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

Constantinos Makassikis[2,3],
Virginie Galtier[1], Stéphane Vialle[1,2]

[1]SUPELEC - UMI-2958, Metz, France

[2]AlGorille INRIA Project Team, Nancy, France

[3]Université Henri Poincaré, Nancy, France

LAHMA, Orléans, France, 14 Dec. 2010

Research Context

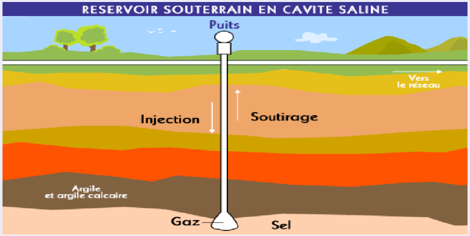
Extensible Machines

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

Intercell PC cluster (Supélec)

Demanding Applications

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


RESERVOIR SOUTERRAIN EN CAVITE SALINE
Puits
Vers le réseau
Injection
Soutirage
Argile et argile calcaire
Gaz
Sel

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 $\longrightarrow$ 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 $\longrightarrow$ 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

  $\longrightarrow$ There still is a risk to miss deadlines

  $\longrightarrow$ 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

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

- 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

MoLOToF: Definition and Aims

MoLOToF

- **Mo**del for **L**ow-**O**verhead **T**olerance **o**f **F**aults

What is MoLOToF ?

- A **set of rules** to develop fault-tolerant parallel applications
- Rules revolve around the concept of **fault-tolerant skeleton**

What are MoLOToF'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

```
FT_Seq_Skel
{
  FT_Loop
  {
    calculations()

    checkpoint()
  }
}
```

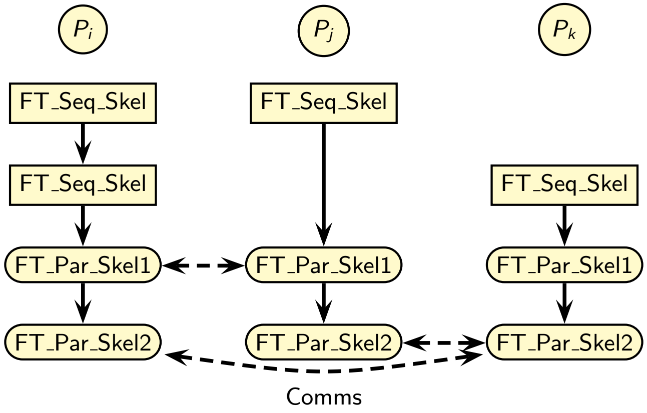
Sequential Skeleton

```
FT_Par_Skel
{
  FT_Loop
  {
    calculations()
    communications()
    checkpoint()
  }
}
```

Parallel Skeleton

MoLOToF: Skeleton-based application organization

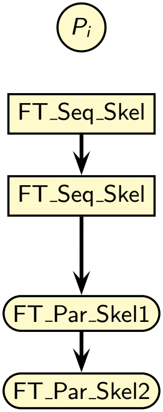
- A distributed application is made of several processes
- In MoLOToF,
  each process is a **succession of fault-tolerant skeletons**



MoLOToF: Save/Restore mechanics

Normal execution mode

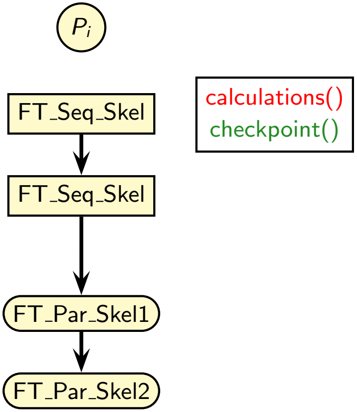
- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds


$$\begin{tikzcd}
P_i \\
\text{FT\_Seq\_Skel} \arrow[d] \\
\text{FT\_Seq\_Skel} \arrow[d] \\
\text{FT\_Par\_Skel1} \arrow[d] \\
\text{FT\_Par\_Skel2}
\end{tikzcd}$$

MoLOToF: Save/Restore mechanics

Normal execution mode

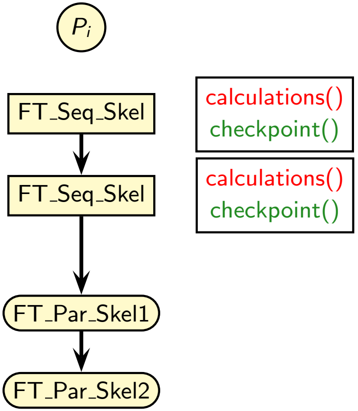
- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds



MoLOToF: Save/Restore mechanics

Normal execution mode

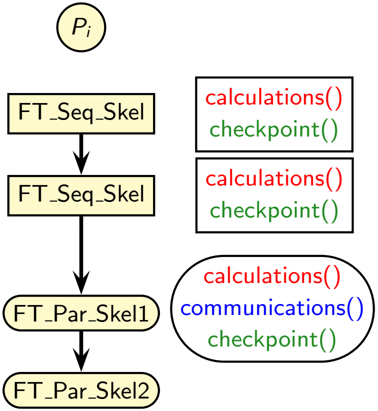
- Application and FT code
- A process saves itself when
  1. at checkpoint locations
  2. checkpoint condition holds



MoLOToF: Save/Restore mechanics

Normal execution mode

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

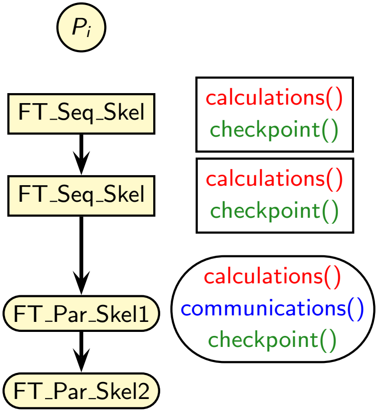


MoLOToF: Save/Restore mechanics

Normal execution mode

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

Suppose P_i checkpoints at iteration i

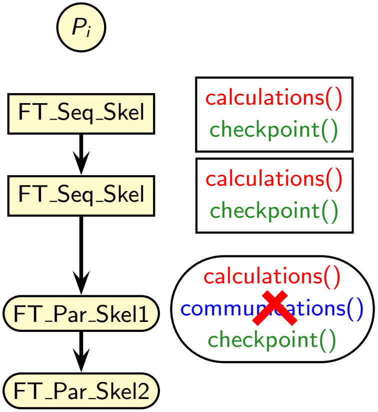


MoLOToF: Save/Restore mechanics

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

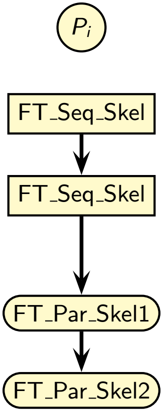


Suppose P_i fails at iteration $i+1$

MoLOToF: Save/Restore mechanics

Recovery execution mode

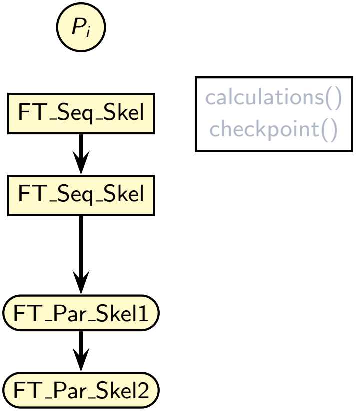
- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations



MoLOToF: Save/Restore mechanics

Recovery execution mode

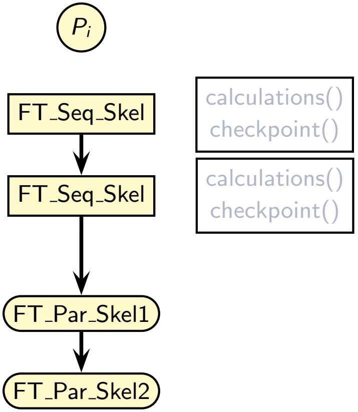
- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations



MoLOToF: Save/Restore mechanics

Recovery execution mode

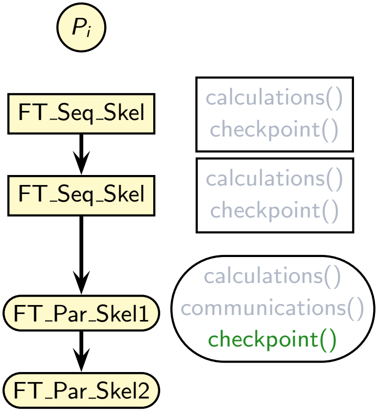
- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations



MoLOToF: Save/Restore mechanics

Recovery execution mode

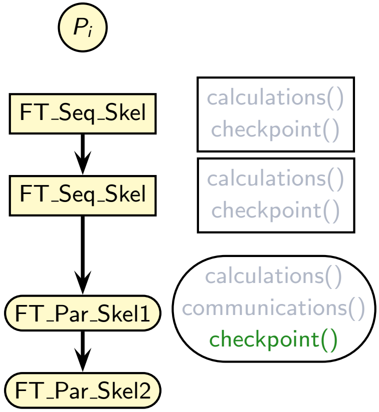
- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations



MoLOToF: Save/Restore mechanics

Recovery execution mode

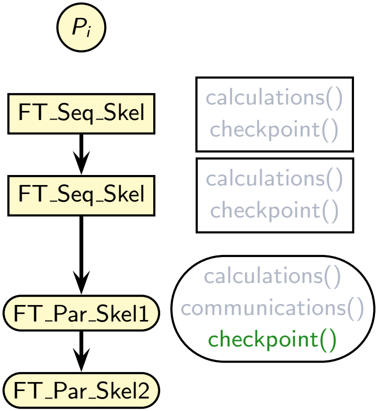
- **Recovery line** determination
- **Selective reexecution** to recover process context:
  1. Light operations reexecution
  2. Omission of already executed heavy operations
  3. Checkpoint data reload on "right" checkpoint location



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*



MoLOToF: Collaborations

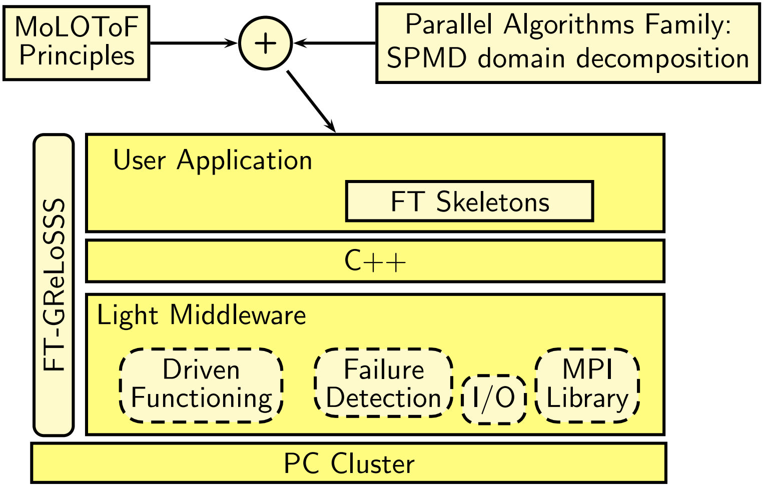
"Programmer–Framework" (require programmer's assistance)

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 ?

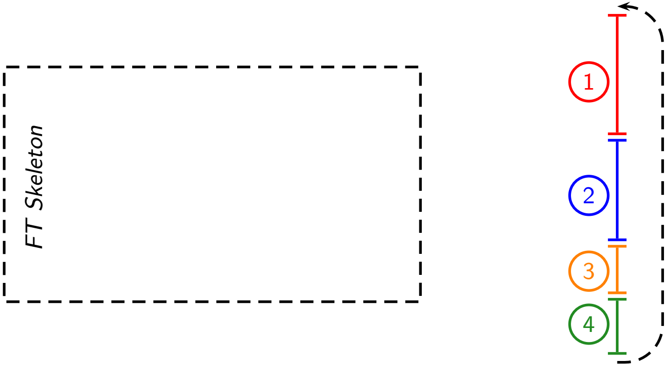
"Framework–Environment" (require environment's assistance)

- 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

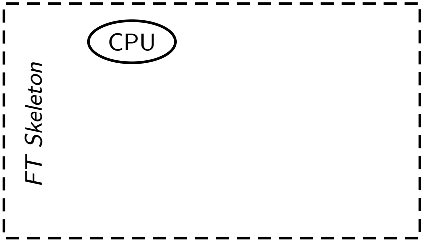


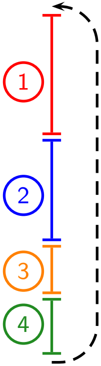
FT-GReLoSSS: Parallelization model



FT-GReLoSSS: Parallelization model

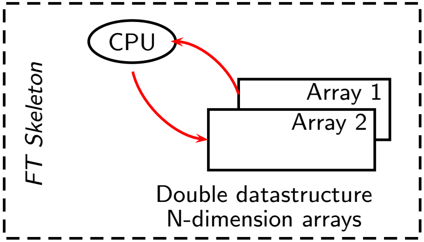
(1) COMPUTATION

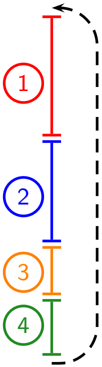




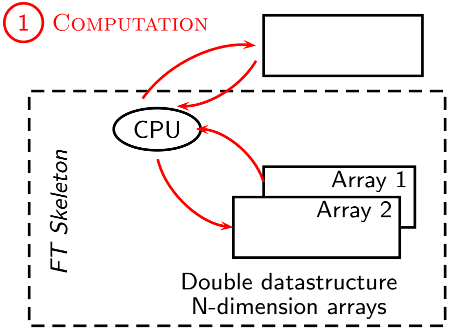
FT-GReLoSSS: Parallelization model

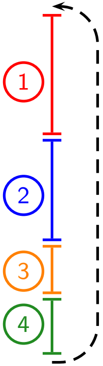
1 Computation



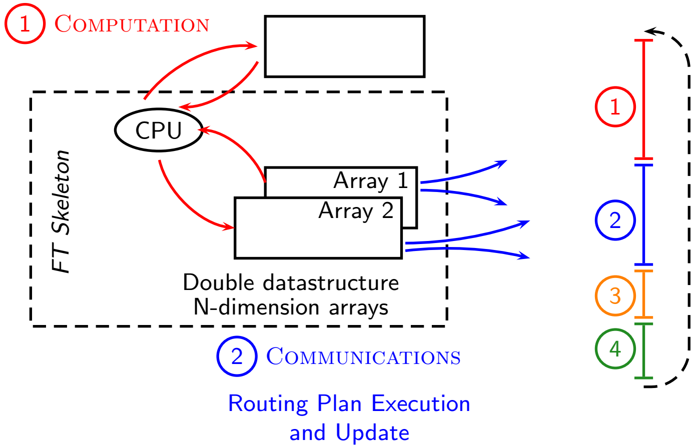


FT-GReLoSSS: Parallelization model



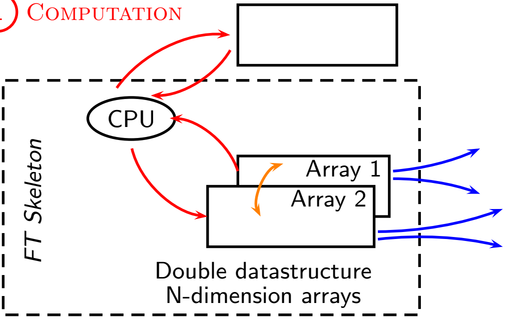


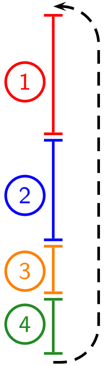
FT-GReLoSSS: Parallelization model



FT-GReLoSSS: Parallelization model

1 Computation



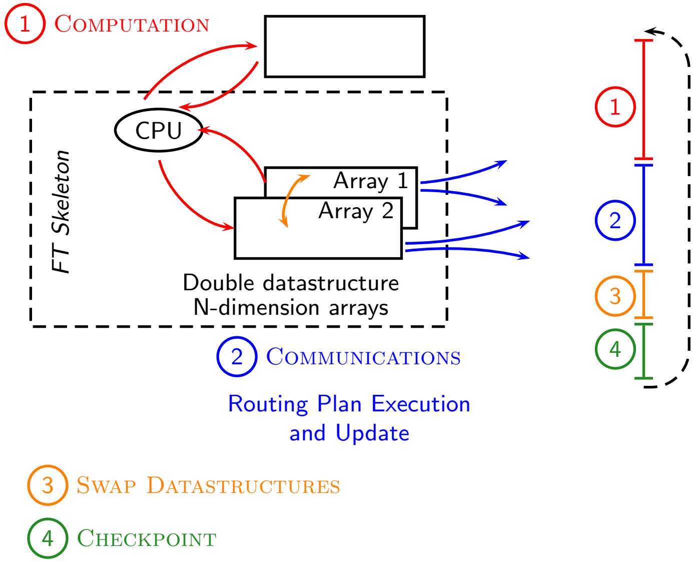


2 Communications

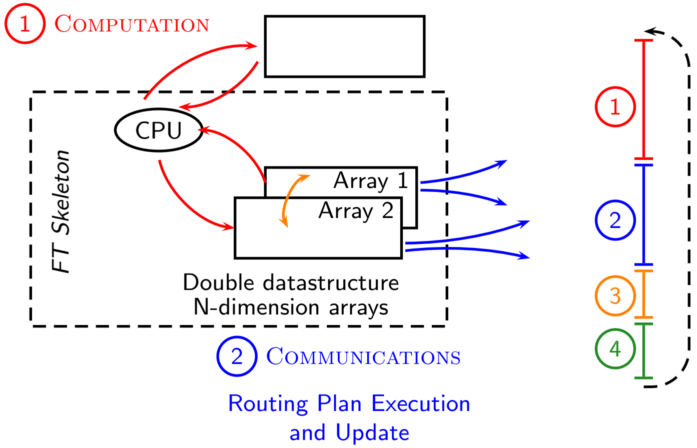
Routing Plan Execution and Update

3 Swap Datastructures

FT-GReLoSSS: Parallelization model



FT-GReLoSSS: Parallelization model

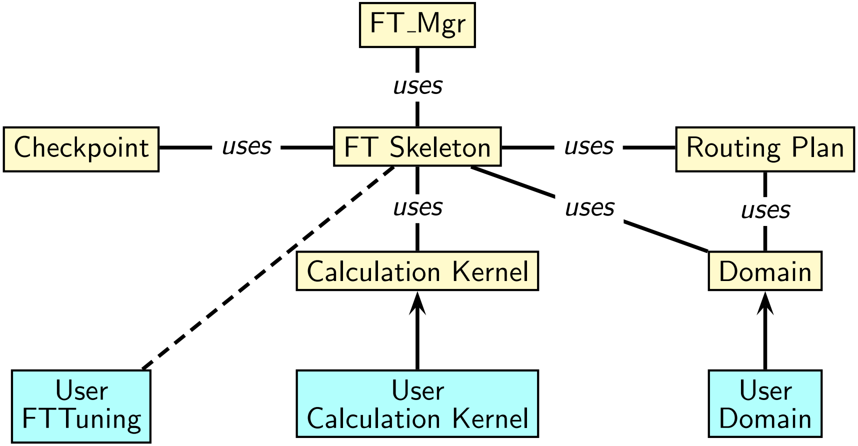


3 Swap Datastructures

4 Checkpoint

GReLoSSS family
Globally **Re**laxed between supersteps
Locally **S**trict **S**ynchronization **S**PMD
within superstep

FT-GReLoSSS: Relationships between concepts



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

Line Type	**Matmult v1**	**Matmult v2**	**Absolute Overhead**	**Relative Overhead (%)**
physical	258	295	+37 lines	+14.3
logical	168	186	+18 lines	+10.7

- 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

Line Type	**Matmult v1**	**Matmult v2**	**Absolute Overhead**	**Relative Overhead (%)**
physical	258	295	+37 lines	+14.3
logical	168	186	+18 lines	+10.7

- 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*

Individual matrix size	16384×16384	32768×32768	65536×65536
Total application size in RAM	$\sim$ 6 GiB	$\sim$ 24 GiB	$\sim$ 48 GiB
Total FT-GReLoSSS application checkpoint size	$\sim$ 4 GiB	$\sim$ 16 GiB	$\sim$ 32 GiB

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)

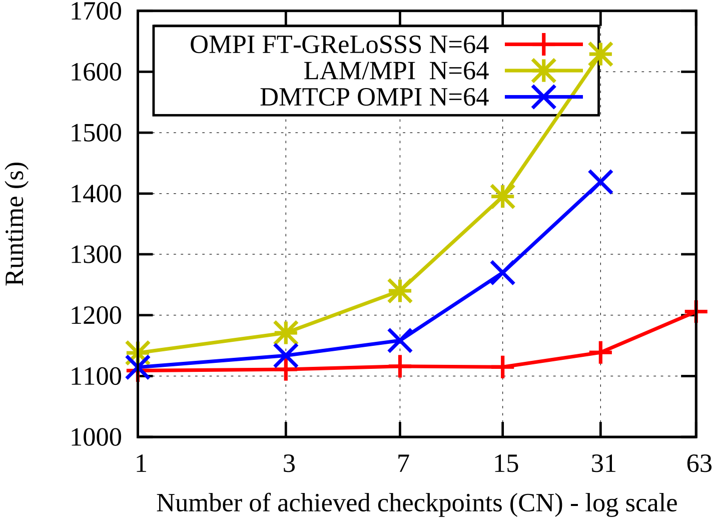
Benchmark Application : *Matmult*

Individual matrix size	16384×16384	32768×32768	65536×65536
Total application size in RAM	$\sim$ 6 GiB	$\sim$ 24 GiB	$\sim$ 48 GiB
Total FT-GReLoSSS application checkpoint size	$\sim$ 4 GiB	$\sim$ 16 GiB	$\sim$ 32 GiB

Lighter checkpoints thanks to Programmer–Framework collaborations

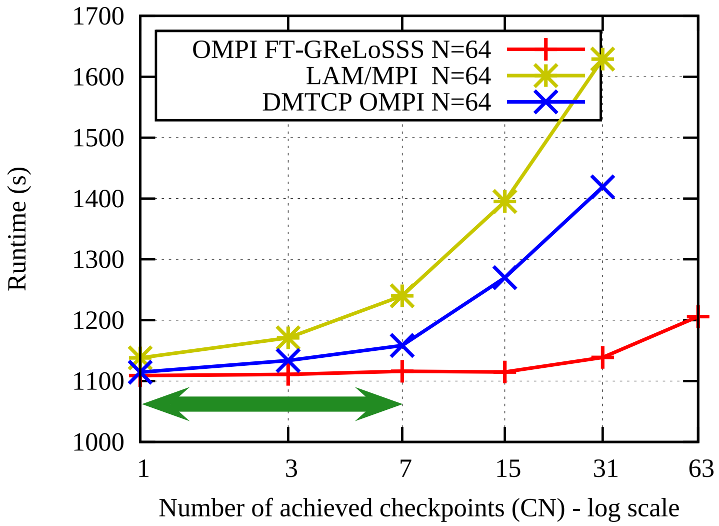
Evaluation: Performance **with FT** and **no failures**

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



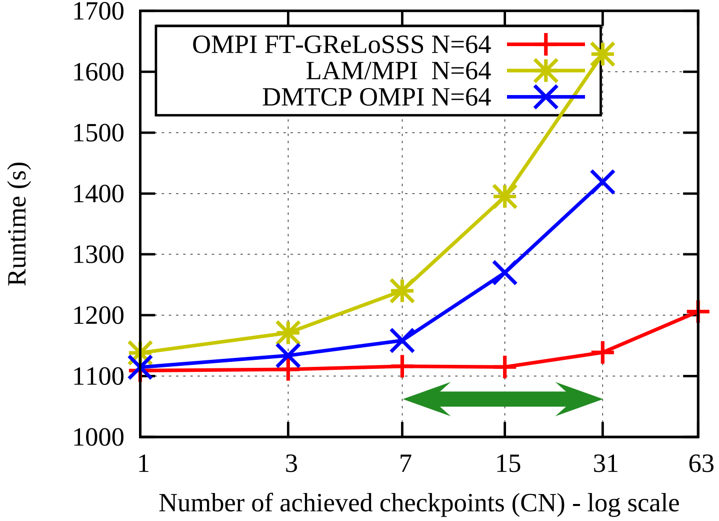
Evaluation: Performance **with FT** and **no failures**

- 32768 × 32768 (24 GiB) - 64 Nodes



Evaluation: Performance **with FT** and **no failures**

- 32768 × 32768 (24 GiB) - 64 Nodes



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

```
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 ----------- //

 //   + Chekpoint correctness: add result matrix to
 //     checkpoint
 //        + C->dataFirst(): address to the first element
```

Source code of Matmult's main II

```
  //                           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();

} // END OF  main()
```

Source code of Matmult's Calculation Kernel I

```
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()
 {
```

Source code of Matmult's Calculation Kernel II

```
    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

```
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);
```

Source code of Matmult's Domain II

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

    return domain_desc; }

  Domain_desc<2> data_possessed(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()); }
};
```

FT-GReLoSSS skeleton: fixed number of supersteps I

```
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**

Size of matrices	Number of nodes	T_{exec} (seconds) OMPI	T_{exec} (seconds) OMPI FT-GReLoSSS	FT-GReLoSSS Framework Relative overhead (%)
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**

Evaluation: Performance **without FT**

Size of matrices	**Number of nodes**	**T_{exec} (seconds)** OMPI	**T_{exec} (seconds)** OMPI FT-GReLoSSS	**FT-GReLoSSS Framework Relative overhead (%)**
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**

Low Overheads $<4\%$

Evaluation: Performance **with FT** and a **single failure**

Recovery Overhead

- $T_{recovery\_overhead} = T_{failure\_detection} + T_{recovery}$

Experiment

- We have only evaluated $T_{recovery}$ after a single failure (no automatic failure detection mechanism yet)
- Measuring $T_{recovery}$ 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

- $T_{recovery\_overhead} = T_{failure\_detection} + T_{recovery}$

Experiment

- We have only evaluated $T_{recovery}$ after a single failure (no automatic failure detection mechanism yet)
- Measuring $T_{recovery}$ 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