Optimized Coordinated Checkpoint/Rollback Protocol using a Dataflow Graph Model

Xavier Besseron and Thierry Gautier
{xavier.besseron | thierry.gautier}@imag.fr

Laboratoire d’Informatique de Grenoble
MOAIS Project

APRETAF Workshop, January 2009
Outline

1. Context
2. Fault-tolerance
3. Data Flow Graph model in Kaapi
4. Coordinated Checkpointing in Kaapi
5. Simulations
6. Perspectives
Outline

1. Context
2. Fault-tolerance
3. Data Flow Graph model in Kaapi
4. Coordinated Checkpointing in Kaapi
5. Simulations
6. Perspectives
Grid computing

What are grids?
- Clusters are computers connected by a LAN
- Grids are clusters connected by a WAN
- Heterogeneous (processors, networks, ...)
- Dynamic (failures, reservations, ...)

Aladdin – Grid’5000
- French experimental grid platform
- More than 4800 cores
- 9 sites in France
- 1 site in Brazil
- 1 site in Luxembourg
Fault-tolerance

Why fault-tolerance?

- Fault probability is high on a grid
- Split a large computation in shorter separated computations
- Dynamic reconfiguration
Outline

1. Context
2. Fault-tolerance
3. Data Flow Graph model in Kaapi
4. Coordinated Checkpointing in Kaapi
5. Simulations
6. Perspectives
Duplication-based protocols [Avizienis76][Wiesmann99]
Application execution is duplicated, spatially or temporally.

Log-based protocols [Alvisi98]
- Assume that the state of the system evolves according to non-deterministic events
- Non-deterministic events are logged in order to rollback from a previous saved checkpoint

Checkpoint/rollback protocols
Periodically save the local process state of the applications.
- Uncoordinated checkpointing [Randell75]
- Coordinated checkpointing [Chandy85]
- Communication-induced checkpointing [Baldoni97]
Why checkpoint/rollback protocol?

- Duplication protocols require too much resources [Wiesmann99] and a computation interruption can be tolerated.
- Logging protocols require too much resources (memory and bandwidth) with large communication applications [Elnozahy04].

Why coordinated checkpointing?

Coordinated checkpointing advantages:

- No domino effect [Elnozahy02]
- Low overhead towards application communications [Bouteiller03][Zheng04]
- Coordination overhead can be amortized using a suitable checkpoint period [Elnozahy04].
Application state

Global state

The global state of an application is composed of:

- the local state of all its processes;
- the state of all its communication channels.

Coherent global state

A coherent global state is a state that can happen during a correct execution of the application.
Classical coordinated checkpoint/rollback protocol

Two steps:

**Checkpoint step, during failure-free execution**
Coordinate all processes to checkpoint a coherent global state:
- Coordinate all the processes
- Flush communication channels between all processes
- Save the processes state

**Rollback step, to recover after a failure**
Global restart:
- Replace failed processes by new ones
- **All processes** restart from their last checkpoint
- Restart time is, in worst case, the checkpoint period
Challenging problems

How to improve performances of coordinated checkpoint/protocols?

- Reduce the synchronization cost [Koo87]
- Speed-up restart [Bouteiller03][Zheng04]
- Reduce lost computation time in case of fault
Applications: simulation of physical phenomena

Characteristics

- Iterative decomposition domain applications
- Large amount of data

Parallelization: static-scheduling

- Iterative applications ⇒ only schedule the loop “kernel”
- Large data ⇒ preserve locality
Applications: simulation of physical phenomena

Characteristics

- Iterative decomposition domain applications
- Large amount of data

Parallelization: static-scheduling

- Iterative applications $\Rightarrow$ only schedule the loop “kernel”
- Large data $\Rightarrow$ preserve locality
**Data Flow Graph**

**How it works?**

- Partition the one-iteration graph
- Generate communication tasks
- Distribute each sub-graph on all the processes
- Repeat the sub-graphs to iterate
Keypoint: abstract representation

The Data Flow Graph

Properties

- A task is the computational unit
- A process is composed of a (dynamic) sequence of tasks
- At any time, Kaapi allows to discover not yet executed tasks and their dependencies
- This abstract representation shows the future of the execution

The data flow graph representation is causally connected to the application execution.

Usage: analyze and transform the application state and behavior

- Schedule tasks (at any time)
- Checkpoint application state
Checkpoint step

Classical protocol checkpoint
Coordinate all processes to checkpoint a coherent global state:
- Coordinate all the processes
- Flush communication channels between all processes
- Save the processes state

CCK: differences with the classical protocol
Optimize the checkpoint step using the abstract representation of the execution (data flow graph):
- Partial flush: only between processes which communicates
- Increment checkpoint: save only modified data
Recovery: classical protocol vs CCK

Classical protocol restart

Global restart:
- Replace failed processes by new ones
- All processes restart from their last checkpoint
- Restart time is, in worst case, the checkpoint period

CCK protocol restart

Partial restart:
- Detect lost communications for the failed processes
- Find the strictly required computation set to make the global state coherent
- Schedule statically this task set
After a checkpoint

Non-failed process

Non-failed process

Non-failed process

Send task
Receive task
Non-executed task
Data
Dependency
Communication

Fault-Tolerance using Dataflow Graph
Xavier Besseron and Thierry Gautier
A process failed

Send task
Receive task
Non-executed task
Executed task
Data
Dependency
Communication

Xavier Besseron and Thierry Gautier
Fault-Tolerance using Dataflow Graph
Incoherent application state
Lost communications

Failed process

Non-failed process

Non-failed process

\( C_{\text{lost}} \)

Send task

Receive task

Non-executed task

Executed task

Task to re-execute

Data

Dependency

Communication
Communications to replay

Failed process

Non-failed process

Non-failed process

Send task
Receive task
Non-executed task
Executed task
Task to re-execute
Data
Dependency
Communication

Xavier Besseron and Thierry Gautier
Fault-Tolerance using Dataflow Graph
Tasks to re-execute

Failed process

Non-failed process

Non-failed process

Send task
Receive task
Non-executed task
Executed task
Task to re-execute
Data
Dependency
Communication

Xavier Besseron and Thierry Gautier
Fault-Tolerance using Dataflow Graph
Recovery: classical protocol vs CCK

**Classical protocol restart**

- Last checkpoint
- Past of the execution
- Processes
- Failure
- Next checkpoint
- Execution

**CCK protocol restart**

- Last checkpoint
- Past of the execution
- Processes
- Failure
- Next checkpoint
- Execution
Recovery: classical protocol vs CCK

Classical protocol restart

CCK protocol restart
Recovery: classical protocol vs CCK

Classical protocol restart

- Last checkpoint
- Past of the execution
- End of recovery
- Execution
- Processes

CCK protocol restart

- Last checkpoint
- Past of the execution
- End of recovery
- Execution
- Processes
Recovery: classical protocol vs CCK

Classical protocol restart

<table>
<thead>
<tr>
<th>Last checkpoint</th>
<th>Past of the execution</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

End of recovery

Execution

CCK protocol restart

<table>
<thead>
<tr>
<th>Last checkpoint</th>
<th>Past of the execution</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

End of recovery

Execution

Xavier Besseron and Thierry Gautier

Fault-Tolerance using Dataflow Graph

20/33
## Recovery: Cost analysis

### Classical protocol restart

<table>
<thead>
<tr>
<th>Required work to recover:</th>
<th>( W_{\text{recovery}}^{\text{std}} = O(N \cdot \tau) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart time on ( N ) processes:</td>
<td>( T_{\text{restart}}^{\text{std}} = O(\tau) )</td>
</tr>
</tbody>
</table>

### CCK protocol restart

<table>
<thead>
<tr>
<th>Required work to recover:</th>
<th>( W_{\text{recovery}}^{\text{cck}} = O(N_{\text{failed}} \cdot \tau + \varepsilon_{\text{application,} \tau}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart time on ( N ) processes:</td>
<td>( T_{\text{restart}}^{\text{cck}} = O\left(\frac{N_{\text{failed}} \cdot \tau + \varepsilon_{\text{application,} \tau}}{N}\right) )</td>
</tr>
</tbody>
</table>

We have to add the CCK-recovery overhead:

\( O(N \cdot K) \) messages + \( O(|G|) \) in time + data distribution cost

\( K \) is an application dependent constant that represent the neighbor number.
Outline

1. Context
2. Fault-tolerance
3. Data Flow Graph model in Kaapi
4. Coordinated Checkpointing in Kaapi
5. Simulations
6. Perspectives
Simulations: case study

Application

- Jacobi method on a 3D-domain
- $2,048^3$ domain (64 GB)
- Split in $64^3$ subdomains (32 KB each)
- Subdomain update computed in 10 ms

Scenario

- One process failed
- Simulation of the restart in worst case
- $\Rightarrow$ % of tasks to re-execute ($W_{recovery}^{cck} / W_{recovery}^{std}$)
- $\Rightarrow$ Involved processes
CCK restart: checkpoint period influence

- 1,024 processors, ie 256 subdomains (64 MB) per process
- one iteration last about 2.5 seconds

For a 60-seconds period, the estimated restart time is:
- 60 seconds with the classical protocol
- 3.6 seconds with CCK (if totally parallelized)
CCK restart: process number influence

- Graph showing the number of involved processes over process number for different checkpoint periods.
- Graph showing the percentage with respect to the classical protocol over process number for different checkpoint periods.

Checkpoint period: 5 s, 10 s, 25 s, 50 s, 100 s.
Local re-ordering

Default execution order

Process 1

Process 2
Local re-ordering

Default execution order

Process 1

Process 2
Local re-ordering

Default execution order

Process 1

Process 2
Local re-ordering

Default execution order

Process 1

Process 2
Local re-ordering

With local re-ordering
Local re-ordering

With local re-ordering
Local re-ordering

With local re-ordering

[Diagram showing a dataflow graph with processes 1 and 2, illustrating local re-ordering with blue arrows.]
Local re-ordering

With local re-ordering
Local re-ordering

With local re-ordering

[Diagram showing the re-ordering process between Process 1 and Process 2]
Local re-ordering

With local re-ordering

Xavier Besseron and Thierry Gautier
Local re-ordering

With local re-ordering

Process 1

Checkpoint

Process 2
CCK restart: local re-ordering influence

---

Xavier Besseron and Thierry Gautier

Fault-Tolerance using Dataflow Graph

27/33
Perspectives

Performance guarantees for failure-free executions

The goal is to optimize the protocol parameters:
- Interval delay between checkpoint events
- Checkpoint server number and mapping

Dynamic reconfiguration

Adding or removing nodes requires to re-schedule statically
- Checkpoint to get a coherent global state
- Schedule statically for the new node number
- Resume the execution
Thanks for your attention

Questions?
The application is described as a data flow graph.

API
- Global address space
- Independent of the number of processors
- Data (Shared<...>): declares an object in the global memory
- Tasks (Fork<...>): creates a new task that may be executed in concurrence with other tasks
- Access mode: given by the task: Read, Write, Exclusive, Concurrent write

```cpp
Shared<Matrix> A;
Shared<double> B;
Fork<Task>() (A,B);
```
Optimized CCK restart

Failed process

Non-failed process

Non-failed process

Send task
Receive task
Non-executed task
Executed task
Task to re-execute
Data
Data in memory
Dependency
Communication

Send task
Receive task
Non-executed task
Executed task
Task to re-execute
Data
Data in memory
Dependency
Communication

Failed process
Non-failed process
Non-executed task
Data
2
3
4
5
6
Communication
Dependency
1
Send task
Receive task
Failed process
Executed task
Task to re-execute
Data in memory
T
T
T
T

Fault-Tolerance using Dataflow Graph
First experiments: 3D-domain decomposition

Preliminary results, Kaapi vs MPICH:

![Bar chart showing mean time for an iteration (s) vs number of nodes for Kaapi and MPICH in 1 and 2 clusters.](chart.png)