A Skeletal-Based Approach towards Application Parallelization and Low-Overhead Fault Tolerance

Constantinos Makassikis\textsuperscript{1,3}, Stéphane Vialle\textsuperscript{2,3}, Xavier Warin\textsuperscript{4}

\textsuperscript{1}Université de Lorraine, Nancy, France
\textsuperscript{2}SUPELEC - UMI-2958, Metz, France
\textsuperscript{3}AlGorille INRIA Project Team, Nancy, France
\textsuperscript{4}EDF R&D, OSIRIS group, France

PDP, Garching, Germany, 15-17 Feb. 2012
Industries need to run faster, big or bigger simulations
  e.g.: EDF (french electricity company) needed to experiment with compute-intensive price models
  Applications under soft time constraints
Parallelism makes it possible with more processing units, but
  How to exploit so many processing units?
  How to deal with increased failures?
  Efficiently?

Need for efficient fault tolerance
  Little slowdown during failure-free periods
  Little wasted work
  Fast recovery
Goals and definition

Goals

- Ease development of fault-tolerant distributed application
- Yield efficient and portable fault tolerance
  - Application-level
  - Checkpoint-based (temporary/permanent crash failures)

MoLOToF

- Programming Model for Low-Overhead Tolerance of Faults
- Set of principles based on Fault-Tolerant (FT) skeletons
Principles 1: Application organization

**P1.1** Distributed application is structured using FT skeletons

**P1.2** Each process is a succession of FT skeletons

**P1.3** FT skeletons define heavy and light operations
  - Operations outside skeletons should be light
Principles 2: Checkpoint/Restart mechanism

P2.1 Checkpoint/restart is made of **two execution modes**:  
- normal  
- recovery

P2.2 Context recovery is restored by **selective reexecution**
Principles 3: Collaborations

P3.1 Programmer–Framework
- Placement: Where to place skeleton?
- Correction and efficiency: What to save? How?
- Frequency: How often to checkpoint?

P3.2 Framework–Environment
- Ability to be driven by external requests

Example
- On demand checkpoint and checkpoint frequency modification
- Administrator, scheduler, FT ecosystem → runtime adaptation
MoLOToF frameworks

- MoLOToF principles applied to parallel computing models
  → Ease parallel programming

MoLOToF
  --
  FT skeletons
  Two exec. modes
  Collaborations

ToMaWork
  --
  JavaSpaces
  -- Master-Worker

FT-GReLoSSS
  --
  C++/MPI
  -- GReLoSSS
GReLoSSS computation model

- Globally Relaxed, Locally Strict Synchronization SPMD
- GReLoSSS ≈ classic BSP + relaxation between supersteps
  e.g.: Jacobi relaxations, Matrix product, Swing
  \[\rightarrow\] Structure of FT-GReLoSSS skeleton
Software architecture: UML class diagram
Swing application description

- Valuation of gas storages facing the energy markets at EDF company
- Provides:
  - Average cash flow generated depending on some prices scenario
  - Management and hedging strategies
- Prices scenario are generated by price models
  - different computation and memory requirements
  - e.g.: two-factor gaussian (g2d) is compute and memory intensive
- Domain decomposition with variable subdomain and shadow zones
Swing application migration to FTG

Development steps and pitfalls

- Inherit from framework base classes
- Reuse existing classes with adjustments when necessary

<table>
<thead>
<tr>
<th>FTG base classes</th>
<th>New Swing classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swing application classes</td>
<td></td>
</tr>
</tbody>
</table>

C. Makassikis (UdL)
Development steps and pitfalls

Definition of domain and its partition functions
(Swing_Domain)

- Interface Blitz++ Arrays of original Swing to ftg_Domain
  \approx 12 \text{ methods}
  \approx 100 \text{ logical lines overall}

- Watch out index computations!
- Partition functions are called often \rightarrow \text{ must be fast}
Development steps and pitfalls

Definition of the calculation kernel (Swing)

- Includes methods to define calculation and skeleton’ main loop iterator (5 methods)
- Required migration of ancillary calculation code to use Swing Domain instead of Blitz++ Arrays (3 methods) (Swing and Gas storage price)
- \( \approx 200 \) logical lines (LSLOC) overall
Quantitative comparison

- Parallel application with/without FTG
  - share \(\approx 10137\) logical lines of legacy code
  - have resp. 1333 and 1125 LSLOC for parallelism (\(\approx 10\%\))
    - comprises adapted existing code and new code

- Detail for Swing FTG
  - Legacy (\(\approx 10137\) LSLOC)
  - Easy modif (\(\approx 700\) LSLOC)
  - More complex modif (\(\approx 300\) LSLOC)
Evaluation methodology

Testbed description

- Intercell cluster at Supélec 256 nodes (4 GB, 1 Gigabit Ethernet)
- No parallel filesystem yet → checkpoints stored in /tmp

<table>
<thead>
<tr>
<th>Application</th>
<th>Swing Original</th>
<th>Swing Migrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution</td>
<td>OpenMPI</td>
<td>FTG + OpenMPI</td>
</tr>
<tr>
<td>Fault tolerance type</td>
<td>OpenMPI BLCR</td>
<td>FTG</td>
</tr>
<tr>
<td></td>
<td>System-level</td>
<td>Application-level</td>
</tr>
</tbody>
</table>
Methodology

Performance Evaluation

- Without Fault tolerance
- With Fault tolerance
  - Without Failures
  - With Failures
Performance without fault tolerance

- Overall same performances with/without FTG
- Similar results with other price models
Performance with FT – no failures

- OpenMPI’s blocking coordinated algorithm accounts for poor performance when increasing checkpoint frequency
Performance experiments

Performance with FT and failures

Recovery overhead

- Measured time to restart from a checkpoint (no detection time)
  - Neither solution has integrated failure detection
- Comprises
  1. Negotiation phase (< 10 ms)
  2. Context recovery
  3. Checkpoint data loading

Overall recovery time: less than 1 s
Conclusion and Perspectives

- MoLOToF introduces a tractable way for efficient fault tolerance
  - Combination of fault-tolerant skeletons
  - Collaborations
- Combination to parallel computation model yielded FTG
- Application to an industrial application showed simple development steps
- Experiments showed effectiveness of the approach
  - Little runtime overhead (< 8%)
  - Little wasted work
  - Fast recovery
Conclusion and Perspectives

- Another framework with similar principles exists for Master-Worker
- What about feasibility of approach
  - with other parallel algorithm families?
  - asynchronous iterative algorithms?
  - on heterogeneous computing setting?
  - GP-GPU?
Thanks for your attention

QUESTIONS?
Checkpoint sizes

![Graph showing average checkpoint size per node (MB) versus number of nodes (N). The graph compares FTG opt., FTG no opt., and OMPI. The x-axis represents the number of nodes (16, 32, 64, 128, 256), and the y-axis represents the average checkpoint size per node (MB).]
Performance with FT – no failures
Performance with FT – no failures

![Graph showing performance with different checkpoint numbers and various runtime models.](attachment:image.png)

- **g2d price model -- 32 nodes**
- **FTG opt.**
- **FTG no opt.**
- **OMPI BLCR**

- Runtime (s)
- Checkpoint Number (CN)

- **FTG opt.**
  - +13.8%
  - +6.4%
  - +2.9%

- **OMPI BLCR**
  - +22.3%
  - +7.2%
  - +4.5%