Fault-Tolerance for Macro Dataflow Parallel Computations on Grid

Samir JAFAR*and Jean-Louis ROCH

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

Abstract

Large scale cluster and grid computer systems gather thousands of nodes for computing parallel applications. At this scale, component failures or disconnections are normal part of operation, and applications have to deal more directly with repeated failures during program runs. In this paper, we present a portable fault tolerant mechanism for execution of macro dataflow parallel programs on a large scale distributed and heteogeneous grid including SMP nodes. Our mechanism is based on a portable checkpoint-rollback and supports both parallel programs with dependencies and addition or resilience of heterogeneous resources. We have implemented this mechanism on top of Athapascan programming interface and experimental results are presented.

1 Introduction

In recent years, parallel computers, workstation clusters and grid systems have attracted more and more attention for high end applications which demand very high computational performance like multi-physics simulations or complex data analyses. The main advantage of these systems is the scalability from small up to very large systems. However, on a large scale system, node failures or disconnections are frequent events, that have to be considered.

A current trend in parallel computing is to be able to execute parallel applications with communication between tasks on a large scale distributed and heteogeneous grid system including SMP (Symetric Multi Processor) nodes. For distributed and parallel systems, the main sources of failure/disconnection are the network and the nodes. In this work, we consider the failure/disconnection as node volatility : the node is no more reachable and the results computed by this node after the disconnection will not be considered in the case of a later reconnection.

In this paper, we present a portable fault tolerant mechanism for execution of macro dataflow parallel programs on a large scale distributed and heteogeneous grid including SMP nodes. This mechanism is based on a portable checkpoint-rollback and supports both parallel programs with dependencies and addition or resilience of heterogeneous resources.

*This work is performed with the supported by a grant of the Syrian Government in the framework of the France-Syria bilateral agreement.

The second section of the paper presents related works on fault tolerance for parallel computing. Section 3 presents our approach for checkpoint/rollback which is based on the graph of dependency associated to a parallel program. In section 4, we present an implementation of this mechanism on top of Athapascan [8] programming interface and experimental results will be presented.

2 Related work

Global Computing platforms, such as large scale clusters and Grid systems, gather thousands of nodes. Since, node failures or disconnections are frequent events. The study of fault tolerance mechanisms suited to such platforms is an active field of research [5, 9, 6, 11, 13, 7]. Namely two categories of applications are considered:

- **independent jobs**, like in the Condor [17] batch system: the application is a set of independent sequential jobs, each job being a process, which may be dynamically created. The related fault tolerant mechanism [10] then consists in checkpointing each sequential process independently. This approach supports addition and resilience of resources, but is restricted to jobs without dependencies.
- **applications based on message passing**, most often developped with MPI [5] or PVM [9] . The number of processors is fixed at the starting of the execution, and the application consists in a fixed number of communicating processes. In MPICH-V [5], each process is checkpointed independently at a given coordinated checkpoint [7]; to build a consistent global state, all communication performed by any process are logged [1]. Egida [13], provides a toolkit for PVM applications to support fault tolerance; thus toolkit similarly enables the checkpointing of a global consistant state. In both MPICH-V and Egida, the checkpoint/recovery algorithm requires a memory space large enough to store all communication events between two checkpoints. Here, the recovery consists in replacing a failure node by a new node; but addition or resilience of resources is not supported.

Both cases are based on a memory core of each sequential process; also, they do not currently support nei-

ther restart on heterogeneous nodes nor checkpoint of concurrent multithreaded processes within a single sequential process.

Besides, addition of resources is based on a greedy scheduling (such as work stealing [4]), in order to achieve portability on platforms when new resources may become available during execution.

When a new resource is added, a new process is started on this resource. First, at initialization, this process establishes logical connection (e.g. TCP/IP sockets) with the other processes to join the application in the grid computation. Then, the application consists in a logical complete network of processes. Then, the ready process tries to steal computation tasks from the other processes.

Due to the dependencies, the results of the stolen task may be used as input of other tasks. Also, to mange dependencies independently from the execution on the dynamic architecture, the application is represented by a DAG (Diect Asyclic Graph), that describes computational tasks and thier data dependencies. In macro dataflow parallel languages (e.g. Athapascan [8], Cilk [2], Jade, Nesl, Smart, ...), such a DAG is implicitly defiened from the sintaxe of the program.

The macro dataflow graph is used to support addition of resources with the help of a greedy scheduling. In the new section, we propose to also use this macro dataflow graph for a distributed checkpointing mechanism to support fault tolerant execution in case of resource failure.

3 Checkpoint/Restart of the dependency graph

In order to deal with heterogeneous resources, a portable fault tolerant mechanism shall be considered. In Porch [16], such a portable checkpoint is proposed for sequential architectures; it is based on the log of the stack which stores each nested procedure call (identity of the procedure, values of its effective parameters and local variables).

To avoid a large overhead, checkpoint is performed just once a failure is detected. Restart mainly consists in rebuilding the stack from the checkpoint and branching to the last procedure call in progress (labels are added in the source code for this branching).

We propose an extension of this mechanism for a macro dataflow computation on a distributed dynamic architecture.

3.1 Dataflow graph

The parallel program is modelled by a bipartite graph (Fig.1): node sets correspond to tasks and data; an edge between two nodes represents a data dependency (either Read or Write dependency). A task is only executable if all of its input data are available.

Although the graph associated to a program is known only at the end of the execution, it evolves during the execution time. Then, at the time i the state of the graph represents the state of the parallel program at this i. A runtime kernel guarantees the correctness of the graph in a distributed environment.

```
void Factorization_LU( matrix<TYPE> A )
{
  for( int k = 0 ; k < A.row_dim();k++){
   FORK Block_Factorization_LU(A(k,k));

  for( int i = k+1 ; i < A.col_dim();i++)
   FORK Block_Times_Inverse_U(A(i,k),A(k,k));

  for( int j = k+1 ; j < A.row_dim();j++)
   FORK Block_Times_Inverse_L(A(k,j),A(k,k));

  for( i = k+1 ; i < A.col_dim() ; i++ )
   for( j = k+1 ; j < A.row_dim() ; j++ )
    FORK Minus_Times( A(i,j),A(i,k),A(k,j));
   }
 }
```

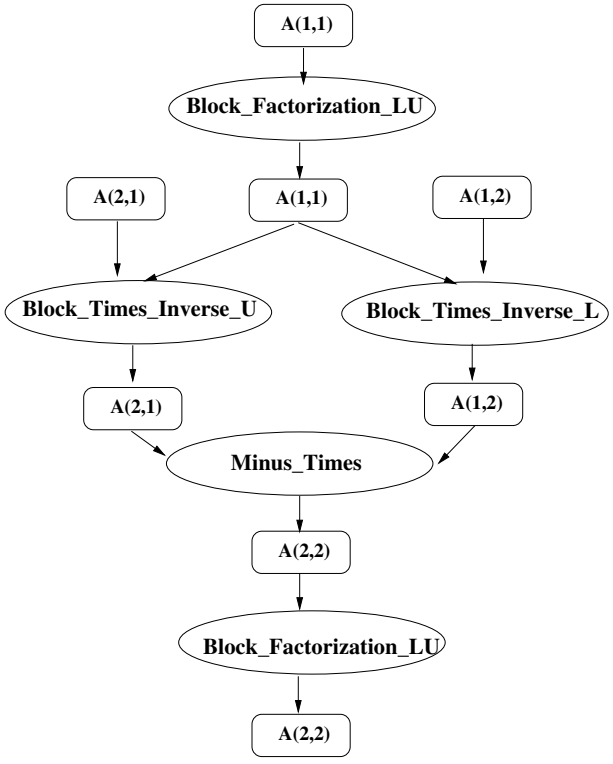

Figure 1: Pseudo-code of an Athapascan macro dataflow program (LU factorization) and a related graph.

As illustration, figure 1 presents a pseudo-parallel code in Athapascan for a Gauss matrix factorization, the macro dataflow graph generated after the execution of the task `Factorization_LU` on a input matrix partitioned on four blocks (`A.row_dim() = A.col_dim() = 2`, `A[i,j]` represents a block of matrix). In the graph, the ellipses represent the tasks executed and the boxes the data handled by the task. The arrows show the data dependencies.

This dependency graph is distributed, dynamic and nested in case of recursive computations; moreover, nodes

that are completed and no more referenced are garbaged. It is -at least implicitly- computed by the runtime kernel in order to ensure the semantics of the computation; e.g. implementations for parallel functional languages for macro dataflow parallel languages rely on cactus stack [15] which can be considered as a coding of this graph. Indeed, the graph may be seen as an extension of the sequential stack considered in [16], the cactus stack represents the state of each stack related to each computation task not yet completed.

We exhibit hypothesis (H1, H2 and H3) on a macro dataflow parallel program that ensure deterministic reexecution. Those hypothesis are verified by most macro dataflow programs; such as Athapascan [8] :

- **H1 :** the synchronizations between the tasks are implicitly defined so as to respect the semantics defined on the accesses to the shared objects;
- **H2 :** a task is carried out until the end of its execution without synchronization. The creation of task is not blocking and a creative task cannot await the results of a task created by it;
- **H3 :** tasks are deterministic; any execution of a task with same input delivers the same result.

Assuming that these hypothesis, the following proposed are proved.

Proposition 1: assuming H1, H2 and H3, the macro dataflow graph describes a consistent global state. Moreover, this global state can be computed in a distributed way locally on each processor with no additional synchronization overhead.

Proposition 2: under these hypothesis H1, H2 and H3, at each time top, the additionnal memory space required to store this global state is bounded by the memory space required by the parallel execution of the program itself.

Moreover, this global state can be computed in a distributed way locally on each processor with no additional synchronization overhead. this compares advantageously to logging all communication events that may require an unbounded memory space.

3.2 Checkpoints

Our fault tolerant mechanism relies on the checkpoint of the macro data flow graph. In order to support multithreaded programming on symmetrical multi-processors (SMP) nodes, this checkpoint is performed at a fine grain. It consists in an asynchronous distributed systematic storage of each task (identifier and parameters) and of the data dependencies (identifier and value of the shared data or object).

Atomic events are registered for each task declaration, start or completion. To ensure scalability, each node in the grid is related to a checkpoint process that uses a stable memory; but two nodes may be related to the same checkpoint process.

Figure 2 presents the main idea of the checkpoint method for a macro dataflow graph, each task is independently checkpointed and its checkpoint is saved on a checkpoints server (SC). This server SC, can be centralized, herarchic or distributed.

On a theoretical point of view, this checkpoint algorithm avoids domino effects (e.g. the program is never restarted from initial state); if the MTBF (Mean Time Between Failure) is larges then the maximal execution τ of a task; τ is lesser then the critical time T_∞ of the program on an infint number og ressources, and this checkpoint algorithm provides provable bounds for memory space (proposition 2).

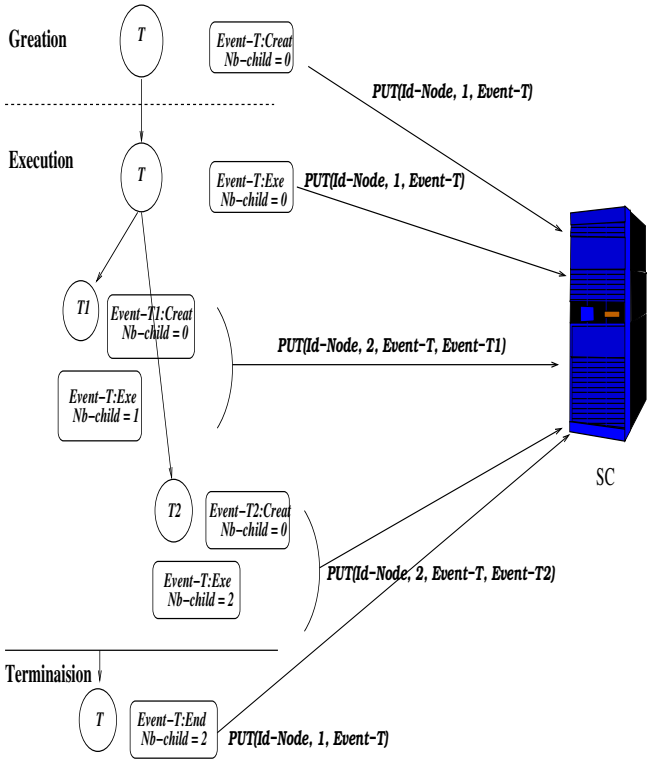

Figure 2: Checkpoint method for a dataflow graph.

The checkpoint algorithm can be presented as follows :

1. **For each shared object do :**
   - before the creation of a shared object `S` :
     (a) assigns an identification to the objet `S`.
     (b) builds the traced event of `S` which contains : identification of the object, the value of the object.
   - at the time of modifying the value of `S`:
     (a) modifies the traced event of `S` (value of `S` = new value).
     (b) saves the traced event of `S` on the checkpoint server (`SC`).
   - When no more reference on the object `S` exists in the system:
     (a) garbages the traced events related to the `S` on the checkpoint server (`SC`).

2. **For each task do :**

   - before the creation of a task `T` :
     (a) assigns an identification to the task `T`.
     (b) builds the traced event of `T` which contains all inputs of `T`.
   - at the time of the creation of `T`:
     (a) modifies the traced event of `T` (status of `T` = creat).
     (b) saves the traced event of `T` on the checkpoint server (`SC`).
   - when the execution of `T` starts :
     (a) if the task `T` is migrated then mark that event at the creator of `T`.
     (b) Modifies the traced event of `T` (status of `T` = start).
     (c) saves the traced event of `T` on the checkpoint server (`SC`).
     (d) at the time of the creation of any children of `T` : modifying the traced event of `T` (number children created = +1).
     (e) saves the traced event of `T` on the checkpoint server (`SC`).
   - when the execution of `T` completes :
     (a) Modifies the traced event of `T` (status of `T` = end).
     (b) saves the traced event of `T` on the checkpoint server (`SC`).
     (c) garbages the events related of `T` on the checkpoint server (`SC`).

3.3 Restart (Recovery)

The recovery algorithm is as follows. In case of failure detection of a node, the stable memory related to the node is marked to be eventually uncompleted. This stable memory contains a set of events related to the dependency graph, i.e. a subgraph. The recovery from this stable memory consists in the rebuilding of this subgraph; tasks that have not yet been completed are reexecuted, while respecting data dependencies. The restart algorithm is the following :

- inisalization to perform a normal execution;
- during this restart execution, all shared objects get the same same identifiers to the shared objects as the execution before failure.
- initializes the value of each shared object created during this restart execution, from the checkpoints server.
- executes the tasks which are not finished in the checkpoints server for a failure node, in the same order that they were created before failure.

Under hypotheses H1, H2 and H3, in the case of a deterministe program, the mechanism of Checkpoint/Rollback proposed verifies the following properties at the time of a recovery :

1. a task is created once and only once;
2. every task finishes correctly its execution once and only once.

In the framework of a parallel program with a dynamic graph of tasks, the algorithm proposed reaches the end, after a limited number of re-executions, given a task `T`, the creation of `T` is executed only once in any execution. At the time algorithm of recovery finishes the execution, any task was effectivly creates once and only once and terminated correctly once.

4 Implementation and evaluation

We have implemented this fault tolerant distributed mechanism on top of Athapascan [14, 8]. Athapascan is developped by the INRIA Apache project [12]; it is a macro-dataflow parallel language (C++ library) dedicated to distributed architectures with SMP nodes.

An Athapascan program describes only computations to be performed and their dependencies: parallelism is described at the grain of a procedure call (`Fork` instruction) that are the tasks; tasks are defined from the access performed by a task on its effective shared parameters (access is typed `Shared_r` – resp. `Shared_w` – for a read – resp. write – access). Figure 1 gives an example of an Athapascan program and the related dependency graph.

In order to register events related to each task and each shared object and to store them in the server of checkpoint related to the node, we have encapsulated Fork and Shared classes into two new classes, "FT::Fork" and "FT::Shared". Indeed, this should enable the use of our fault tolerance mechanism for any Athapascan program.

Figure 3 exhibits experimental results on the *knary* benchmark [3] for recursive tree computations; this experimenal was made on a SMP node with 4 processes and the number of tasks varies between 75 and 1500. S_0 presents the execution time without checkpoints, S_1 shows the execution time with checkpoints. We remark that :

$$\frac{S_1}{S_0} \simeq 1$$

$$\frac{S_1 - S_0}{\#tasks} \simeq 1ms$$

Thus, checkpoint overhaed is constant (1 ms by task).

Figure 4 exhibits experimental results on the *knary* benchmark [3] for recursive tree computations (15000 tasks); the number of nodes varies between 2 and 16 nodes. The failure scenario is as follows: the execution starts on a cluster with 16 nodes (Pentium III, 733 MHZ, 256 MB, 15 GB, Ethernet 100 Mb/s); a failure occurs; recovery is performed from the files of checkpoints,

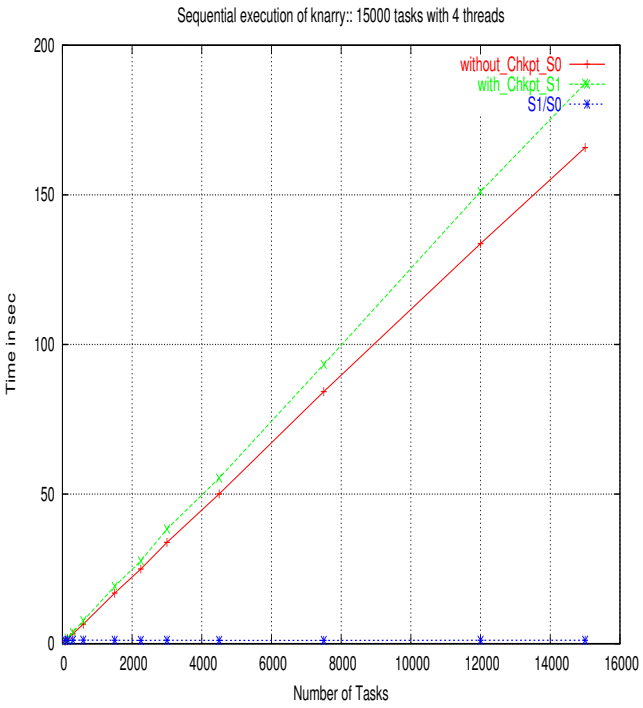

Figure 3: Experimental performances with the application benchmark *knary* : SMP with 4 threads.

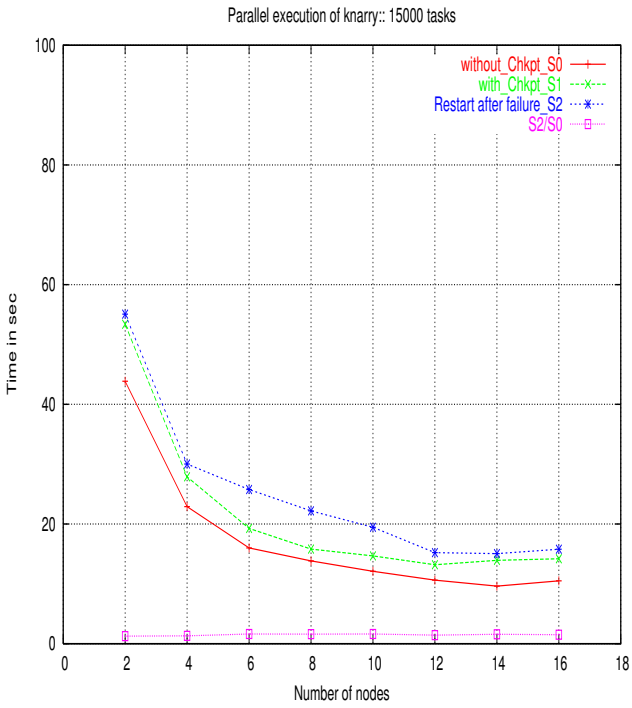

Figure 4: Experimental performances with the application benchmark *knary* : parallel execution on 16 nodes.

S_0 presents the execution time without checkpoints; S_1 shows the execution time with checkpoints; S_2 presents the complete run time with 1-fault execution. We remark that :

$$\frac{S_1}{S_0} \simeq 1$$

$$\frac{S_1 - S_0}{\#tasks} \simeq 1ms$$

$$\frac{S_2}{S_0} \simeq 1$$

Thus, checkpoint overhaed is constant too in this scenario. It can be seen that for tasks of 1ms, overhead is about 10% compared to a normal execution with no fault tolerance; for longer unit tasks (0.1s) the overhead becomes lesser than 1%.

5 Conclusion and perspectives

We have presented a portable mechanism for fault-tolerant macro dataflow computations, which supports both heterogeneous and SMP nodes, both addition and resilience of resources. Our approach is based on the checkpoint/rollback of the dynamic graph of tasks which gives raise to a consistent global state.

A prototype has been developped on top of Athapascan programming interface. Experiments exhibit a small overhead and make the system suited to middle grain applications; within the french ACI DOCG, we are currently adapting this prototype to provide fault-tolerance to a quadratic assignment application developped in Athapascan by Van Dat Cung at PRISM-Versailles.

In the case of fine grain applications, to decrease the overhead of systematic checkpointing of any atomic event, a complementary approach consists in performing coordinate checkpoint of the Athapascan cactus stack, distributed on the nodes. Our future work will compare both approaches.

References

[1] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *Software Engineering*, 24(2):149–159, 1998.

[2] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems Special Issue on SPAA00*, 35:289–304, 2002.

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[4] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *SuperComputing 2002*, Baltimore, USA, 2002.

[6] Maehle E. and Markus F.-J. Fault-tolerant dynamic task scheduling based on dataflow graphs. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, April 1997.

[7] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[8] 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, *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris, France, October 1998.

[9] Leon Juan, Fisher Allan L., and Steenkiste Peter. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Feb 93.

[10] 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.

[11] G. Manimaran and C. Siva Ram Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11), 1998.

[12] INRIA APACHE Project. http://www.inrialpes.fr/apache.html, 2002.

[13] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.

[14] J.-L. Roch, T. Gautier, and R. Revire. Athapascan: Api for asynchronous parallel programming. Technical Report RT-0276, INRIA Rhône-Alpes, projet APACHE, February 2003.

[15] S. Sardesai, D. McLaughlin, and P. Dasgupta. Distributed cactus stacks: Runtime stack-sharing support for distributed parallel programs. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume I, pages 57–65, 1998.

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

[17] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.