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

Constantinos Makassikis\textsuperscript{2,3},
Virginie Galtier\textsuperscript{1}, Stéphane Vialle\textsuperscript{1,2}

\textsuperscript{1}SUPELEC - UMI-2958, Metz, France
\textsuperscript{2}AlGorille INRIA Project Team, Nancy, France
\textsuperscript{3}Université Henri Poincaré, Nancy, France

LAHMA, Orléans, France, 14 Dec. 2010
Research Context

Extensible Machines

- Easily increase processing power
- Cluster-like architecture
- Wide acceptance

Intercell PC cluster (Supélec)

Demanding Applications

- Increased needs in computation resources for bigger simulations
- Need to respect some deadline
- Diverse application domains:

Energy Industry
Gaz Management Optimization
Application by EDF R&D and Supélec
Research Context

Some of the problems

- Writing parallel applications
- Dealing with failures
  - Node increase → Machine reliability decrease
  - Mostly fail-stop faults/failures

Some consequences

- Uncertain termination of long-running applications
- Miss of deadlines
- Waste of computations, energy and money
Research Context

### Some of the problems
- Writing parallel applications
- Dealing with failures
  - Node increase $\rightarrow$ Machine reliability decrease
  - Mostly fail-stop faults/failures

### Some consequences
- Uncertain termination of long-running applications
- Miss of deadlines
- Waste of computations, energy and money

**Need for fault tolerance**
Research Context: Checkpoint/Restart (CPR)

Distributed Checkpoint/Restart (CPR)

- Saves consistent intermediate states of distributed application
- Avoids restart of application from very beginning
- Inherent overheads: runtime, recovery, disk usage
  → There still is a risk to miss deadlines
  → Need to minimize overheads
## Research Context: CPR Implementation levels duality

### System-level
- Dumps in-memory bytes of processes to disk
  - High transparency to the programmer
  - Low portability
  - Low efficiency (e.g.: checkpoint size, protocol)

### Application-level
- Requires complex application source code transformations
  - Low transparency to the programmer (most of the time)
  - High portability
  - Potentially high efficiency
    - Exploit application semantics to reduce FT overheads
Research Context: CPR Implementation levels duality

<table>
<thead>
<tr>
<th>System-level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dumps in-memory bytes of processes to disk</strong></td>
</tr>
<tr>
<td>- High transparency to the programmer</td>
</tr>
<tr>
<td>- Low portability</td>
</tr>
<tr>
<td>- Low efficiency (e.g.: checkpoint size, protocol)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application-level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Requires complex application source code transformations</strong></td>
</tr>
<tr>
<td>- Low transparency to the programmer (most of the time)</td>
</tr>
<tr>
<td>- High portability</td>
</tr>
<tr>
<td>- Potentially high efficiency</td>
</tr>
<tr>
<td>* Exploit application semantics to reduce FT overheads</td>
</tr>
</tbody>
</table>

- But, both levels do not address directly easiness of programming
Our approach

- Work at application level for
  - Natural portability
  - Exploitation of application semantics

- Addresses easiness of
  - Adding efficient application-level FT
  - Programming distributed applications

- Means:
  - New skeletal-based fault tolerance model
  - Specialized framework derivation
MoLOT oF: Definition and Aims

**MoLOT oF**
- Model for Low-Overhead Tolerance of Faults

**What is MoLOT oF?**
- A set of rules to develop fault-tolerant parallel applications
- Rules revolve around the concept of fault-tolerant skeleton

**What are MoLOT oF’s aims?**
- Facilitate fault-tolerant distributed applications development
- Achieve efficient and portable fault tolerance
MoLOToF: Fault-tolerant skeletons

- **Focus** fault tolerance on **important parts** of the application
  - computation intensive pieces of code → heavy operations
  - other operations are known as light operations

- Two kinds: sequential and parallel

**Example of simple skeletons with compute-intensive loops**

```plaintext
FT_SEQ_Skel
{
    FT_LOOP
    {
        calculations()
        checkpoint()
    }
}
Sequential Skeleton

FT_PAR_Skel
{
    FT_LOOP
    {
        calculations()
        communications()
        checkpoint()
    }
}
Parallel Skeleton
```
MoLOTof: Skeleton-based application organization

- A distributed application is made of several processes
- In MoLOTof, each process is a **succession of fault-tolerant skeletons**
MoLOToF: Save/Restore mechanics

Normal execution mode
- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds

\[ P_i \]

- \( FT_{\text{Seq}_i} \)
- \( FT_{\text{Par}_i} \)
MoLOToF: Save/Restore mechanics

Normal execution mode

- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds
MoLOToF: Save/Restore mechanics

Normal execution mode
- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds

```
P_i
```

```
FT_Seq_Skel
```

```
FT_Seq_Skel
```

```
FT_Par_Skel1
```

```
FT_Par_Skel2
```

- calculations()
- checkpoint()
MoLOToF: Save/Restore mechanics

Normal execution mode

- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds

```
  P_i
  |---->
  |     
  |     
FT_Seq_Skel
  |---->
  |     
  |     
FT_Seq_Skel
  |---->
  |     
  |     
FT_Par_Skel1
  |---->
  |     
  |     
FT_Par_Skel2
```

```
calculations()
checkpoint()
calculations()
checkpoint()
calculations()
communications()
checkpoint()
```
Normal execution mode

- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds

Suppose $P_i$ checkpoints at iteration $i$
MoLOToF: Save/Restore mechanics

Normal execution mode
- Application and FT code
- A process saves itself when:
  1. at checkpoint locations
  2. checkpoint condition holds

Suppose $P_i$ checkpoints at iteration $i$

Suppose $P_i$ fails at iteration $i + 1$
Recovery execution mode

- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations
Recovery execution mode

- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations

```
Makassikis, Galtier, Vialle
A Skeletal-based Approach... LAHMA 16 / 43
```
MoLOTof: Save/Restore mechanics

**Recovery execution mode**

- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations

Diagram:

```
  P_i

  FT_Seq_Skel

  FT_Seq_Skel

  FT_Par_Skel1

  FT_Par_Skel2
```

```
calculations()
checkpoint()
calculations()
checkpoint()
```
MoLOToF: Save/Restore mechanics

Recovery execution mode

- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations

```
FT_Seq_Skel
   ↓
FT_Seq_Skel
   ↓
FT_Par_Skel1
   ↓
FT_Par_Skel2
```

(Makassikis, Galtier, Vialle)
MoLOTToF: Save/Restore mechanics

Recovery execution mode

- **Recovery line** determination

- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations
  3. Checkpoint data reload on “right” checkpoint location

Diagram:

\[ P_i \]

- **FT_Seq_Skel**
  - calculations()
  - checkpoint()

- **FT_Seq_Skel**
  - calculations()
  - checkpoint()

- **FT_Par_Skel1**
  - calculations()
  - checkpoint()

- **FT_Par_Skel2**
  - calculations()
  - checkpoint()
MoLOTof: Save/Restore mechanics

Recovery execution mode

- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations
  3. Checkpoint data reload on “right” checkpoint location
  4. Return to *normal execution mode*

$P_i$

- $\text{FT\_Seq\_Skel}$
  - `checkpoint()`
  - `calculations()`
- $\text{FT\_Par\_Skel1}$
  - `checkpoint()`
  - `calculations()`
  - `communications()`
- $\text{FT\_Par\_Skel2}$
  - `checkpoint()`

Makassikis, Galtier, Vialle (A Skeletal-based Approach...)

LAHMA 16 / 43
MoLOTToF: Collaborations

“Programmer–Framework” (require programmer’s assistance)

1. Collaboration for **placement**
   - Where to place skeletons?

2. Collaboration for **correctness** and **efficiency**
   - Which data to include in checkpoints?

3. Collaboration for **frequency**
   - How often a checkpoint must be achieved?

“Framework–Environment” (require environment's assistance)

- Enable **externally driven functioning** to tune fault tolerance

- Examples:
  - Ondemand checkpoint or checkpoint frequency modification
  - Requests by administrator/FT ecosystem
    (e.g.: maintenance operation, predicted failure)
FT-GReLoSSS: Framework architecture

- MoLOTof Principles
- Parallel Algorithms Family: SPMD domain decomposition

- User Application
  - FT Skeletons
- C++

- Light Middleware
  - Driven Functioning
  - Failure Detection
  - I/O
  - MPI Library

- PC Cluster
FT-GReLoSSS: Parallelization model

FT Skeleton

1
2
3
4
FT-GReLoSSS: Parallelization model

1. Computation

FT Skeleton

CPU
FT-GReLoSSS: Parallelization model

1. **Computation**

![Diagram showing FT Skeleton, CPU, Array 1, Array 2, Double datastructure, N-dimension arrays connected with arrows.]
FT-GReLoSSS: Parallelization model

1. **Computation**

   - CPU
   - Double datastructure
   - N-dimension arrays
   - Array 1
   - Array 2

   FT Skeleton

1. 1
2. 2
3. 3
4. 4
FT-GReLoSSS: Parallelization model

1. **Computation**
   - FT Skeleton
   - CPU
   - Array 1
   - Array 2
   - Double datastructure
   - N-dimension arrays

2. **Communications**
   - Routing Plan Execution and Update

Makassikis, Galtier, Vialle (A Skeletal-based Approach...)

LAHMA 21 / 43
FT-GReLoSSS: Parallelization model

1. **Computation**
   - CPU
   - FT Skeleton
   - Array 1
   - Array 2
   - Double datastructure
   - N-dimension arrays

2. **Communications**
   - Routing Plan Execution
   - and Update

3. **Swap Datastructures**
FT-GReLoSSS: Parallelization model

1. **Computation**

   - CPU
   - Array 1
   - Array 2
   - Double datastructure
   - N-dimension arrays

2. **Communications**

   - Routing Plan Execution and Update

3. **Swap Datastructures**

4. **Checkpoint**
FT-GReLoSSS: Parallelization model

1. **Computation**

2. **Communications**
   - Routing Plan Execution and Update

3. **Swap Datastructures**

4. **Checkpoint**

**GReLoSSS family**
- Globally Relaxed between supersteps
- Locally Strict Synchronization SPMD within superstep
FT-GReLoSSS: Relationships between concepts

- **FT_Mgr**
  - Uses:
    - **Checkpoint**
    - **FT Skeleton**
    - **Routing Plan**
  - **Calculation Kernel**

- **Checkpoint**
  - Uses:
    - **FT_Mgr**

- **FT Skeleton**
  - Uses:
    - **FT_Mgr**
    - **Routing Plan**
    - **Calculation Kernel**
  - **User**
    - **FT Tuning**

- **Routing Plan**
  - Uses:
    - **FT_Mgr**
    - **Domain**
  - **User**
    - **Domain**

- **Calculation Kernel**
  - Uses:
    - **FT_Mgr**
    - **User**
  - **User**
    - **Calculation Kernel**

- **Domain**
  - Uses:
    - **Routing Plan**
    - **User**
  - **User**
    - **Domain**
Evaluation: Ease of development

- **Metrics**: Number of source code lines (physical and logical)
- **Comparison**: framework vs frameworkless versions of \textit{Matmult}
- **Matmult** application: dense matrix multiplication on a ring of processors

### Results

<table>
<thead>
<tr>
<th>Line Type</th>
<th>Matmult v1</th>
<th>Matmult v2</th>
<th>Absolute Overhead</th>
<th>Relative Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical</td>
<td>258</td>
<td>295</td>
<td>+37 lines</td>
<td>+14.3</td>
</tr>
<tr>
<td>logical</td>
<td>168</td>
<td>186</td>
<td>+18 lines</td>
<td>+10.7</td>
</tr>
</tbody>
</table>

- Acceptable overheads
  (most additional instructions have low algorithmic complexity)
Evaluation: Ease of development

- **Metrics**: Number of source code lines (physical and logical)
- **Comparison**: framework vs frameworkless versions of *Matmult*
- **Matmult** application: dense matrix multiplication on a ring of processors

### Results

<table>
<thead>
<tr>
<th>Line Type</th>
<th>Matmult Type</th>
<th>Matmult v1</th>
<th>Matmult v2</th>
<th>Absolute Overhead</th>
<th>Relative Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical</td>
<td></td>
<td>258</td>
<td>295</td>
<td>+37 lines</td>
<td>+14.3</td>
</tr>
<tr>
<td>logical</td>
<td></td>
<td>168</td>
<td>186</td>
<td>+18 lines</td>
<td>+10.7</td>
</tr>
</tbody>
</table>

- **Acceptable overheads**
  (most additional instructions have low algorithmic complexity)
Evaluation: Testbed and benchmark

Compared systems: system and application level

- FT-GReLoSSS with Open MPI 1.3.3 (OMPI FT-GReLoSSS)
- LAM/MPI 7.1.4 (LAM/MPI)
- DMTCP r481 with Open MPI 1.3.3 (DMTCP OMPI)

Testbed description

- Intercell cluster at Supélec 256 nodes (4 GiB, 1 Gigabit Ethernet)

Benchmark Application: Matmult

<table>
<thead>
<tr>
<th>Individual matrix size</th>
<th>16384 × 16384</th>
<th>32768 × 32768</th>
<th>65536 × 65536</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total application size in RAM</td>
<td>~ 6 GiB</td>
<td>~ 24 GiB</td>
<td>~ 48 GiB</td>
</tr>
<tr>
<td>Total FT-GReLoSSS application checkpoint size</td>
<td>~ 4 GiB</td>
<td>~ 16 GiB</td>
<td>~ 32 GiB</td>
</tr>
</tbody>
</table>
Evaluation: Testbed and benchmark

Compared systems: system and application level
- FT-GReLoSSS with Open MPI 1.3.3 (OMPI FT-GReLoSSS)
- LAM/MPI 7.1.4 (LAM/MPI)
- DMTCP r481 with Open MPI 1.3.3 (DMTCP OMPI)

Testbed description
- Intercell cluster at Supélec 256 nodes (4 GiB, 1 Gigabit Ethernet)

Benchmark Application: Matmult

<table>
<thead>
<tr>
<th>Individual matrix size</th>
<th>16384 × 16384</th>
<th>32768 × 32768</th>
<th>65536 × 65536</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total application size in RAM</td>
<td>~ 6 GiB</td>
<td>~ 24 GiB</td>
<td>~ 48 GiB</td>
</tr>
<tr>
<td>Total FT-GReLoSSS application checkpoint size</td>
<td>~ 4 GiB</td>
<td>~ 16 GiB</td>
<td>~ 32 GiB</td>
</tr>
</tbody>
</table>

Lighter checkpoints thanks to Programmer–Framework collaborations
Evaluation: Performance with FT and no failures

- $32768 \times 32768$ (24 GiB) - 64 Nodes
Evaluation: Performance **with FT and no failures**

- $32768 \times 32768$ (24 GiB) - 64 Nodes

![Graph showing runtime vs. number of achieved checkpoints](image_url)

- OMPI FT-GReLoSSS N=64
- LAM/MPI N=64
- DMTCP OMPI N=64

Makassikis, Galtier, Vialle (A Skeletal-based Approach...}

LAHMA 28 / 43
Evaluation: Performance **with FT and no failures**

- $32768 \times 32768$ (24 GiB) - 64 Nodes

![Graph showing runtime vs. number of achieved checkpoints for different MPI implementations.](image-url)
Conclusion and Perspectives

Contributions

- New application-level approach to ease addition of fault tolerance
  - Based on MoLOTof fault tolerance model which involves
    - Skeleton-based application organization
    - Collaborations
  - Combines MoLOTof with parallel algorithms families
- The derived FT-GReLoSSS framework shows good results

Perspectives

- Improve further ease of development
- Endow FT-GReLoSSS with “Framework-Environment” collaborations
- Apply FT-GReLoSSS to an industrial application
  - stochastic control algorithm with complex boundary exchanges
  - 46 minutes on 1024 nodes of a BlueGene/L supercomputer
Thanks for your attention

QUESTIONS ?
Source code of Matmult’s main I

```c
int main(int argc, char **argv)
{
    // Initializations ——————————————————
    
    // + MPI related initializations.
    MPI_Init(&argc, &argv)
    // ...

    // + Init. of FT-GReLoSSS’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 ——————————————————

    // + Checkpoint correctness: add result matrix to
    // checkpoint
    // + C->dataFirst(): address to the first element
```
Source code of Matmult’s main II

```c
// 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-GReLoSSS ---
FT_Mgr::finalize();
MPI_Finalize();
}
```
```cpp
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()
    {
```
```c
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);
};
```
Source code of Matmult’s Domain I

```cpp
template<typename T_numtype, int N_rank>
class Matmult_Domain: public Domain<double, 2, Matmult_Domain>
{
private:
  blitz::Array<double, 2> data;

public:
  Matmult_Domain(int rank, int numprocs, TinyVector<int, 2> extent):
    // Call the base class constructor for proper initialization.
    Domain<double, 2, ::Matmult_Domain>(rank, numprocs, extent)
  {
    Domain_desc<2> dd = data_needed(rank, numprocs, 0);
    data.resize(dd.extent(1), dd.extent(2));
  }

  Domain_desc<2> data_needed(int rank, int numprocs, int step)
  {
    int size = this->get_extent(blitz::firstDim);
    int partition_size = size / numprocs;

    int dim1_lbound, dim1_rbound;

    // Compute boundaries
    ((dim1_lbound = (rank + step) * partition_size) == size)?
      dim1_lbound = 0, dim1_rbound = partition_size - 1:
      dim1_rbound = dim1_lbound + partition_size - 1;

    Domain_desc<2> domain_desc;
    domain_desc.set_bounds(1, dim1_lbound, dim1_rbound);
  }
```

Makassikis, Galtier, Vialle () A Skeletal-based Approach LAHMA 37 / 43
Source code of Matmult’s Domain II

```cpp
    domain_desc.set_bounds(2, 0, size - 1);

    return domain_desc; }

Domain_desc<2> data_posessed(int rank, int numprocs, int step)
{ return data_needed(rank, numprocs, step); }

double lget(blitz::TinyVector<int, 2> &coord)
{ return data(coord(0), coord(1)); }

void lset(blitz::TinyVector<int, 2> &coord, double e)
{ data(coord(0), coord(1)) = e; }

void swap(Matmult_Domain<double, 2> *md)
{ blitz::cycleArrays(this->data, md->get_data()); }
```
class FT_GReLoSSS_Skel // Fault-tolerant skeleton
{
    // Framework for iterator (internal definition)
    Skel_for_iter sfi;
    int it;
    Checkpoint c;

    // Double datastructure (two N-dimension arrays)
    Domain *V1, *V2;

    void execute()
    {
        // Routing plan init
        Routing_plan *rp = new Routing_plan(/*...*/);
        for (it = sfi.beg(); it != sfi.end(); it = sfi.next())
        {
            ft_compute(sfi);  // Computation phase
            rp->ft_comms(sfi); // Communication phase
            V1->swap(V2);      // Swap datastructures
            c.run(it);         // Possible checkpoint
        }
    }
};
Evaluation: Fault tolerance correctness

Current validation process

- Implementation of two classic parallel applications:
  - Matmult: dense matrix multiplication on a ring of processors
  - Jacobi: Jacobi relaxation
- Validation through extensive testing
Evaluation: Performance **without FT**

<table>
<thead>
<tr>
<th>Size of matrices</th>
<th>Number of nodes</th>
<th>$T_{\text{exec}}$ (seconds)</th>
<th>FT-GReLoSSS Framework Relative overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>OMPI</td>
<td>OMPI FT-GReLoSSS</td>
</tr>
<tr>
<td>$16384 \times 16384$</td>
<td>4</td>
<td>2027</td>
<td>2027</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1025</td>
<td>1027</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>522</td>
<td>526</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>274</td>
<td>277</td>
</tr>
<tr>
<td>$32768 \times 32768$</td>
<td>32</td>
<td>2107</td>
<td>2113</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1094</td>
<td>1103</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>597</td>
<td>609</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>352</td>
<td>362</td>
</tr>
<tr>
<td>$65536 \times 65536$</td>
<td>64</td>
<td>8405</td>
<td>8439</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>4444</td>
<td>4469</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2406</td>
<td>2445</td>
</tr>
</tbody>
</table>
Evaluation: Performance **without FT**

<table>
<thead>
<tr>
<th>Size of matrices</th>
<th>Number of nodes</th>
<th>$T_{exec}$ (seconds)</th>
<th>FT-GReLoSSS Framework Relative overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>OMPI</td>
<td>OMPI</td>
</tr>
<tr>
<td>16384 × 16384</td>
<td>4</td>
<td>2027</td>
<td>2027</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1025</td>
<td>1027</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>522</td>
<td>526</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>274</td>
<td>277</td>
</tr>
<tr>
<td>32768 × 32768</td>
<td>32</td>
<td>2107</td>
<td>2113</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1094</td>
<td>1103</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>597</td>
<td>609</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>352</td>
<td>362</td>
</tr>
<tr>
<td>65536 × 65536</td>
<td>64</td>
<td>8405</td>
<td>8439</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>4444</td>
<td>4469</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2406</td>
<td>2445</td>
</tr>
</tbody>
</table>

**Low Overheads <4%**
### Evaluation: Performance with FT and a single failure

**Recovery Overhead**

\[ T_{\text{recovery\_overhead}} = T_{\text{failure\_detection}} + T_{\text{recovery}} \]

**Experiment**

- We have only evaluated \( T_{\text{recovery}} \) after a single failure (no automatic failure detection mechanism yet)
- Measuring \( T_{\text{recovery}} \) proved more difficult than expected because of
  - System-level and application-level heterogeneity
  - Distributed setting

**Results**

- Both LAM/MPI and FT-GReLoSSS display negligible overheads < 1%
- DMTCP recovery failed on Intercell cluster
## Evaluation: Performance with FT and a single failure

### Recovery Overhead

\[ T_{\text{recovery\_overhead}} = T_{\text{failure\_detection}} + T_{\text{recovery}} \]

### Experiment

- We have only evaluated \( T_{\text{recovery}} \) after a single failure (no automatic failure detection mechanism yet)
- Measuring \( T_{\text{recovery}} \) proved more difficult than expected because of:
  - System-level and application-level heterogeneity
  - Distributed setting

### Results

- Both LAM/MPI and FT-GReLoSSS display negligible overheads < 1%
- DMTCP recovery failed on Intercell cluster