A Skeletal-Based Approach for the Development of Fault-Tolerant SPMD Applications

Constantinos Makassikis
SUPELEC - UMI-2958 &
AlGorille INRIA Project Team
France
Email: Constantinos.Makassikis@supelec.fr

Virginie Galtier
SUPELEC - UMI-2958
France
Email: Virginie.Galtier@supelec.fr

Stéphane Vialle
SUPELEC - UMI-2958 &
AlGorille INRIA Project Team
France
Email: Stephane.Vialle@supelec.fr

***Abstract*—Distributing applications over PC clusters to speed-up or size-up the execution is now commonplace. Yet efficiently tolerating faults of these systems is a major issue. To ease the addition of checkpoint-based fault tolerance at the application level, we introduce a *Model for Low-Overhead Tolerance of Faults* (MoLOToF) which is based on structuring applications using *fault-tolerant skeletons*. MoLOToF also encourages collaborations with the programmer and the execution environment. The skeletons are adapted to specific parallelization paradigms and yield what can be called *fault-tolerant algorithmic skeletons*. The application of MoLOToF to the SPMD parallelization paradigm results in our proposed FT-SPMD framework.**

Experiments show that the complexity for developing an application is small and the use of the framework has a small impact on performance. Comparisons with existing system-level checkpoint solutions, namely LAM/MPI and DMTCP, point out that FT-SPMD has a lower runtime overhead while being more robust when a higher level of fault tolerance is required.

***Keywords*-fault tolerance, SPMD, application-level checkpointing, programming skeletons, framework**

I. Introduction

As the need for more computation power arises, several industries have begun to embrace parallelism, and hence, high performance distributed systems such as supercomputers and PC clusters. Such systems enable running existing computation-intensive applications faster. Eventually, as explained per *Gustafson's law* [1], new contributions in computation power are used to solve bigger problems in more or less the same time. Additionally, these contributions come in the form of a multiplication of components — in particular more processing units (processors, cores) — whose reliability rate remains unchanged [2]. As a result, the overall reliability rate of the distributed system decreases and endangers the successful completion of applications. Hence, fault tolerance becomes mandatory.

Moreover, because of the industrial nature of the environment, such applications are often subject to time constraints which had better be respected. Some long applications in financial institutions are run overnight. Their results are then expected by morning in order to decide on the strategy to follow for the day. Consequently, fault tolerance must also be efficient. In other words, fault tolerance must be such as not to slow down significantly the application during failure-free time intervals. Also, in case of failure(s), wasted work should be kept low and restart times should be short.

Failures striking high performance clusters can be of various nature and have various effects on victim applications. We focus on *fail-stop failures* (or crash failures) which halt the application when they occur [3]. Such failures can emanate from software as a result of OS weaknesses, memory leaks in long-running services, device occasional problems while I/O operations are insufficiently checked, etc. They can also emanate from defective hardware at the node level as well as the network/interconnection level: faulty RAM modules or network switches for example.

Fail-stop failures can be dealt with using checkpointing techniques. The strategy consists in periodically saving the state of an application. Following a failure, the most recent saved state is then used to restart the application. Checkpointing can be achieved at various levels ranging from the system level to the application level. The latter level appeared extremely interesting to fulfill our aim of efficiency since we can take better advantage of the semantics of the application. However, working at that level also means adding further burden to the programmer as he has to deal with fault tolerance issues in addition to parallel programming ones. Indeed, endowing an application's source code with checkpoint restart capability requires to interleave checkpoint-restart code with normal application code.

The approach we undertook in order to meet the requirements in terms of facilitating the development of parallel applications and endowing them with efficient fault tolerance is based on the following observations. Many parallel applications use a common parallelization paradigm — often *Master Worker* (MW) or *Single Program Multiple Data* (SPMD) — and algorithms used share a similar structure. Relying on these observations, we introduce a model for fault tolerance named MoLOToF (*Model for Low-Overhead Tolerance of Faults*). In order to guide and facilitate the addition of fault tolerance based on checkpointing, MoLOToF features *fault-tolerant skeletons*. These latter provide for fault tolerance while also pre-defining some flexible structures to ease parallelization work. FT-SPMD is a framework built according to MoLOToF's principles and aims a subset of SPMD applications, namely domain decomposition SPMD application which may involve

various communication schemes such as data circulation and complex boundary data exchanges.

The rest of the paper is organized as follows. Section II discusses related work. Section III presents the foundations of MoLOToF while section IV describes our framework FT-SPMD which is derived from MoLOToF model. The evaluation of FT-SPMD is detailed in section V. Finally, section VI includes our conclusions and perspectives for future work.

II. Related works

Since imposing itself as the *de facto* standard for writing message passing parallel applications, MPI has received lots of attention. In particular, many efforts have been geared towards bringing checkpoint-based fault tolerance to MPI applications.

By providing the highest levels of transparency to the programmer and being independent of applications' semantics, *the system level* has been priviledged. LAM-MPI [4], MPICH-V [5] and Open MPI [6] are some examples of MPI libraries that have been endowed with checkpoint-restart capabilities. They result from the combination of a sequential system-level checkpointer such as BLCR [7] and a rollback recovery protocol specific to the MPI library. The purpose of such protocols is to ensure that individual checkpoints of MPI processes – taken by the sequential checkpointer on each node – remain consistent in spite of communication messages. DMTCP [8] aims at checkpoiting general distributed applications connected by sockets. Therefore, its dependency to a specific MPI library is very low, and it supports MPICH as well as Open MPI applications. All previously cited systems use a blocking rollback recovery protocol. Only MPICH-V does provide implementations for several other protocols. In section V we compare FT-SPMD with LAM/MPI and DMTCP (MPICH-V was not operational on our testbed (Section V-A)).

While solutions at the system-level provide great transparency, they suffer from efficiency and portability issues. These result from checkpoints containing many system and architecture dependent informations. The situation is reversed at the application level where transparency becomes an issue.

C^3 [9] and CPPC [10] rely on compiler technology to automate the transformation of C/Fortran programs into a self-checkpointable and self-restartable versions. Many legacy and recent applications limited to C/Fortran can benefit from this approach. That is not the case of newly written applications which use more complex languages such as C++, and for which providing similar transformations proves to be very difficult. Moreover, the task of writing a parallel application is not facilitated.

Finally, several frameworks targeting SPMD applications have been proposed in the past. We focus on those which provide fault tolerance. Such frameworks aim at facilitating programming and easing fault tolerance. Our framework FT-SPMD falls in this category. DOME [11] was among the first and relied on compiler technology to facilitate checkpointing. PUL-RD [12] was a parallel library which provided a skeletal interface to ease the programming of fault-tolerant parallel applications. However, the skeletal interface was limited to regular decomposition applications with fixed boundary exchanges as defined by a stencil operator. Cactus [13] is another framework which as PUL-RD provides a fixed structure and is limited to regular decomposition applications.

FT-SPMD is closest to PUL-RD and Cactus for they also provide skeletons enclosing the algorithmic structure. However, fault-tolerant skeletons provided by FT-SPMD are more flexible since they allow to implement domain decomposition applications with irregular boundary exchanges. Moreover, FT-SPMD skeletons support the implementation of domain decomposition applications with data circulation (cf. section IV-D). Finally, FT-SPMD emphasizes the importance of collaborations with the programmer and the environment in order to improve the efficiency of fault tolerance. Our MoLOToF model, that serves as a base for FT-SPMD, describes how the programmer can easily achieve efficient fault-tolerance using *fault-tolerant skeletons* and *collaborations*.

III. MoLOToF: a Model for Low-Overhead Tolerance of Faults

MoLOToF is a model for developing parallel fault-tolerant applications. It is made of several principles (or rules) which describe how to easily write such applications. It aims to focus fault tolerance on important parts of an application, contributes to the process of saving and restoring the application, and allows various collaborations with the programmer and the execution environment.

A. Fault-tolerant skeletons concept

MoLOToF consists in **developing applications using structured pieces of code called *fault-tolerant skeletons* (Rule 1)**.

A fault-tolerant skeleton consists of a loop or a succession of loops each of which is endowed with a checkpoint. The remaining body of the loop has computation phases and possibly communication phases. Such loops are termed *fault-tolerant loops*.

An example of the simplest skeletons is given in Fig. 1a and Fig. 1b where a sequential and a parallel skeleton are represented respectively. The parallel skeleton differs by the presence of a communication phase. Such phase enables the cooperation with other processes of the application.

```
1 FT_Seq_Skel
2 {
3   FT_Loop
4   {
5     calculations()
6
7     checkpoint()
8   }
9 }
```

(a) Sequential skeleton.

```
1 FT_Par_Skel
2 {
3   FT_Loop
4   {
5     calculations()
6     communications()
7     checkpoint()
8   }
9 }
```

(b) Parallel skeleton.

Fig. 1: Possible MoLOToF fault-tolerant skeletons.

A parallel application **written according to MoLOToF has each of its processes made of a succession of fault-tolerant skeletons (sequential or parallel) (Rule 2)** as illustrated in Fig. 2.

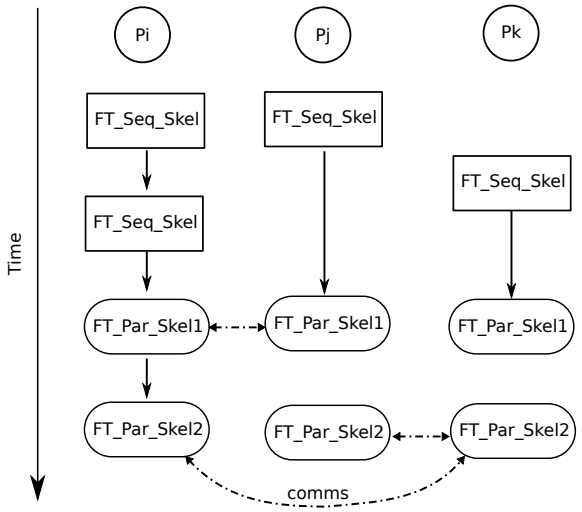


Fig. 2: Example of MoLOToF application.

Naturally, MoLOToF allows operations outside skeletons. **Operations other than those enclosed in computation and communication phases should be *light* (Rule 3)**. In MoLOToF's terminology, this means that their duration is small compared to operations enclosed in computation and communication phases of fault-tolerant skeletons. More importantly, for a given application input, their behaviour is deterministic with respect to the application's control flow. For instance, control flow conditioned by random values is not supported. These rules stem from the checkpoint-restart mechanism and contribute to both correct and fast recovery (cf. Rule 5 in section III-B).

B. Checkpoint-restart mechanism of MoLOToF

Achieving distributed checkpoints (*i.e.*: checkpoints of distributed applications) requires to be able (1) to save and restore the context of a sequential process; and (2) to ensure consistency between each process context using some rollback recovery protocol. In this context, a *recovery line* is made of a set of process states from which the application can recover. Process states often correspond to process checkpoints but may also include the state of some healthy process(es).

MoLOToF considers two execution modes (normal and recovery) according to whether the application recovers from a failure or not (Rule 4).

In *normal mode*, whenever a checkpoint is encountered in the application's code, a test is done to check whether a checkpoint needs to be taken or not. By default, each check is tied to a skeleton and is function of the computation loop iterator within that skeleton. In case a checkpoint needs to be taken, we must ensure consistency with other processes.

In *recovery mode*, failing processes are restarted with the most recent available checkpoint that forms a recovery line with the states of healthy processes. These latter may also be forced to rollback (*i.e.*: restart from a checkpoint).

In order **to recreate the context of the application at the moment of checkpointing, the application is *selectively reexecuted* (Rule 5)**. MoLOToF considers two kinds of reexecutions according to whether it happens within or outside a fault-tolerant skeleton. Instructions outside a skeleton are merely reexecuted as in the initial run. This allows to recover values of read-only variables and to initialize possible more complex data structures. Reexecution within skeletons differs: if it is not the skeleton associated with the checkpoint then it is merely skipped. Otherwise, it is reexecuted until meeting the checkpoint location. However, any computation and communication met before reaching the checkpoint location is skipped. Only light operations are reexecuted on the way. Such behaviour is highly desirable in order to achieve fast restarts as we avoid time-consuming portions of code.

C. Collaborations with programmer and environment

Another characteristic of MoLOToF is that it **requires and encourages various collaborations with the programmer and the environment (Rule 6)**:

a) Collaboration for placement (Rule 6a): this collaboration consists in the determination by the programmer of parts of his code that need fault tolerance. As said earlier, these consist of computation-intensive ones. This is where fault-tolerant skeletons will be placed.

b) Collaboration for correctness and efficiency (Rule 6b): by default, checkpoints contain only some skeleton-related variables such as the loop iterator. As a result, the checkpoint size is minimal but not sufficient to yield correct checkpoints. In order to ensure correctness, the programmer has to add appropriate data. Incrementally adding data contributes in keeping checkpoint size minimal.

c) Collaboration for frequency (Rule 6c): the programmer may adjust the frequency of checkpointing for checkpoints defined inside the loops within skeletons. By default, checkpointing is arbitrarily set to occur at each superstep.

d) Collaboration with the environment (Rule 6d): such collaborations rely on the capacity of being driven. That is, the application has the capacity to receive orders or information from the environment. For instance, the application should be able to take on demand checkpoints. Such request may be issued by the environment in order to migrate the application on other nodes should it detect imminent failures. A system engineer may issue such request before a maintenance routine.

IV. FT-SPMD: A FRAMEWORK FOR FAULT-TOLERANT SPMD APPLICATIONS

Following MoLOToF principles, it is possible to derive programming models which are dedicated to some family of parallel algorithms (*e.g.*: master-worker, domain decomposition SPMD, ...). These latter may be embodied as skeletons of fault-tolerant parallel programs. Such skeletons constitute tools for programming fault-tolerant parallel applications since they allow to setup checkpoint-based fault tolerance. Fault tolerance can also be tuned if necessary and, in our case, we encourage the programmer to do so.

FT-SPMD is a C++-based framework resulting from such a derivation process. Indeed, it comes with a parallelization (or

programming) model for domain decomposition SPMD applications which has been embodied in skeletons. At this level, FT-SPMD provides some additional "tools" which facilitate programming. The set of these tools constitute the FT-SPMD *application software architecture*. In support to the latter, FT-SPMD may be augmented with a *light middleware layer* which enables collaborations with the environment (*e.g.*: on demand checkpointing, ...). This layer may integrate as well its own fault detection and provide for automatic restart. Currently, only the MPI library used by FT-SPMD is part of that layer.

A. SPMD concepts of FT-SPMD framework

This section focuses on non fault-tolerant abstractions proposed by our framework to easily develop domain decomposition SPMD applications.

1) Proposed skeletons: FT-SPMD features two skeletons. In the first one, the number of iterations in the computation loop (or supersteps) is fixed whereas it is not in the second one. In the latter, the termination may depend on initial data as well as results available at runtime. Such condition is common in iterative solvers where the solution is approached through successive iterations. Usually, the computation is stopped when the solution has reached the desired precision.

There exists two skeletons to provide for the aforementioned situations. A simplified representation of the first one is present in Fig. 3. Apart from the difference in definition of the loop, both skeletons share the same structure which consists in a body loop having a calculation phase followed by a communication phase and a swap phase. The swap phase is part of the parallelization model used by FT-SPMD.

```
1  class SPMD_Skel // Non Fault-tolerant skeleton
2  {
3    // Framework for iterator
4    // (internal definition)
5    Skel_for_iter sfi;
6    int it;
7
8    // Double datastructure
9    // (two N-dimension arrays)
10   Domain *V1, *V2;
11
12   void execute()
13   {
14     // Routing plan init
15     Routing_plan *rp =
16                  new Routing_plan(/*...*/);
17     for (it = sfi.beg();
18           it != sfi.end(); it = sfi.next())
19     {
20       compute(sfi);      // Computation phase
21       rp->comms(sfi);    // Communication phase
22       V1->swap(V2);      // Swap datastructures
23     }
24   }
25 };
```

Fig. 3: Non fault-tolerant FT-SPMD skeleton with fixed number of supersteps.

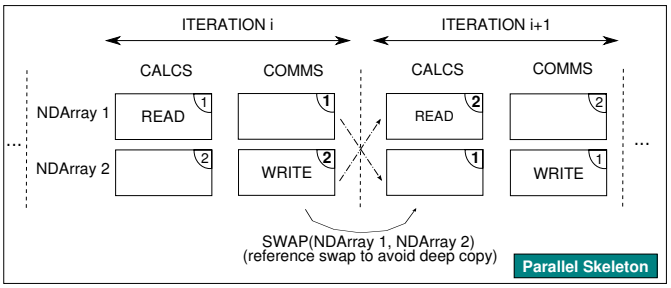


Fig. 4: FT-SPMD parallelization model.

2) Parallelization model of FT-SPMD: As illustrated in Fig. 4, FT-SPMD proposes a parallelization model based on the use of a double data structure consisting in two N-dimensional arrays. N depends on the application which is coded. In this parallelization model, the first array (*NDArray 1*) is used to provide calculation data. The second array (*NDArray 2*) is destined to contain the input data for the calculations of the next superstep. Thus, the first array is always read-only during the calculation phase while the second array is always write-only during the communication phase. This parallelization model may seem really contraintful at first sight. Yet it allows to implement equally well applications involving circulation of initial data (as in some linear algebra algorithms) and applications with boundary exchanges (as in some relaxation algorithms using domain decomposition). The strong point is that all applications using FT-SPMD will benefit almost automatically from an efficient fault tolerance.

3) Calculation kernel, distributed data structure and routing plan: *Calculation kernel* and *routing plan* (cf. Fig. 5) concepts allow to define the calculation and communication phases of the FT-SPMD skeletons. These concepts are linked by the *distributed data structure*.

- The *calculation kernel* is an entry point of the skeleton which allows the programmer to initialize the calculation loop and to provide the calculation function. The latter may host any kind of optimized code the programmer may want to use.
- The *distributed data structure* is a classic concept in parallel programming. In FT-SPMD, it consists of an interface to N-dimensional arrays which leaves the programmer the freedom to choose the underlying implementation. This data structure is important in FT-SPMD since the programmer defines therein the way data is distributed among application processes. Each process knows what data it possesses and what data it needs at any given superstep.
- The pair of *partition functions* defined within the distributed data structure provide information related to *data possessed* and *data needed* by each process at each superstep of the distributed computation.
- The *routing plan* establishes the set of communications to achieve during the communication phase at each superstep. The pair of partition functions allows the plan

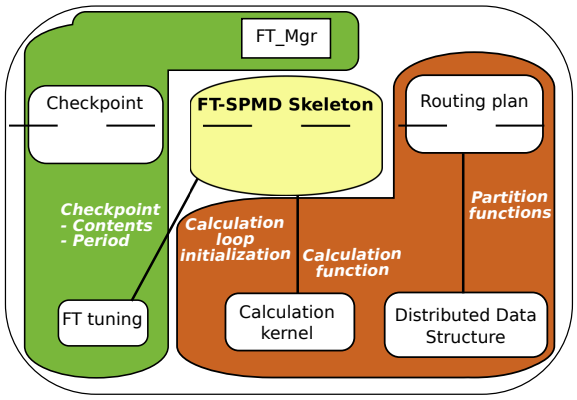


Fig. 5: FT-SPMD concepts organization.

to determine, for each application process, the data it is expecting to receive and the data other processes expect it to send. As demonstrated in [14], [15], the routing plan is a useful concept which easily allows to use alternate predefined schemes to schedule communications so as to avoid overloading the network.

B. Fault tolerance concepts of FT-SPMD framework

Besides SPMD concepts, the FT-SPMD architecture includes concepts related to fault tolerance:

- *checkpoint concept* is an abstraction for checkpoints. This abstraction achieves I/O for sequential checkpoints and provides an interface to control checkpoints. Notably, it allows the programmer to establish the *checkpointing frequency* but also to choose the *data to include in the checkpoint*. The frequency is function of the number of iterations of the computation loop.
- *fault tolerance manager* (`FT_Mgr` in Fig. 5) maintains informations related to fault tolerance such as the location where to store checkpoints. It also keeps track of the application status: whether on recovery or normal execution. This component is also responsible for determining the most recent *recovery line* (cf. section III-B).

Fault-tolerant skeletons are obtained by defining checkpoint locations within the computation loop using the checkpoint abstraction provided by FT-SPMD. Fig. 6 represents the FT-SPMD fault-tolerant version of skeleton with fixed number of supersteps. The skeleton has a declaration of a checkpoint (l. 7). The latter is placed after the `swap` call (l. 23). Furthermore, functions `compute` and `comms` are replaced by their fault-tolerant equivalents ones `ft_compute` and `ft_comms`. These latter are wrapper functions which avoid executing any computations and communications during recovery in accordance with MoLOToF's selective reexecution principle (cf. Rule 5 in Section III-B).

C. Rollback recovery protocol of FT-SPMD framework

Checkpoints in FT-SPMD are taken by all processes at the same point in execution: at the same superstep within a fault-tolerant skeleton. The routing plan guarantees the absence

```
1  class FT_SPMD_Skel // Fault-tolerant skeleton
2  {
3    // Framework for iterator
4    // (internal definition)
5    Skel_for_iter sfi;
6    int it;
7    Checkpoint c;
8
9    // Double datastructure
10   // (two N-dimension arrays)
11   Domain *V1, *V2;
12
13   void execute()
14   {
15     // Routing plan init
16     Routing_plan *rp =
17                  new Routing_plan(/*...*/);
18     for (it = sfi.beg();
19           it != sfi.end(); it = sfi.next())
20     {
21       ft_compute(sfi);    // Computation phase
22       rp->ft_comms(sfi); // Communication phase
23       V1->swap(V2);       // Swap datastructures
24       c.run(it);          // Possible checkpoint
25     }
26   }
27 };
```

Fig. 6: Fault-tolerant FT-SPMD skeleton with fixed number of supersteps.

of communications during checkpointing and thus ensures individual checkpoints form a recovery line. Upon a failure, the whole application is terminated and is restarted from the most recent recovery line. Since several recovery lines may be kept, a simple consensus protocol is used to determine the most recent one.

D. Matmult: an example of parallel product of dense matrices

In the following we focus on a parallel matrix multiplication algorithm which computes C, the product of initial matrices A and B ($C = A.B$).

1) Algorithm description: As illustrated in Fig. 7, matrices A and B are distributed statically among the P processes of the parallel computation. Initially, each process is entrusted with a block of N/P lines of A and a block of N/P columns of B. At the end of a superstep, each processor sends his block of N/P lines of A to his left neighbour and receives a new block of N/P lines from his right neighbour. After N supersteps, each process possesses a block of N/P columns of the result matrix C. In FT-SPMD terminology, data possessed by a process corresponds to its N/P lines. Data needed by a process correpond to the N/P lines of its right neighbour.

2) FT-SPMD implementation: The described algorithm is suited to FT-SPMD parallelization model. At each superstep, each process computes the product of the matrix blocks of A and B it owns. The result is written in the result block. In Fig. 4, the block of matrix A corresponds to *NDArray 1*. During the communication phase, *NDArray 1* is read and sent to the left neighbour while *NDArray 2* is used to write the block sent by the right neighbour. Following the swap phase, a checkpoint

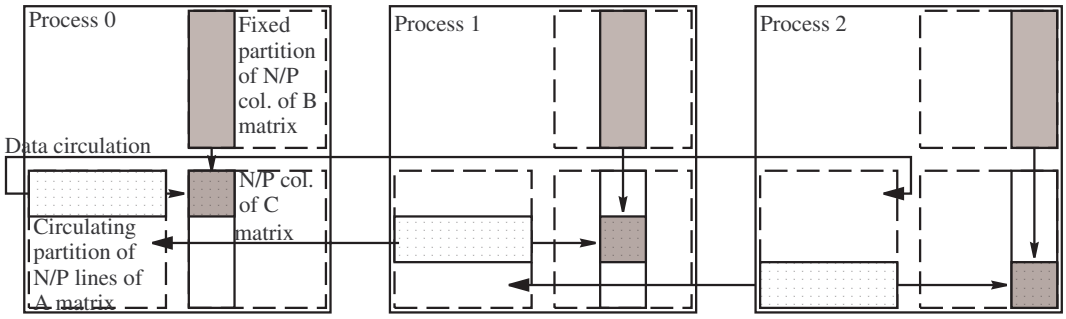


Fig. 7: Data partitioning and circulation for a dense matrix product on 3 processes

may be taken as shown in the fault-tolerant skeleton (cf. Fig. 6, l. 24).

3) Development steps: After the programmer has assertened the compatibility of his application with the parallelization model of FT-SPMD, he has to go through the following steps:

1) *Interface his array with FT-SPMD* (cf. Fig. 8). This step consists in deriving `Matmult_Domain` class from FT-SPMD `Domain` class (l. 3) and implementing some methods. The `Domain` class is the interface to N-dimensional arrays proposed by FT-SPMD. Here the programmer uses arrays from the popular Blitz library [16] (l. 6). Among the methods to define are `data_needed` (l. 20-38) and `data_possessed` (l. 39-41). Partitioning has to be expressed in terms of `Domain_desc`. The latter is a FT-SPMD class describing a `Domain` with a list of intervals (one for each dimension). The programmer needs also to define methods on how to access elements of the array since it is something that varies from one array implementation to another (cf. methods `lget` and `lset` (l. 43-47)). The programmer needs also to provide a method to swap the contents pointed by two `Domain` classes (l. 49-50). Such a method should disappear in future releases.
2) *Write the calculation kernel* (cf. Fig. 9). This step consists in inheriting from FT-SPMD `FT_SPMD_Calc_Kernel` class (l. 1) and defining a `compute` method (l. 30-44) as well as the computation loop iterator of the skeleton.
3) *Write the main function* (cf. Fig. 10). During this step, the programmer initializes and finalizes FT-SPMD using `FT_Mgr::init` (l. 10) and `FT_Mgr::finalize` (l. 47). After initializing FT-SPMD, the programmer instantiates the calculation kernel (l. 12-16) and the skeleton (l. 18-23). Afterwards he achieves a collaboration for correctness and efficiency. Indeed, matrix C needs to be saved in the checkpoint (l. 35-37) and the data structure for writing the data in the next superstep need not (l. 40-42). Finally, the fault-tolerant skeleton can be executed (l. 44).

From the above description of the implementation of the *Matmult* application, we see that the FT-SPMD framework

```
template<typename T_numtype, int N_rank>                                    1
class    Matmult_Domain:                                                    2
    public Domain<double, 2, Matmult_Domain>                                3
{                                                                           4
private:                                                                    5
 blitz::Array<double, 2>    data;                                           6
                                                                            7
public:                                                                     8
 Matmult_Domain(int rank, int numprocs,                                     9
                TinyVector<int, 2> extent):                                 10
   // Call the base class constructor                                       11
   // for proper initialization.                                            12
   Domain<double, 2, ::Matmult_Domain>(rank, numprocs,                      13
                                       extent)                              14
 {                                                                          15
   Domain_desc<2> dd = data_needed(rank, numprocs, 0);                      16
   data.resize(dd.extent(1), dd.extent(2));                                 17
 }                                                                          18
                                                                            19
 Domain_desc<2> data_needed(int rank, int numprocs,                         20
                            int step)                                       21
 {                                                                          22
   int size = this->get_extent(blitz::firstDim);                            23
   int partition_size = size / numprocs;                                    24
                                                                            25
   int dim1_lbound, dim1_rbound;                                            26
                                                                            27
   // Compute boundaries                                                    28
   ((dim1_lbound = (rank + step) * partition_size) == size)?                29
     dim1_lbound = 0, dim1_rbound = partition_size - 1:                     30
     dim1_rbound = dim1_lbound + partition_size - 1;                        31
                                                                            32
   Domain_desc<2>  domain_desc;                                             33
     domain_desc.set_bounds(1, dim1_lbound, dim1_rbound);                   34
     domain_desc.set_bounds(2, 0, size-1);                                  35
                                                                            36
   return domain_desc; }                                                    37
                                                                            38
 Domain_desc<2> data_possessed(int rank, int numprocs,                      39
                               int step)                                    40
 { return data_needed(rank, numprocs, step); }                              41
                                                                            42
 double lget(blitz::TinyVector<int, 2>  &coord)                             43
 { return data(coord(0), coord(1)); }                                       44
                                                                            45
 void lset(blitz::TinyVector<int, 2> &coord, double e)                      46
 { data(coord(0), coord(1)) = e; }                                          47
                                                                            48
 void swap(Matmult_Domain<double, 2>    *md)                                49
 { blitz::cycleArrays(this->data, md->get_data()); }                        50
};                                                                          51
```

Fig. 8: *Matmult* distributed data structure definition.

and the MoLOToF model guide a lot the programmer so that he correctly positions checkpoint locations and defines checkpoint contents. We believe such guidance to be advantageous for the programmer, despite the fact that it does not prevent him from making errors. We expect errors will result most of the time from incomplete checkpoint contents and more rarely from non-obvious incompatibility of the application's algorithm with FT-SPMD's skeletons.

```
class   Matmult_Kernel: public FT_SPMD_Calc_Kernel
{
 // Domain definition.
 Matmult_Domain<double, 2>   A1, // Calc. Read buffer
                             A2; // Comm. Write buffer

 Array<double, 2>  TB, // Fixed local block of Transposed
                       // matrix B.
                   C;  // Fixed local block of result
                       // matrix C.

 // Constructor.
 Matmult_Kernel(int myid, int numprocs, TinyVector<int, 2>
                                                    extent):
      myid(myid),
      numprocs(numprocs),
      A1(myid, numprocs, extent),
      A2(myid, numprocs, extent),
      size(extent(0)),
      local_size(extent(0)/numprocs),
      TB(local_size, size),
      C(size, local_size)
 {
   // Private member method which
   // initializes A1, A2, TB and C.
   LocalMatrixInit();
 }

 //  Calculation method.
 void compute()
 {
   int   i, j, k;
   int   OffsetLigneC;

   // At step "step", the processor compute the C block
   // starting at line: ((myid+step)*local_size)%size
   OffsetLigneC =
        ((myid + A1.get_step()) * local_size) % size;
   for (i = 0; i < local_size; ++i)
     for (j = 0; j < local_size; ++j)
          for (k = 0; k < size; ++k)
               C(i + OffsetLigneC, j)
                    += A1.get(i, k) * TB(j, k);
  }
};
```

Fig. 9: *Matmult* calculation kernel definition.

V. Evaluation of FT-SPMD

We evaluate FT-SPMD by comparing it to LAM/MPI and DMTCP. In order to do so, we consider two implementations of *Matmult*. The first one (*Matmult_v1*) does not use FT-SPMD. The second one (*Matmult_v2*) — described in section IV-D — does use FT-SPMD and is actually written based on the first one. Though it can be considered as a toy application, it has the advantage of allowing to easily change the problem size while preserving an interesting amount of computations. Checkpoints taken for each application process are stored locally (*i.e.*: on nodes' local hard disk).

```
int main(int argc, char **argv)
{
 // Initializations ---------------------- --//

 //   + MPI related initializations.
 MPI_Init(&argc, &argv)
 // ...

 //   + Init. of FT-SPMD's fault tolerance manager.
 FT_Mgr::init(&argc, &argv);

 //   + Init. of 'skeleton input'
 TinyVector<int, 2> extent(size, size); // Extents of each
                                        // dimension of the
                                        // matrices
 Matmult_Kernel<double, 2, Matmult_Domain> mk(extent);

 //   + Init. of skeleton using 'skeleton input'
 FT_SPMD_skel<double, 2, Matmult_Domain>
      Matmult_FT_SPMD_Skel(&mk,
                           &mk.A1,   // Calc. read buffer
                           &mk.A2,   // Comm. write buffer
                           checkpoint_period);

 // Some fault tolerance fine-tuning ------------ //

 //   + Chekpoint correctness: add result matrix to
 //     checkpoint
 //       + C->dataFirst(): address to the first element
 //                         of result datastructure
 //       + C->numElems(): number of elements of result
 //                        datastructure
 //       + PRECONDITION: elements must be contiguous
 //                       in memory.
 Array<double, 2> *C = mk.get_C();
 Matmult_FT_SPMD_Skel.do_register_var(C->dataFirst(),
                                      C->numElems());

 //   + Checkpoint size optimization: unregister the write
 //     buffer from checkpoint.
 Matmult_FT_SPMD_Skel.do_unregister_var(WRITE_BUFFER);

 // Fault-tolerant skeleton execution ---------- --//
 Matmult_FT_SPMD_Skel.execute();

 // Clean up of FT-SPMD ---------------------- --//
 FT_Mgr::finalize();

 MPI_Finalize();

} // END OF  main()
```

Fig. 10: *Matmult* main application function.

The experiments we have lead aim at evaluating different aspects of FT-SPMD. These include (1) the easiness of developing applications with FT-SPMD (Section V-B), (2) the benefit from collaboration with the programmer in checkpoint sizes (Section V-C), (3) the runtime overhead of using FT-SPMD (Section V-D), (4) the impact of checkpointing frequency on application runtimes (Section V-E), (5) the recovery time from a failure (Section V-F), and (6) the maximum wasted work achieved given an almost constant overhead (Section V-G).

A. Testbed description

Experiments were lead on the *Intercell* cluster hosted at SUPELEC. *Intercell* features 256 nodes running on 64-bit Fedora Core 8 and inteconnected through a CISCO 6509 Gigabit Ethernet switch. Each node has an Intel Xeon-3075 dual-core processor (*i.e.*: a total of 512 cores) at 2.66 GHz with a FSB at $1333MHz$ and 4 GB of RAM. However, experiments were run with one process per node.

Matmult_v2 application was linked to OpenMPI 1.3.3 for FT-SPMD experiments. *Matmult_v1* was linked to OpenMPI 1.3.3 for experiments with DMTCP $r481$ and to LAM/MPI 7.1.4 for experiments with LAM/MPI. In any case, the g++ 4.1.2 compiler was used with level 3 optimizations.

B. Evaluation of development effort

Table I reports the number of physical and logical lines of application source code as obtained with `Unified CodeCount` tool [17]. Based solely on these numbers, FT-SPMD introduces a $10.7 - 14.3\%$ overhead. The latter is the result of some steps which are inexistant in the source code of *Matmult_v1*: (1) initialization and finalization of FT-SPMD, (2) code to interface programmer's data structure to FT-SPMD's distributed data structure representation, (3) code to interface with fault-tolerant skeleton. However, each of these steps is guided by the framework and corresponds mostly to code with low algorithmic difficulty (cf. Section IV-D3). Furthermore, a computation code may be reused from existing applications with possible but straightforward modifications in notations. We achieved such modifications when writing *Matmult_v2* using FT-SPMD from existing *Matmult_v1* application. Lastly, the number of additional lines per used skeleton is not expected to vary significantly in other applications. Consequently, in more realistic applications, with far more lines of code, the relative overhead is expected to be lower.

TABLE I: Development overhead introduced by FT-SPMD.

Line type	Matmult_v1	Matmult_v2	Absolute	Relative (%)
physical	258	295	+37	+14.3
logical	168	186	+18	+10.7

C. Evaluation of checkpoint size efficiency

One important aspect of MoLOToF and FT-SPMD is that it involves the programmer in order to minimize checkpoint sizes. In this section, we compare the checkpoint size per process obtained by *OMPI FT-SPMD* with programmer optimizations to the size obtained by *OMPI FT-SPMD* but without such optimizations. We also make comparisons to the size obtained by *OMPI DMTCP* and the one obtained by LAM/MPI. Optimizations for FT-SPMD were described in section IV-D3 and Fig. 10.

Fig. 11 displays checkpoint sizes obtained by the aforementioned configurations for input matrices of size 32768×32768. Results are similar for other matrix sizes. As can be seen, *OMPI FT-SPMD* without optimizations, *DMTCP OMPI* and LAM/MPI achieve checkpoint sizes at least 50% higher than *OMPI FT-SPMD* with optimizations. *DMTCP OMPI* and LAM/MPI achieve higher checkpoint sizes since they work at the system-level. Therefore they additionally save system information as part of the checkpoint (including MPI's library buffers). As for *OMPI FT-SPMD* without optimizations, it is a fairly good approximation of what can be obtained by systems such as C^3 or CPPC which use compiler technology. The checkpoint sizes are indeed lower than those obtained by system-level approaches as they do not save system dependent data. Moreover, using dead variable analysis, they bring further improvement. However, to our best knowledge, they cannot automatically provide for optimizations similar to ours.

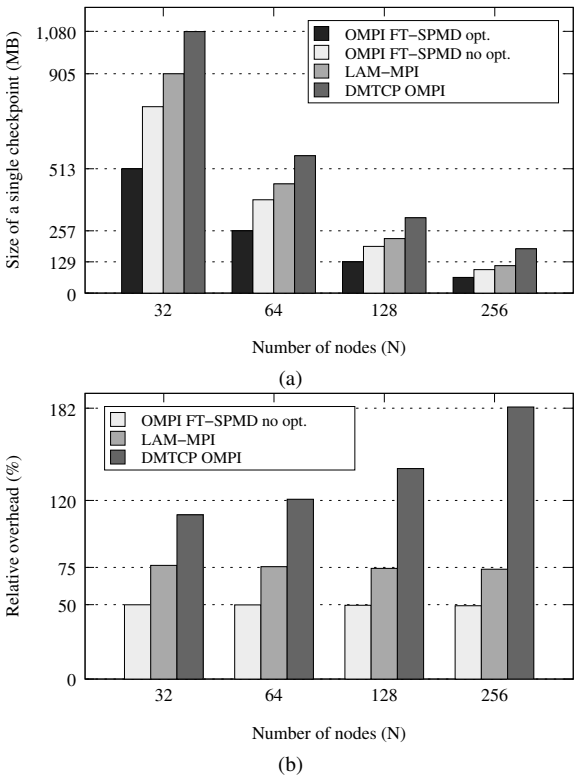


Fig. 11: Comparison of checkpoint sizes obtained by *OMPI FT-SPMD* with optimizations, *OMPI FT-SPMD* without optimizations, *DMTCP OMPI* and LAM/MPI for 32768×32768 input matrices.

D. Runtime overheads comparison in absence of checkpoints

In this third experiment, we determine runtime overheads incurred by *OMPI FT-SPMD*, *DMTCP OMPI* and LAM/MPI compared with OMPI in absence of fault tolerance (and failures) (*i.e.*: no checkpoint is taken). Table II reports the runtime overhead displayed by *OMPI FT-SPMD* compared with *OMPI*. Measurements were achieved for three different sizes of input matrices and different numbers of nodes (one process per node). Results testify to very low overheads ($\leq 3\%$).

As for *DMTCP OMPI*, in our configurations there is no overhead. Finally, for LAM/MPI we observe overheads up to 9.5% compared with OMPI. Some additional experiments showed that the observed gap stemed from longer communication times in LAM/MPI.

E. Impact of checkpointing frequency

In this fourth experiment, *Matmult* was run with *OMPI FT-SPMD*, *DMTCP OMPI* and LAM/MPI under different input matrix sizes, different number of nodes and an increasing number of checkpoints (*i.e.*: increasing checkpointing frequency).

TABLE II: *OMPI FT-SPMD*'s runtime overhead with *Matmult_v2* compared with *Matmult_v1* and OMPI.

Size of matrices	Number of nodes	T_{exec} (seconds)		Relative overhead (%)
		OMPI	OMPI FT-SPMD	
16384×16384	4	2027	2027	0.0
	8	1025	1027	0.3
	16	522	526	0.7
	32	274	277	0.9
32768×32768	32	2107	2113	0.3
	64	1094	1103	0.8
	128	597	609	1.9
	256	352	362	3.0
65536×65536	64	8405	8439	0.4
	128	4444	4469	0.6
	256	2406	2445	1.6

Fig. 12 displays the results on 64 nodes and 32768×32768 matrices. Because of the important slow down displayed by LAM/MPI when compared with OMPI (cf. Section V-D), fault tolerance overheads of LAM/MPI are computed relative to LAM/MPI runtimes in Fig. 12b. Proceeding this way allows for a fair comparison.

As expected, observed overheads increase as the number of checkpoints taken increases. However, for *OMPI FT-SPMD*, overheads increase differently from those of *DMTCP OMPI* and LAM/MPI. Below 3 checkpoints, incurred overheads are small for the three solutions. However, beyond 3 checkpoints, the overhead incurred by *DMTCP OMPI* and LAM/MPI grows significantly faster than the one incurred by *OMPI FT-SPMD*. Moreover, the latter succeeds in keeping its overhead almost constant and below 5% up to 31 checkpoints.

Similar trends were observed for runs with 16384×16384 and 65536×65536 matrices as well as node numbers between 4 and 256. Such behaviour for *OMPI FT-SPMD*, is the combined result of smaller checkpoint sizes and a communication-free rollback recovery protocol when taking checkpoints. Both *DMTCP OMPI* and LAM/MPI achieve bigger checkpoint sizes and use a blocking rollback recovery protocol. In other words, before taking a checkpoint all processes halt their execution in order to flush their network channels.

F. Evaluation of recovery performance

We have attempted to measure the time needed by LAM/MPI and *OMPI FT-SPMD* to recover from a distributed checkpoint taken at approximately half-way in the application execution. For LAM/MPI it corresponds to the time needed for all processes to load their respective checkpoints in memory. For *OMPI FT-SPMD* it also includes the consensus protocol used to determine the recovery line as well as the reexecution time in order to reconstruct the application context which existed at the time of checkpoint. As in previous experiments, measures were achieved for different configurations of matrix sizes and node numbers. Both systems display negligible overheads: below 1% on most tested configurations.

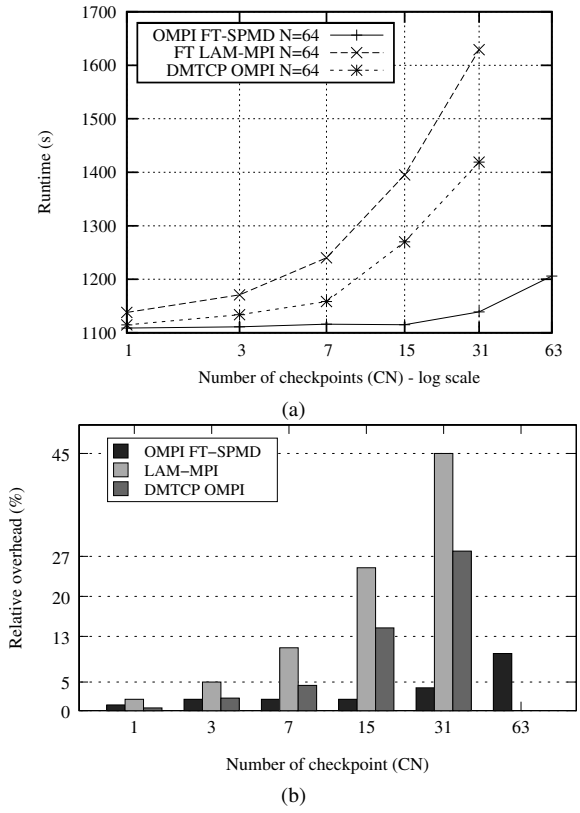


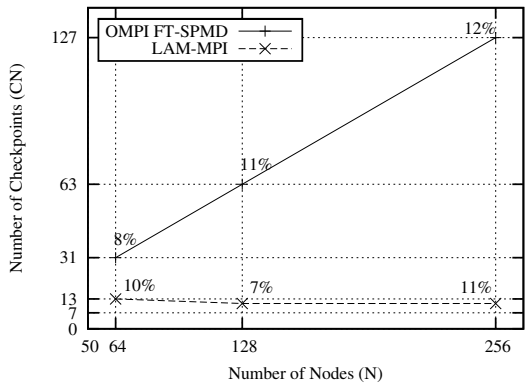
Fig. 12: Impact of checkpointing frequency on runtimes for 32768×32768 input matrices and 64 nodes.

G. Evaluation of fault tolerance with constant overhead

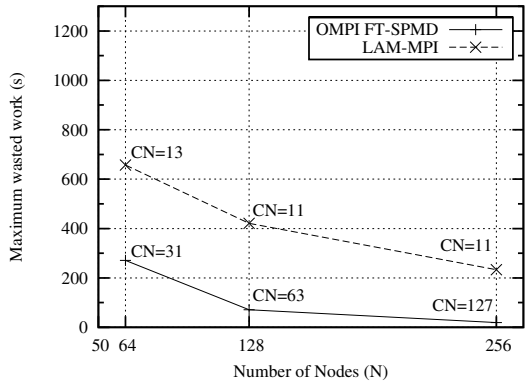
Knowing that (1) only a small overhead is involved during failure-free runs, and that (2) recovery will be fast in case of failures, are important properties of a checkpointing system. From a user's viewpoint, it is important that fault tolerance helps him meet his deadlines. Hence, what also matters is the amount of wasted work following a failure and the associated cost: the more checkpoints are taken, the less work is wasted but the higher the cost of fault tolerance is. Therefore, a good compromise should be sought so as to meet the user's time constraint(s). In the following, we show that *OMPI FT-SPMD* achieves better compromises than LAM/MPI.

Assuming the user has fixed a maximum 10% runtime overhead, we determine the maximum number of checkpoints that can be taken, within this limit, by LAM/MPI and *OMPI FT-SPMD*. Since the amount of time between two consecutive checkpoints is roughly the same, we are able to deduce the maximum amount of wasted work (in seconds). This actually corresponds to the worst case scenario where failure occurs right before a checkpoint is taken.

Results for 65536×65536 matrix sizes appear respectively in Fig. 13a and Fig. 13b. Whether it be for 64, 128 or 256 nodes, LAM/MPI does not exceed 13 checkpoints. *OMPI FT-SPMD*'s number of checkpoints starts at 31 on 64 nodes and doubles when doubling the nodes number. As a result, *OMPI FT-SPMD* succeeds in wasting less work than LAM/MPI.



(a)



(b)

Fig. 13: Maximum checkpoint number (CN) and maximum wasted work achieved by *OMPI FT-SPMD* and LAM/MPI for a maximum 10% runtime overhead and 65536×65536 input matrices.

Therefore, *OMPI FT-SPMD* exhibits an overall fault tolerance which is more efficient and more suited to our target applications.

VI. Conclusion

This paper addressed the problem of facilitating programming of parallel applications and endowing them with efficient fault tolerance. In order to do so, we propose a model for fault tolerance named MoLOToF which relies on a structured approach based on fault-tolerant skeletons and collaborations with the programmer and the environment. From the set of principles defined by MoLOToF, we have derived a framework named FT-SPMD. The latter applies the principles inherited from MoLOToF to a subset of SPMD domain decomposition applications. The experiments we have lead, show the efficiency of our fault tolerance approach over two other existing solutions which implement system-level checkpointing.

In the near future, we plan to experiment FT-SPMD with a greater variety of applications. Particularly, we plan to apply FT-SPMD on an industrial application for energy trading. This application distributes a stochastic control algorithm involving complex boundary data exchanges [15], and can last 46 minutes on 1024 nodes of an BlueGene/L supercomputer. Other projects include several improvements on FT-SPMD so as to support hardware failures, to lessen the work to be done by the programmer, and to implement an original protocol which relies on MoLOToF's skeleton-based application structure to enable on demand checkpointing in FT-SPMD. Finally, by applying MoLOToF to the Master-Worker we have derived another framework which we are currently experimenting with.

Acknowledgment

The authors would like to thank *Region Lorraine* for supporting this work.

References

[1] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[2] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," in *Journal of Physics: Conference Series*, vol. 78, 2007.

[3] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 222–238, 1983.

[4] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing," *In LACSI Symposium*, 2003.

[5] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V: a Multiprotocol Fault Tolerant MPI," *International Journal of High Performance Computing and Applications*, vol. 20, no. 3, pp. 319–333, 2006.

[6] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[7] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," in *Proceedings of SciDAC*, June 2006.

[8] J. Ansel, K. Aryay, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[9] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Recent Advances in Checkpoint/Recovery Systems," *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[10] G. Rodriguez, M. J. Martín, P. González, and J. Tourino, "Controller/Precompiler for Portable Checkpointing," *IEICE Transactions on Information and Systems, Special Issue on Parallel/Distributed Computing and Networking*, vol. E89-D, no. 2, pp. 408–417, Fabruary 2006.

[11] A. Beguelin, E. Seligman, and P. Stephan, "Application Level Fault Tolerance in Heterogeneous Networks of Workstations," *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 147–155, 1997.

[12] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke, "Fault-Tolerance on Regular Decomposition Grid Applications," in *Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 358–365.

[13] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf, "Cactus Tools for Grid Applications," *Cluster Computing*, vol. 4, no. 3, pp. 179–188, 2001.

[14] C. Makassikis and X. Warin and S. Vialle, "Distribution of a Stochastic Control Algorithm Applied to Gas Storage Valuation," *The 7th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT '07)*, 2007.

[15] S. Vialle and X. Warin and C. Makassikis, "Large Scale Distribution of Stochastic Control Algorithms for Financial Applications," *The 1st Workshop on Parallel and Distributed Computing in Finance (PDCoF08), IPDPS international conference*, 2008.

[16] "The Blitz++ library," http://www.oonumerics.org/blitz/.

[17] "Unified CodeCount," http://sunset.usc.edu/research/CODECOUNT/.