceedings of the 3th European Research Seminar on Advances in Distributed Systems(ERSADS99)*, pages 351–360, 1999.
16. S. Yemini R. Strom. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.
17. R. Revire. *Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée*. Thèse de doctorat en informatique, INPG, septembre 2004.
18. L. V. Kale S. Chakravorty. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.
19. G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.
20. V. Strumpen. Compiler technology for portable checkpoints. Technical Report MA-02139, MIT Laboratory for Computer Science, Cambridge, 1998.
21. K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley and Sons, New York, 2001.