FT-GReLoSSS: a Skeletal-Based Approach towards Application Parallelization and Low-Overhead Fault Tolerance

Constantinos Makassikis, Stéphane Vialle, Xavier Warin

▶ To cite this version:

Constantinos Makassikis, Stéphane Vialle, Xavier Warin. FT-GReLoSSS: a Skeletal-Based Approach towards Application Parallelization and Low-Overhead Fault Tolerance. 20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing - PDP 2012, Feb 2012, Garching, Germany. 8 p. hal-00681664

HAL Id: hal-00681664
https://centralesupelec.hal.science/hal-00681664v1

Submitted on 22 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FT-GReLoSSS: a Skeletal-Based Approach towards Application Parallelization and Low-Overhead Fault Tolerance

Constantinos Makassikis
Université Henri Poincaré &
AlGorille INRIA Project Team
France
Email: Constantinos.Makassikis@loria.fr

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

Xavier Warin
EDF R&D & OSIRIS group
France
Email: Xavier.Warin@edf.fr

***Abstract*—FT-GReLoSSS (FTG) is a C++/MPI framework to ease the development of fault-tolerant parallel applications belonging to a SPMD family termed GReLoSSS. The originality of FTG is to rely on the MoLOToF programming model principles to facilitate the addition of an efficient checkpoint-based fault tolerance at the application level. Main features of MoLOToF encompass a structured application development based on *fault-tolerant "skeletons"* and lay emphasis on collaborations. The latter exist between the programmer, the framework and the underlying runtime middleware/environment. Together with the structured approach they contribute into achieving reduced checkpoint sizes, as well as reduced checkpoint and recovery overhead at runtime. This paper introduces the main principles of MoLOToF and the design of the FTG framework. To properly assess the framework's ease of use for a programmer as well as fault tolerance efficiency, a series of benchmarks were conducted up to 128 nodes on a multicore PC cluster. These benchmarks involved an existing parallel financial application for gas storage valuation, originally developed in collaboration with EDF company, and a rewritten version which made use of the FTG framework and its features. Experiments results display low-overhead compared to existing system-level counterparts.**

***Keywords*-distributed fault tolerance; application-level checkpointing; SPMD paradigm; framework; skeletons**

I. Introduction

As a result of increased competition, many industries have strived for new, more accurate simulation models. Their design, assessment and exploitation often requires huge amounts of computational power which eventually led those industries to adopt high performance distributed systems (HPDS).

In many cases, computations may be broken into several independent parts which can straightforwardly exploit HPDS. However, not all computations fit efficiently the embarrassingly parallel computation model. Programmers with little parallel background face three major issues. Firstly, they need to learn how to make their applications run efficiently on HPDS. Secondly, as the probability of failure increases with the number of computational resources (or nodes) used, developped applications have to be robust and fault-tolerant. Finally, HPDS systems involve rapidly evolving hardware and software which compels the use of *portable* fault tolerance (FT) solutions. Efficiency of these solutions is not an option as industrial environments are subject to time constraints: for example, some long-running applications in financial institutions are run overnight, and their results are expected in the morning in order to decide on the daily strategy to follow. Hence, FT efficiency translates into little slowdown of applications during failure-free time intervals. In case of failure, it translates into little wasted work and short restart times.

Checkpointing is a widely spread FT technique which consists in periodically saving the application state to stable storage. Following a failure, application execution is interrupted and the most recent saved state is then used to resume execution. This paper presents FTG, a specialized framework derived from the MoLOToF programming model [1]. MoLOToF aims to facilitate the development of efficient parallel applications and their endowment with efficient application-level checkpointing.

After covering related works (Section II), we describe the MoLOToF programming model and the software architecture of the FTG programming framework (Sections III and IV). Ease of development is assessed through the use of FTG on an existing industrial application for gas storage valuation from EDF company (Section V). Finally, the efficiency of the approach is evaluated experimentally (Section VI) before concluding (Section VII).

II. Related Works

Existing FT research for message passing applications has focused a lot on transparency. By combining a sequential checkpointer (*e.g.*: BLCR [2], MTCP [3] ...) and some rollback recovery (RR) protocol, it was possible to endow existing MPI libraries such as OpenMPI [4] or MPICH [5] with transparent FT or to provide more general solutions such as DMTCP [6] for socket-based distributed applications. RR protocols ensure that individual checkpoints of MPI processes remain consistent despite interdependencies created by communications: they form a *recovery line*. Most available solutions implement a so called blocking RR protocol which "freezes" communications while a checkpoint is made.

MPICH-V [5] is the exception, for it implements several other RR protocols. From our experience with the OpenMPI-BLCR pair or DMTCP, such solutions yield bulky checkpoint files, and are very sensitive to changes of the underlying runtime system. As a result of including too much system-dependent information, issues are often raised as the system is updated.

C^3 [7] and CPPC [8] strive for similar levels of transparency to the user, but at the application level. By leveraging source-to-source compilers, source code of C/Fortran MPI applications is automatically transformed such as the resulting application can checkpoint and restart itself. Besides unburdening the user from non-trivial source code transformations, this approach benefits from high portability for it works at the application level. Many legacy and recent applications written in C/Fortran can benefit from this approach. However, providing similar transformations is challenging for languages such as C++ which is commonly used in many applications.

On the other end, FT can be tackled manually. Though not clearly stated in the litterature, the manual approach to FT is not unusual. Admittedly, the approach is tedious even with the aid of third-party libraries (*e.g.*: TCS [9], SCR [10] checkpointing libraries). Nevertheless, resulting FT is very efficient and portable, and the user may improve it further based on his knowledge of the application. For production applications, the efficiency gained quickly outweighs the endeavour.

Finally, users can rely upon frameworks to achieve FT. PUL-RD [11] and Cactus [12] are examples where users are subject to some programming constraints in exchange of which they benefit among others from FT. Compared to the previous two approaches, frameworks also facilitate writing parallel applications. As a framework, FTG shares similarities with PUL-RD and Cactus as far as the parallelization model is concerned. All three involve iterative calculations with two array datastructures, swapped at the end of each iteration. FTG supports applications with none-trivial distribution of calculations among processes. It differs by relying on a specific programming model for fault tolerance named MoLOToF. The introduction of *fault-tolerant skeletons* yields an explicit structuration of the application, which in turn provides a simple, yet efficient, way to endow applications with FT. Hence, users are led to "actively" interact with the framework. Our fault-tolerant skeletons are inspired by the skeleton programming approach, and are designed at lower level. Hence they should be named "sub-skeletons" (or "low-level skeletons"). But for simplicity, in the rest of this paper we call them "skeletons".

III. MoLOToF programming model for FT

MoLOToF (**Mo**del for **L**ow-**O**verhead **T**olerance **o**f **F**aults) is a *programming model* geared towards easing the development of fault-tolerant parallel applications. MoLOToF relies (1) on a peculiar structuring of the application and (2) on establishing collaborations through interaction functionalities between the programmer, the framework and the underlying environment. To achieve these, MoLOToF introduces the concept of *fault-tolerant skeletons*. The latter are structured pieces of code endowed with fault-tolerant properties. Usually, they are made of fault-tolerant loops each of which has the ability to save and restore itself. As illustrated in fig. 1, a typical loop body contains calculation, and possibly communication phases (l. 5-6). Checkpoint definition (l. 7) within the loop allows to save calculations and related application state.

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

Fig. 1: MoLOToF fault-tolerant skeleton example.

A. Skeleton-based code structuring

Using fault-tolerant skeletons, the application is split into two broad types of code operations: *heavy* and *light* operations. Heavy operations correspond to time-consuming code. Such code is usually found within calculation and communication phases of a skeleton. Light operations designate every other piece of code which is really fast to reexecute, and hence not interesting to checkpoint. Such separation results in a straightforward save/restore (checkpointing) mechanism based on application reexecution. To illustrate the mechanics, let us assume an application made of a single skeleton as the one in fig. 1. As represented in fig. 2, the application comprises an initialization phase, a skeleton and a cleanup phase.

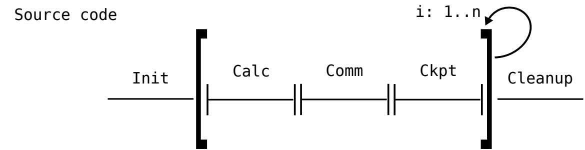

Fig. 2: Application source code representation.

During normal application execution (cf. Fig. 3), checkpointing occurs whenever a checkpoint location and a checkpoint condition are met. In our example, a checkpoint occurs on the second iteration. Among saved data is the iterator's value (*i.e.*: 2). Consistency of individual checkpoints is ensured by some adapted RR protocol. An application executes until it quits gracefully or until a failure occurs. In our example it fails somewhere during communications of the third iteration.

After the detection of a failure (cf. Fig. 4), the application is restarted from its most recent recovery line: here, the checkpoint taken at iteration 2. The application enters *recovery mode* (cf. "`restarting:true`" in fig. 4). Namely, each process restarts from the very beginning as in its initial run til it reaches the checkpoint location where the checkpoint it is supposed to restart from was achieved. During this course, only light operations are reexecuted therefore resulting in fast restarts. When reaching the appropriate checkpoint location, application context is restored. To complete recovery, checkpoint file contents are loaded back into the application, which can then normally resume its execution: the application falls back into *normal execution mode* (cf. "`restarting:false`" in fig. 4) and resumes with the third iteration.

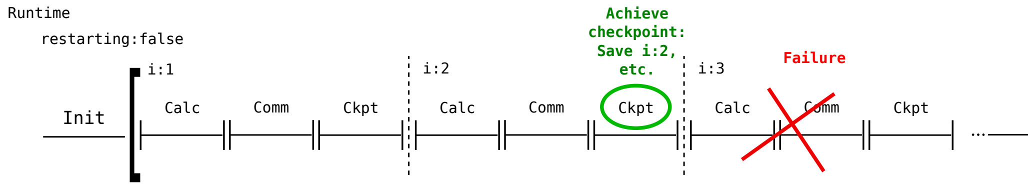

Fig. 3: Normal execution with checkpoint and failure.

B. Collaborations

To ease parallel programming, skeletons enclose a parallelization paradigm and come usually as part of a specialized framework. Through their use, the programmer easily develops his parallel application and installs checkpoint/restart semantics in it. Further involvement of the user may be required when checkpoints of fault-tolerant skeletons are not "self-contained". This means that their default contents do not capture entirely the application state. The user has to register as part of the checkpoint contents missing data in order to capture the application state. When fault-tolerant skeletons are self-contained, such action on behalf of the user is unnecessary as far as correctness is concerned. However, it is desirable in order to reduce the resulting checkpoint size and hence improve checkpointing efficiency. Therefore, in order to either constitute a consistent checkpoint or improve efficiency, the programmer's intervention may be needed. It makes all the more sense as the programmer, knowing the application semantics, may come up with smart choices.

Moreover, checkpoint efficiency may depend on the execution environment characteristics over time. For example, platforms under heavy load or aged platforms are more prone to failures. Hence, it is important that the application is able to adapt its fault tolerance based on external information.

To allow such interventions, FT skeletons come up with additional functionality allowing the programmer to control data which is enclosed in a checkpoint as well as setting the checkpointing frequency. Such interaction is seen as collaboration between the framework and the programmer. The other collaboration envisionned by MoLOToF lies between the framework and the underlying middleware, and is fully compatible with the spirit of fault-tolerant ecosystems such as the *Fault-Tolerant Backplane* [13].

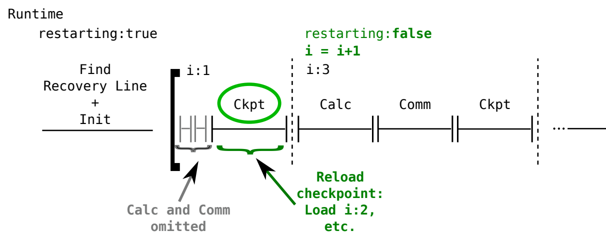

Fig. 4: Recovery execution.

To date, two framework implementations following the MoLOToF programming model exist. The first implementation is a Javaspace-based Master-Worker framework [14] which considers self-contained skeletons. The second implementation is a framework named FT-SPMD [1] which targets a broad family of SPMD applications named GReLoSSS (cf. Section IV). FT-SPMD benefitted from several modifications in its architecture and API which resulted in a seriously improved version called FT-GReLoSSS (FTG for short) which is presented hereafter. Compared to [1] the framework architecture is presented more in-depth and it is validated using an industrial application (cf. Section IV).

IV. FTG PROGRAMMING FRAMEWORK

A. GReLoSSS computation model

The GReLoSSS (**G**lobally **Re**laxed, **Lo**cally **S**trict **S**ynchronization **S**PMD) parallel computation model encompasses SPMD applications consisting in a main loop where each iteration (or superstep) contains a calculation and a communication phase. Such applications follow globally a classic BSP scheme [15]. However, these applications differ from BSP in two ways:

1) to improve efficiency on big parallel architectures, algorithms do not have a global synchronization between supersteps: each process starts its next superstep as soon as it has issued, but not necessarily completed, all its communications. Hence, global synchronization is relaxed as in the PRO model [16].
2) yet, to improve checkpointing efficiency, each process completes all its communications before starting the next superstep. Dependencies due to communications between processes disappear and consistent checkpointing is facilitated. Hence, compared to PRO, GReLoSSS has a locally strict synchronization which prevents overlapping of calculations with communications.

B. Application class supported by GReLoSSS and examples

The applications of the class supported by FTG (FT-GReLoSSS) involve two array datastructures one of which contains data used in the current superstep. The role of the second one depends on the application. A domain decomposition application such as Jacobi relaxations [17] will use that datastructure to store new results at the current superstep. A data circulation application such as in some parallel matrix multiplication algorithms [1] will use it to receive data for the next superstep from neighbouring processes.

During communications, processes exchange data corresponding to initial data or to intermediary results. Communications between processes may be quite varied, yet, in most cases, they are sufficiently foreseeable to be specified by the programmer. For example, in a Jacobi relaxations application, it consists in exchanging borders (or *shadow regions*) between

subdomains assigned to different processes. A subdomain designates the subset of the entire domain which was assigned to a process. Thus, a subdomain has the same datatype as array datastructures. At the end of communications, the two array datastructures are swapped.

Algorithms using two array datastructures (of 1 or more dimensions) such as the ones described may seem restrictive. But from our experience, they cover the needs of a fairly wide class of applications.

C. Software architecture and features

FTG is a C++ framework implementing programming principles described in MoLOToF. It is built on top of the MPI specification which makes it compatible with every MPI library. It provides a set of classes to ease the development of fault-tolerant GReLoSSS parallel applications.

Relations between main FTG classes are depicted in fig. 5. The `ftg_Skel` class represents a GReLoSSS skeleton and comprises:

- a *calculation kernel* (cf. `ftg_Calc_Kern`) which provides the calculation method and the skeleton's main loop iterator. It is also possible to specify a condition for communications achievement.
- a *routing plan* (cf. `ftg_Routing_plan`) which is responsible for determining and scheduling communication exchanges between processes.
- a *checkpoint* (cf. `ftg_Checkpoint`) which sets the location in the skeleton where checkpoints will be taken.
- two *array datastructures* (cf. `ftg_Domain`) as introduced previously in the GReLoSSS computation model (cf. Sections IV-A and IV-B).

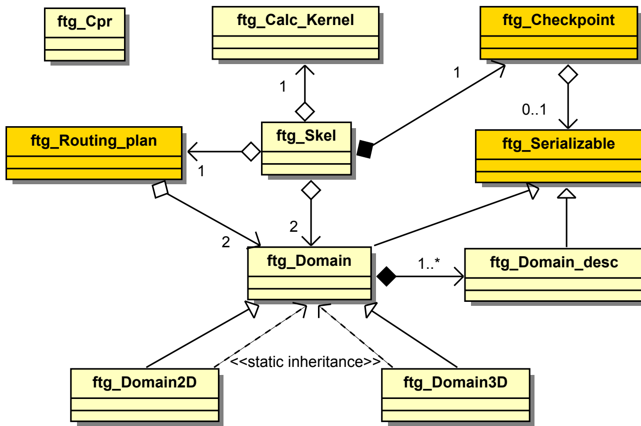

Fig. 5: FTG UML class diagram

The source code excerpt of `ftg_Skel` class (cf. Fig. 6) shows how these abstractions are layed out within the `execute` method. The resulting structure closely follows the example skeleton in fig. 1. It differs in the existence of conditional communications (l. 50-51) and the presence of a `swap` method (l. 61) where the two datastructures exchange their data according to the GReLoSSS computation model. Condition has to be verified globally: all processes enter communication phase or none. Notice the name prefixes: `ftgu_` are user-provided (l. 29-30, 41, 44), `ftg_` are framework provided (l. 16), and `ftgf_` are framework-provided based on user-provided methods (l. 38).

Concerning FT, the GReLoSSS skeleton defines a checkpoint right after the `swap` operation. By default, it contains data internal to the skeleton such as the two datastructures and skeleton iterators. `ftg_Skel` exposes to the programmer an interface allowing him to control data to include in the checkpoint. Furthermore, data which can be included in checkpoint files has to be either a primitive C++ type or derived from the framework's `ftg_Serializable` class (cf. Fig. 5), in which case the programmer has to provide a pair of `save` and `load` methods. This approach was used within FTG to make `ftg_Domain` and `ftg_Domain_desc` classes serializable. The latter is a helper class which describes the extent of a domain (or subdomain) along each dimension.

Upon instantiation of the skeleton, the user has to inform the routing plan whether application communication consists in borders exchanges or in circulating data. Aside from this, he may choose among different communication schemes. Currently, the routing plan integrates two schemes based on the `MPI_Issend-MPI_Irecv` pair of MPI primitives. The first scheme performs all communications in parallel, while the second tries to “pace” them so that there is only a limited amount of ongoing concurrent communications at a time [18]. Concerning data movement, the routing plan groups sparse data into contiguous large messages and also minimizes copies from/to communication buffers by detecting ranges of contiguous elements. This last feature proved useful in applications where data to transfer is contiguous as it avoids using intermediate buffers.

The calculation kernel is merely made of pure virtual methods which the programmer has to define.

The array datastructure provides the programmer with an interface to N-dimensional arrays enclosed in `ftg_Domain`. When interfacing his own array datastructure (by inheriting from `ftg_Domain`), the programmer provides information regarding (1) the way the domain is split among processes as well as (2) the way to access an element given its coordinates $(x_1, x_2, \cdots, x_n)$. Moreover, the programmer provides information regarding the storage order. In 2D, C storage order consists in storing the array line-wise, while a Fortran storage order consists in storing the array column-wise. Storage order generalizes in higher dimensions and can lead to more different storages depending on the order of storing each dimension.

Since the interface proposed by `ftg_Domain` targets N-dimensional domains, it manipulates a vector of coordinates which can be inconvenient to the programmer in small dimensions. It can also be less efficient. Therefore, FTG proposes specific interfaces for dimensions 2 and 3 (resp. `ftg_Domain2D` and `ftg_Domain3D` in fig. 5).

Finally, after splitting a domain among processes (cf. Fig. 7), any given element can be located either relative to the first element of the entire domain or relative to the first element of

```
template
<
  class T_numtype,   // Domain numerical type
  int N_rank,        // Domain dimension
  class T_iter,      // Main loop iterator type
>
class ftg_Skel
{
private:
   T_iter it; // Main loop iterator (user-defined)
   int step;  // Local iterator

   // DOUBLE DATASTRUCTURE POINTERS
   ftgu_Domain_t  *read_buffer, *write_buffer;

   ftg_Checkpoint_t  c; // Skeleton's checkpoint

   bool is_circulation; // Communication type

public:
  // Constructor, Destructor and other methods
  // ...

  // Executes the skeleton.
  void execute (void)
  {
    // Init routing plan and iterators
    rt = new ftg_Routing_plan_t(/*...*/);
    T_iter it_beg = ck->ftgu_beg();
    T_iter it_end = ck->ftgu_end();
    T_iter it_nxt;
    step = 0;

    // MAIN LOOP
    for (it = it_beg; it != it_end; it = it_nxt) {

      // CALCULATION PHASE
      ck->ftgf_calculate(read_buffer, write_buffer,
                         it);

      it_nxt = ck->ftgu_nxt(it);

      // CONDITIONAL COMMUNICATION PHASE
      if (ck->ftgu_do_execute_routing_plan(it))
        rt->ft_comms(it, it_nxt);

      // CHECKPOINT PHASE
      c.run(step++);

      // DATASTRUCTURES SWAP
      swap();
    }
  }
};
```

Fig. 6: FTG's fault-tolerant skeleton.

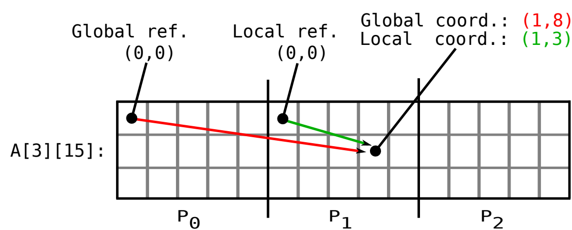

Fig. 7: Global versus local array access.

the subdomain it belongs to (resp. `Global ref` and `Local ref` in fig. 7). The first are called *global coordinates* while the second are called *local coordinates*.

FTG allows the programmer to use either of them without any further involvement. Access through local coordinates is preferable since it is faster, but it is not always the most convenient. Using global coordinates incurs a slight overhead due to an additional offset computation.

From a design standpoint, the use of dynamic polymorphism (through classic inheritance) is sufficient. However, since array access methods are often used, the resulting overhead is tremendous. That is why FTG relies also on *static polymorphism*, and more specifically on the *Curiously Recurring Template Pattern* (CRTP) [19].

In the following section, we decribe the *Swing* application and the steps to interface it efficiently with FTG.

V. *Swing* FINANCIAL APPLICATION MIGRATION

Due to house heating, demand in gas is higher in winter than in summer. Gas is mainly provided by pipes that cannot deliver more than a specific amount of gas per day and it results in far higher gas prices in winter. In order to provide energy to their customers, gas market actors have to own some storages that can help them smooth peak demand. In order to assess a project's rentability, gas companies can use a financial real option approach that can be implemented in a software.

A. Swing *application goal and implementation*

The *Swing* application is used at EDF company to valuate a gas storage facing the energy market. It aims at giving the average cash flow generated by the asset depending on some prices scenario. It also gives the management and hedging strategies [20]. These calculated strategies are used in a second application simulating the way the storage is used. This second application gives some cash flow distributions obtained by the storage management. While the time needed for the second application is short, the swing valuation can be very costly depending on the price models used to generate scenarios. The price models used at EDF for this software are a gaussian one-factor model (g), a normal inverse gaussian model (nig), and a two-factor gaussian model (g2d) [21]. The resolution method for the *Swing* application is the dynamic programming method that has been written in C++ and uses MPI for parallelization following the methodology in [18]. It also makes extensive use of the Blitz library for its convenient array manipulation facilities [22], and is about 18380 logical lines of code.

From an algorithmic standpoint, the *Swing* application fits perfectly the GReLoSSS computation model. While being a domain decomposition application, it is more complex than classic Jacobi relaxations. Indeed, over supersteps:

- calculations involve a subdomain of the entire calculation domain; that subdomain may change;
- shadow regions between processes have no fixed extent and may change as well.

To interface the existing application with FTG, we follow some typical development steps which are described hereafter and summarized in fig. 8.

B. *Development workflow*

As part of the typical development steps, we have to inherit from `ftg_Calc_Kern` to define a calculation kernel:

```
template <class T_numtype, int N_rank, class T_iter>
class Swing:
  public ftg_Calc_Kernel<T_numtype, N_rank, T_iter,
                         Swing_Domain>
{ }
```

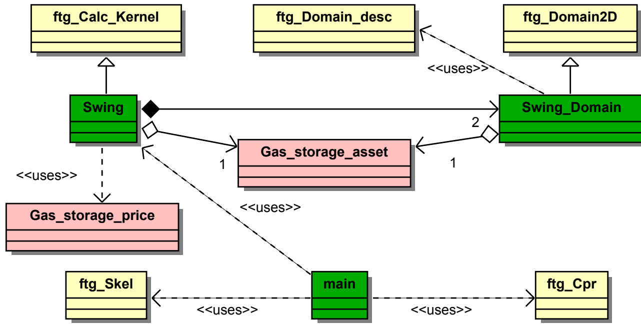

Fig. 8: *Swing* application within FTG.

and also to inherit from one of the available `ftg_Domain` classes to define the calculation domain:

```
template <class T_numtype, int N_rank, class T_iter>
class Swing_Domain:
  public ftg_Domain2D<T_numtype, N_rank, T_iter,
                      Swing_Domain>
  // implements Curiously Recurring Template Pattern
{ }
```

When inheriting from `ftg_Calc_Kern` we provide the calculation method:

```
void ftgu_calculate (ftgu_Domain_t *d1,
                     ftgu_Domain_t *d2,
                     T_iter step);
```

the main loop calculation iterator:

```
T_iter ftgu_beg (void);
T_iter ftgu_end (void);
T_iter ftgu_nxt (T_iter);
```

and a method which tells when to achieve communications:

```
bool  ftgu_do_execute_routing_plan (T_iter step);
```

When inheriting from one of the available `ftg_Domain` classes, we have to define two so called *partition methods*:

```
ftg_Domain_desc<N_rank>
  ftgu_data_possessed_def (int rank, int numprocs,
                           int step)
ftg_Domain_desc<N_rank>
  ftgu_data_needed_def    (int rank, int numprocs,
                           int step)
```

which tell what data is owned and what is needed by each process at each superstep. In an application with borders exchange, such as the *Swing* application, data owned is a subset of the data needed.

Moreover, we have to define a method telling the framework how to resize a domain:

```
void   ftgu_resize (TinyVector<int,N_rank> &extent)
```

and some methods on how to access any element or retrieve its memory address, given its local coordinates:

```
double   ftgu_lget     (int lx, int ly)
void     ftgu_lset     (int lx, int ly, double e)
double*  ftgu_lgetAddr(TinyVector<int,N_rank> &lcoord)
```

The resize method is not relevant to every application. Yet, in the case of *Swing*, it is interesting as subdomains attributed to each process and subdomains sizes vary across the course of an execution. Leveraging this feature, FTG can optimize memory management. The resize, the partition and the local access methods which the **u**ser provides (`ftgu_` prefix) are used to build the routing plan. The “`ftg_Domain*`” classes also use local access methods to implement the global access methods they provide the user with.

Finally, the application’s main function consisted in instantiating a calculation kernel and using it to initialize and execute the GReLoSSS skeleton. The corresponding source code has to be enclosed between the initialization and finalization statements of the framework provided by the `ftg_Cpr` (**C**heck**p**oint **r**ecovery) singleton class (cf. Fig. 8). The last step consists in choosing relevant checkpoint data. In particular, we unregister from the checkpoint the Domain which is useless at the end of the iteration, and register some application specific data which is not restored upon recovery (by mere reexecution), and hence has to be saved.

C. Ease of development assessement

The definition of the calculation method within our `Swing` calculation kernel involves the reuse of calculation functions in `Gas_storage_price`, and access to data in `Gas_storage_asset` (cf. Fig. 8). Both classes stem from the original *Swing* application. Similarly, partition methods in our `Swing_Domain` class require access to data and code reuse from `Gas_storage_asset`.

The calculation and partition methods are usually the most time-consuming to write. Especially the partition methods, which are prone to error due to indexes. Since compatible partition functions already exist in the original parallel *Swing* application, it is a matter of adapting them to use `ftg_Domain_desc` type to describe subdomains. The same applies for the calculation method, but some pieces of code have to be rewritten to use the accessors provided by `ftg_Domain` instead of those provided by Blitz Arrays. In the process, the interface of `Swing_Domain` was enriched to allow efficient and convenient access to ranges of data.

Another challenge we encountered concerned efficiency. In particular, partition methods have to be really fast as they are often called by the framework either to access elements through global coordinates or to build the routing plan.

In the end, the full fault-tolerant *Swing* application with FTG is about 353 logical lines of code less than the original application. Provided we are careful, the resulting application displays a clean and simple design as shown in fig. 8. Moreover, the application is fault-tolerant and the programmer has less lines of code to write. As shown in the next section, performance is as good as without FTG, sometimes slightly better.

VI. Performance experiments

The evaluation of FTG consists in a comparison with the popular OpenMPI (OMPI) library which implements a blocking checkpoint protocol in combination with BLCR. OMPI applications benefit from a system-level checkpoint/restart solution. Experiments led, assess FTG and OMPI performances without and with FT enabled. When FT is enabled, checkpoints are taken, and we consider application runtimes in the failure-free case and recovery times otherwise. We also consider checkpoint size reduction and their impact on runtime overhead.

In these experiments, we consider temporary crash failures of nodes which can be dealt with by having both FTG and

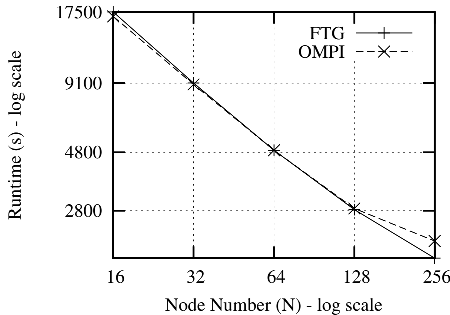

Fig. 9: Runtime comparison under the g2d price model.

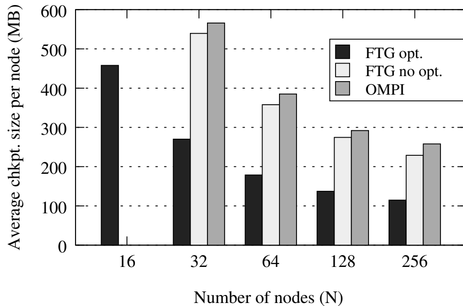

Fig. 10: Checkpoint sizes comparison under the g2d model.

OMPI store checkpoints locally to each node (*i.e.:* `/tmp`). Also, this setting avoids disrupting measurements due to additional communications resulting from significant data transfer across the network.

A. Testbed description

Experiments were led 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) and 4 GB of RAM. However, experiments were run with one process per node. Benchmarks applications use OpenMPI $1.5.3$ and were compiled with g++ $4.1.2$ compiler and $-O3$ optimization flag.

B. Runtime overheads comparison in absence of checkpoints

Fig. 9 shows that FTG exhibits negligible to no overhead for the g2d price model, and scales well up to 256 nodes where it performs better than the application not using FTG. Similar results were observed for the g and nig price models.

C. Checkpoint size

Checkpoint size is dominated by the size of the two array datastructures used for parallelization. Since at the end of a superstep, the input data is useless, it can be omitted from the checkpoint. This optimization results in checkpoint sizes per node twice as light compared to system-level checkpoints (cf. Fig. 10). The same can be observed for the g and nig models. These models require less memory and result in checkpoints size ranging between 20 MB and several kilobytes with FTG compared to 40 MB et 7 MB for *OMPI BLCR*. In what follows, we focus on the g2d model: its' long runtimes make it a good candidate for FT.

D. Fault tolerance performance without failures

For this experiment, we have run both versions of our application with the g2d model and different number of checkpoints: the lengthier the benchmark the more checkpoints were achieved in order to minimize the wasted amount of time in case of failure. In the present experiments we set the maximum allowed wasted amount of time to 4 minutes. Fig. 11 reports the runtimes on 128, 64 and 32 nodes. The plots compare OMPI with BLCR and FTG. In the latter's case we measured runtimes without and with checkpoint size optimization in an attempt to quantify the impact of checkpoint size reduction. The difference remains marginal ($0-3\%$) and is the highest with the biggest checkpoint sizes as the ones involved on 32 nodes. The difference is expected to grow further with bulkier checkpoint files. The remaining overhead observed with OMPI-BLCR ranges between 6% and 40%. Overhead incurred by FTG is always lower and does not exceed 8% in all cases. OMPI-BLCR's overhead is mainly attributed to its blocking checkpoint protocol [4], which, unlike FTG's protocol, involves communications and "freezes" the application execution in order to checkpoint.

E. Recovery overhead

Recovery from a checkpoint comprises (1) a negotiation phase where processes decide from which recovery line to recover, followed by a (2) context recovery phase, and finally (3) a recovery from checkpoint file phase (*i.e.*: time to load data). Measured negotiation phase time is negligible ($< 10ms$). Context recovery phase is small as well. Overall recovery time from a checkpoint is rather small as it does not exceed 1s for the g2d model on 128, 64 and 32 nodes. Thus, fast restarts combined with the low-overhead checkpointing pointed out earlier make FTG suited for applications with time constraints.

VII. Conclusion and Perspectives

Endowing parallel applications with efficient checkpoint-based FT at the application level can be a tedious task which adds up to the existing difficulties of parallelization. Our approach based on the MoLOToF programming model introduces fault-tolerant skeletons and results in a tractable way for users to endow efficient fault tolerance into their applications. Moreover, combined with a specialized framework, MoLOToF eases parallel programming, and encourages a synergy between the user, the framework and the runtime environment to improve FT efficiency. MoLOToF is applied to the GReLoSSS family of applications which we characterized in this paper. The application of the resulting FTG framework to an industrial application of EDF company showed that initial development with FTG involved simple steps and points out some elements to watch in order to minimize runtime overhead. Finally, experiments show the effectiveness of the overall approach and especially the efficiency of FT. For the same number of checkpoints achieved, FTG yields smaller checkpoint sizes than OMPI-BLCR and incurs at most 8%

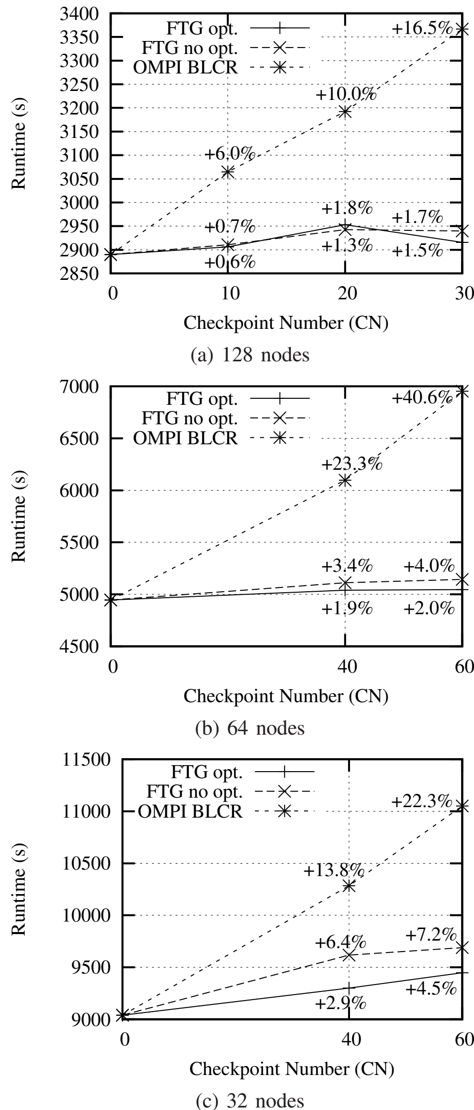

Fig. 11: Runtimes comparison under the g2d model when increasing number of achieved checkpoints.

runtime overhead. Recovery from achieved checkpoints exhibited negligible overheads. These results confirm previous ones we achieved on more elementary but varied benchmark applications [1].

Many principles of MoLOToF were used in the design of FTG. But some of them, such as the integration with fault-tolerant ecosystems, still have to be integrated and experimented with. Moreover, due to its inherent portability, the approach appears viable for hybrid GP-GPU applications. FTG might further be extended to support other parallelization models such as asynchronous distributed iterative algorithms. Future works are planned along the aforementioned lines.

Acknowledgment

This research is partially supported by *Region Lorraine*.

References

[1] C. Makassikis, V. Galtier, and S. Vialle, "A Skeletal-Based Approach for the Development of Fault-Tolerant SPMD Applications," in *Proceedings of the 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2010.

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

[3] M. Rieker, J. Ansel, and G. Cooperman, "Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux," in *PDPTA*, 2006, pp. 492–498.

[4] 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.

[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. 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.

[7] 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.

[8] G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo, "CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[9] N. Stone, J. Kochmar, R. Reddy, J. R. Scott, J. Sommerfield, and C. Vizino, "A Checkpoint and Recovery System for the Pittsburgh Supercomputing Center Terascale Computing System," Pittsburgh Supercomputing Center, Tech. Rep., 2001.

[10] A. Moody and G. Bronevetsky and K. Mohror and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010.

[11] 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*. IEEE Computer Society, 1995, pp. 358–365.

[12] 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.

[13] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A Coordinated infrastructure for Fault-Tolerant Systems," in *Proceedings of the 38th International Conference on Parallel Processing (ICPP)*, 2009.

[14] V. Galtier, C. Makassikis, and S. Vialle, "A Javaspace-based Framework for Efficient Fault-Tolerant Master-Worker Distributed Applications," in *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE Computer Society, 2011, pp. 272–276.

[15] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[16] A. H. Gebremedhin, I. G. Lassous, J. Gustedt, and J. A. Telle, "PRO: A Model for Parallel Resource-Optimal Computation," in *Proceedings of the 16th International Symposium on High Performance Computing Systems and Applications (HPCS)*. IEEE Computer Society, 2002.

[17] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. The MIT Press, 1999, ch. 4.

[18] 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)*, 2007.

[19] J. O. Coplien, "Curiously recurring template patterns," *C++ Report (SIGS Publications)*, vol. 7, no. 2, 1995.

[20] X. Warin, "Gas storage hedging," in *Numerical methods in finance , proceedings in mathematics*. Springer, 2011.

[21] X. Warin and W. van Ackooij, "Electricity asset management with future hedging," EDF, Tech. Rep. H-R33-2006-04103-FR, 2008.

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