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

L I G

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA

APRETAF Workshop, January 2009

Outline

Outline

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

Grid computing

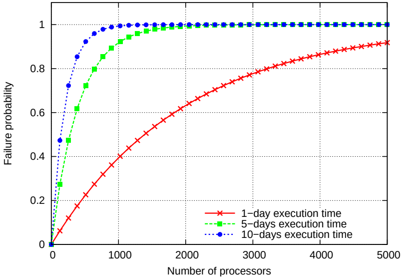
What are grids?

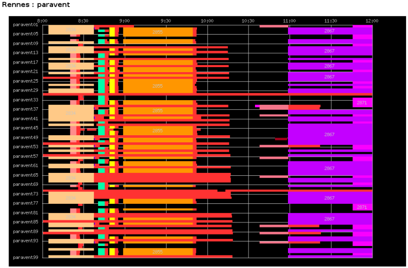
- Clusters are computers connected by a LAN
- Grids are clusters connected by a WAN
- Heterogeneous (processors, networks, ...)
- Dynamic (failures, reservations, ...)

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

Fault-tolerance survey [Elnozahy02]

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]

Checkpoint/rollback protocols

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

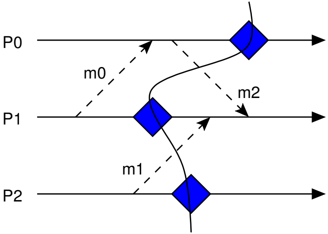
Global state

The global state of an application is composed of:

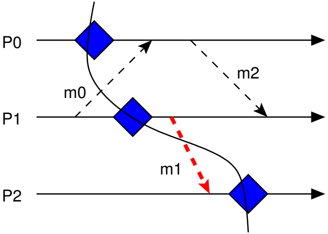
- the local state of all its processes;
- the state of all its communication channels.

Coherent global state

A coherent global state is a state than can happen during a correct execution of the application.



Coherent global state



Incoherent global state

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

Outline

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

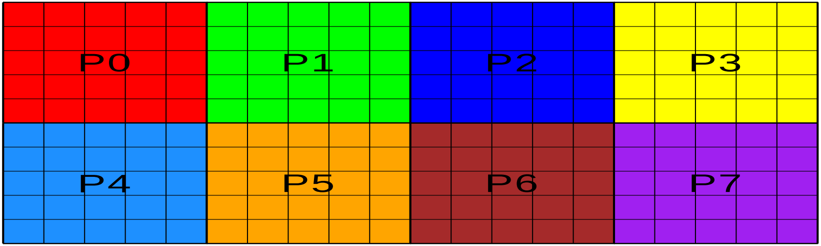
Applications: simulation of physical phenomena

Characteristics

- Iterative decomposition domain applications
- Large amount of data

Parallelization: static-scheduling

- Iterative applications ⇒ only schedule the loop “kernel”
- Large data ⇒ preserve locality



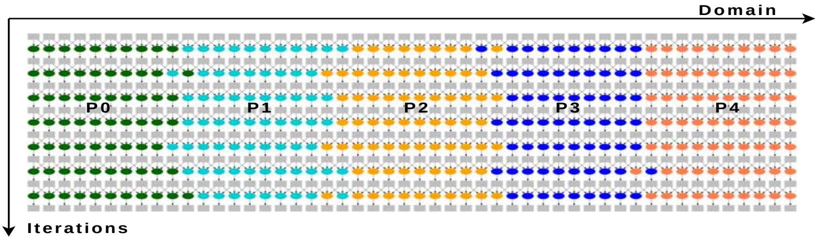
Applications: simulation of physical phenomena

Characteristics

- Iterative decomposition domain applications
- Large amount of data

Parallelization: static-scheduling

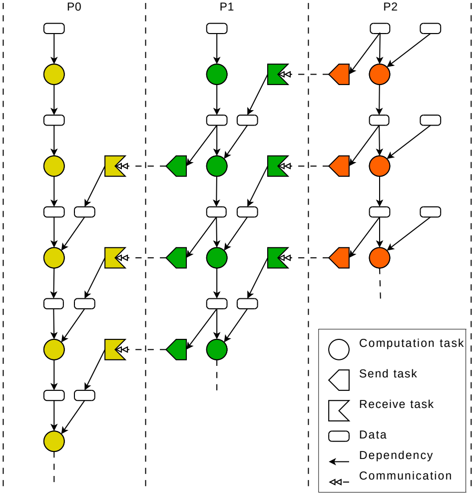
- Iterative applications ⇒ only schedule the loop “kernel”
- Large data ⇒ preserve locality



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

Outline

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

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:

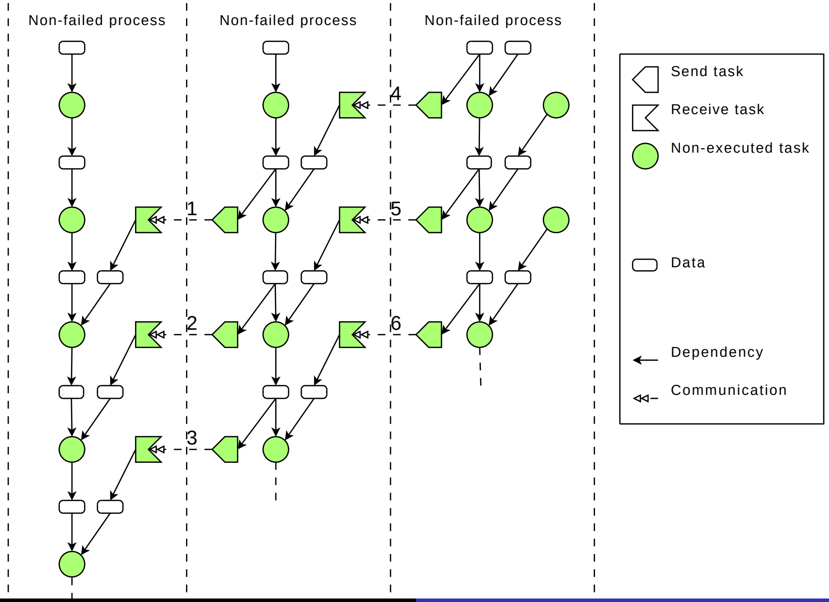
- Replace failed processes by new ones
- **All processes** restart from their last checkpoint
- Restart time is, in worst case, the checkpoint period

CCK protocol restart

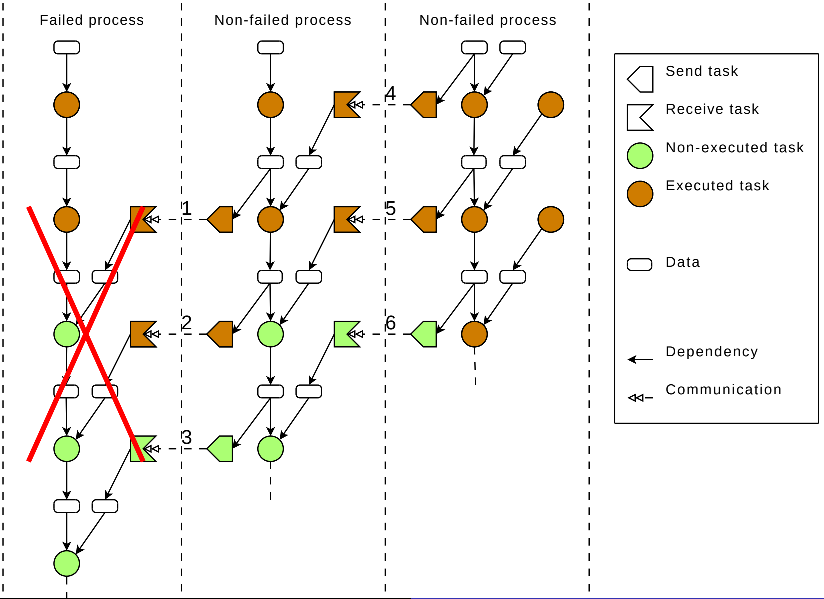
Partial restart:

- Detect lost communications for the failed processes
- Find the strictly required computation set to make the global state coherent
- Schedule statically this task set

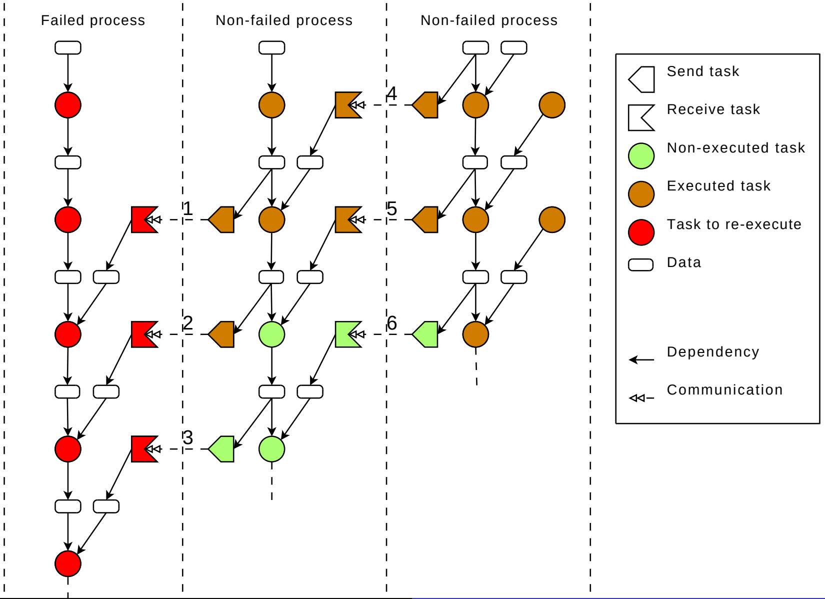
After a checkpoint



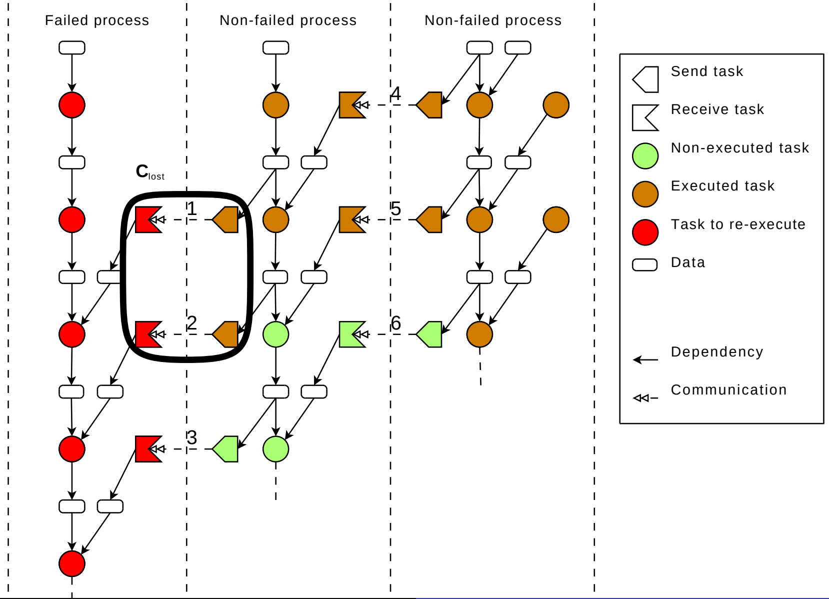
A process failed



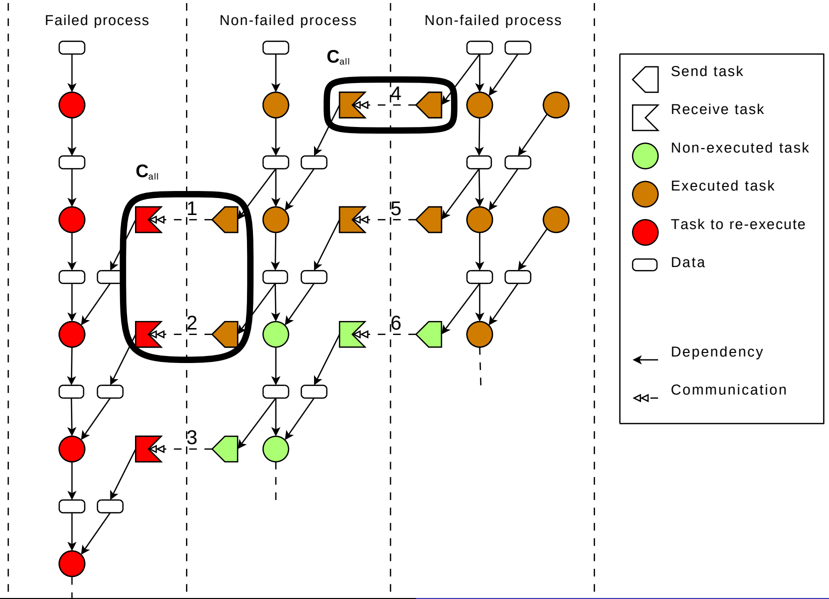
Incoherent application state



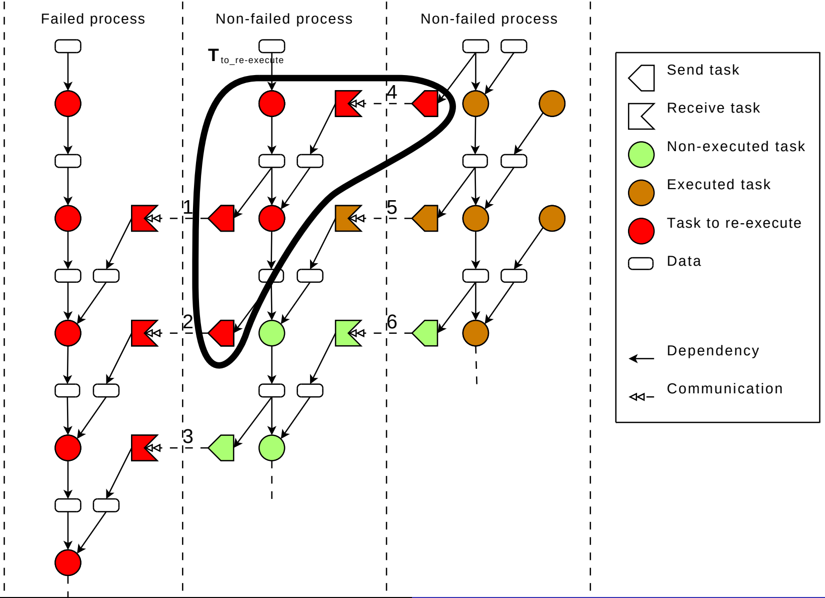
Lost communications



Communications to replay

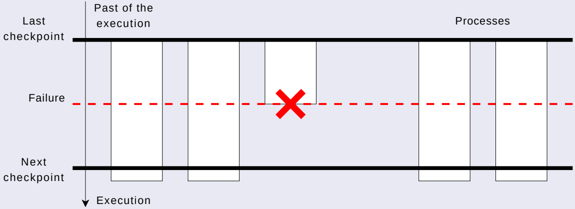


Tasks to re-execute

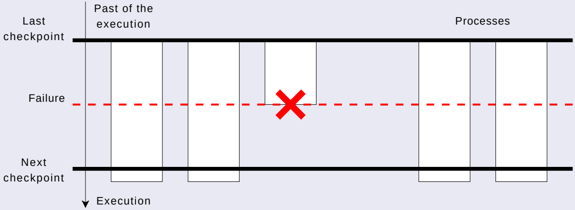


Recovery: classical protocol vs CCK

Classical protocol restart

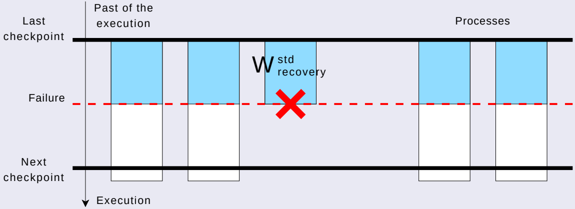


CCK protocol restart

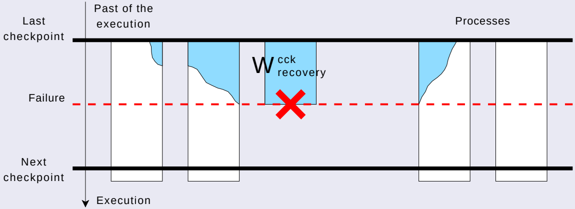


Recovery: classical protocol vs CCK

Classical protocol restart

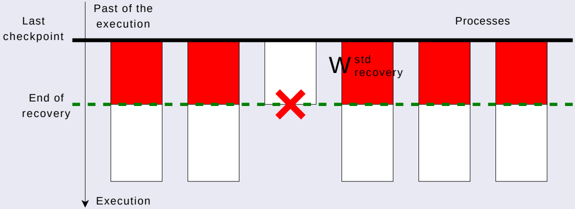


CCK protocol restart

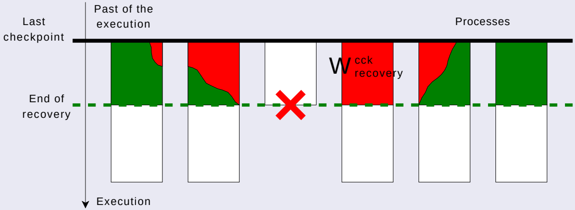


Recovery: classical protocol vs CCK

Classical protocol restart

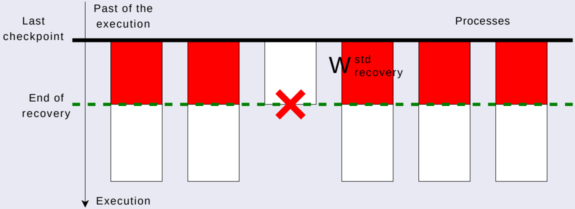


CCK protocol restart

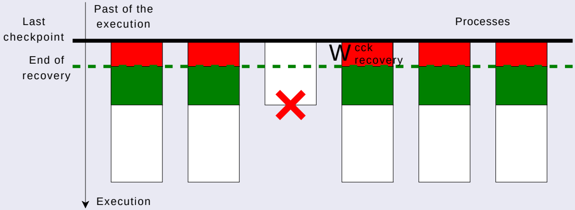


Recovery: classical protocol vs CCK

Classical protocol restart



CCK protocol restart



Recovery: Cost analysis

Classical protocol restart

Required work to recover: $W^{std}_{recovery} = O(N \cdot \tau)$

Restart time on N processes: $T^{std}_{restart} = O(\tau)$

CCK protocol restart

Required work to recover: $W^{cck}_{recovery} = O(N_{failed} \cdot \tau + \varepsilon_{application,\tau})$

Restart time on N processes: $T^{cck}_{restart} = O(\frac{N_{failed} \cdot \tau + \varepsilon_{application,\tau}}{N})$

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

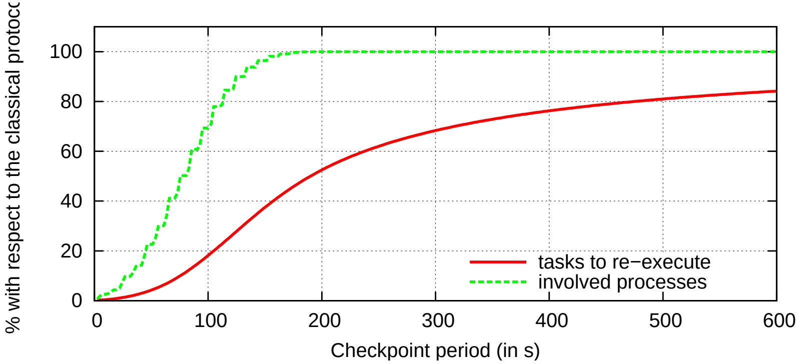
- Jacobi method on a 3D-domain
- $2,048^3$ domain (64 GB)
- Split in 64^3 subdomains (32 KB each)
- Subdomain update computed in 10 ms

Scenario

- One process failed
- Simulation of the restart in worst case
- $\Rightarrow$ % of tasks to re-execute ($W^{cck}_{recovery} / W^{std}_{recovery}$)
- $\Rightarrow$ Involved processes

CCK restart: checkpoint period influence

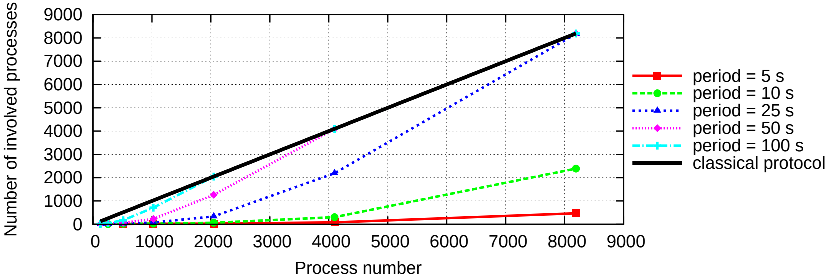
- 1,024 processors, ie 256 subdomains (64 MB) per process
- one iteration last about 2.5 seconds

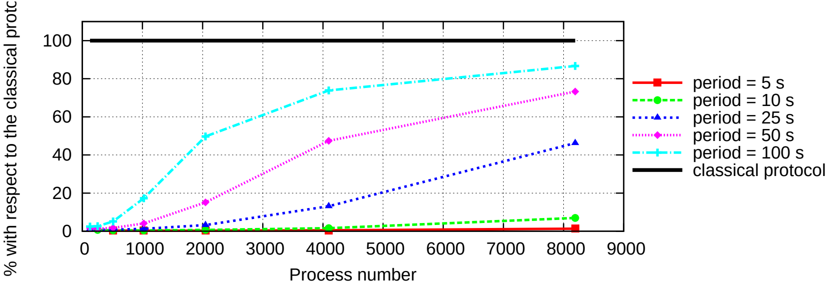


For a 60-seconds period, the estimated restart time is:

- 60 seconds with the classical protocol
- 3.6 seconds with CCK (if totally parallelized)

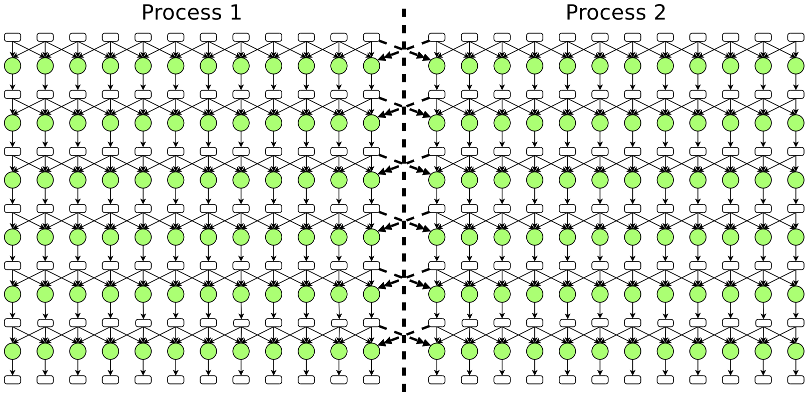
CCK restart: process number influence





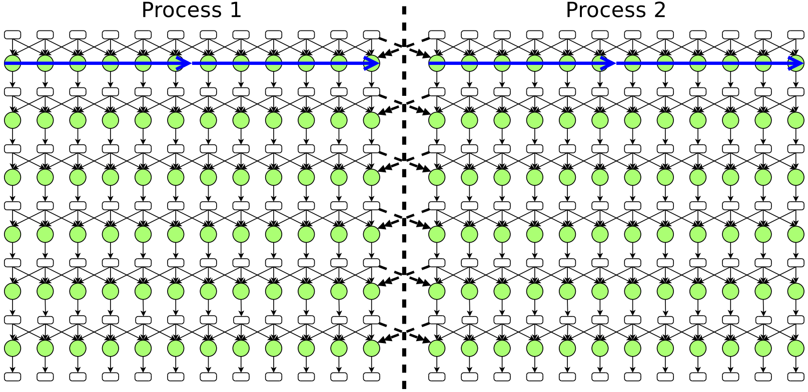
Local re-ordering

Default execution order



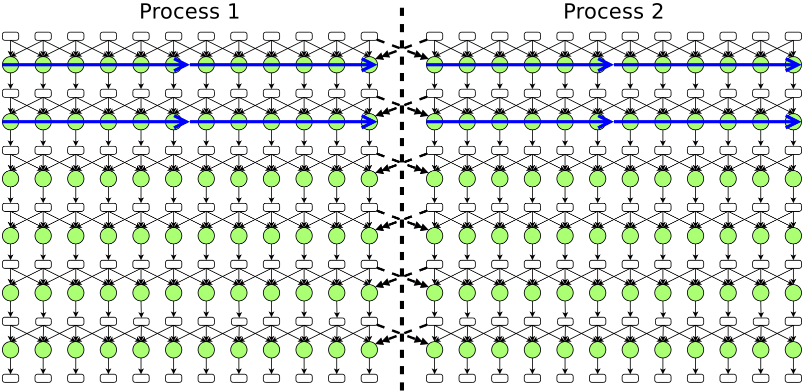
Local re-ordering

Default execution order



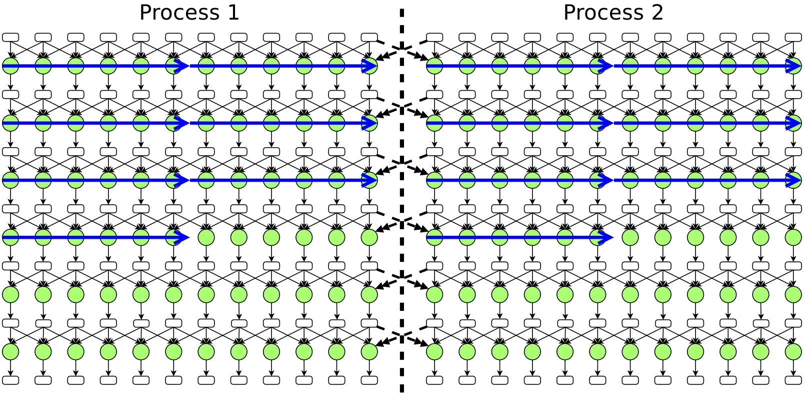
Local re-ordering

Default execution order



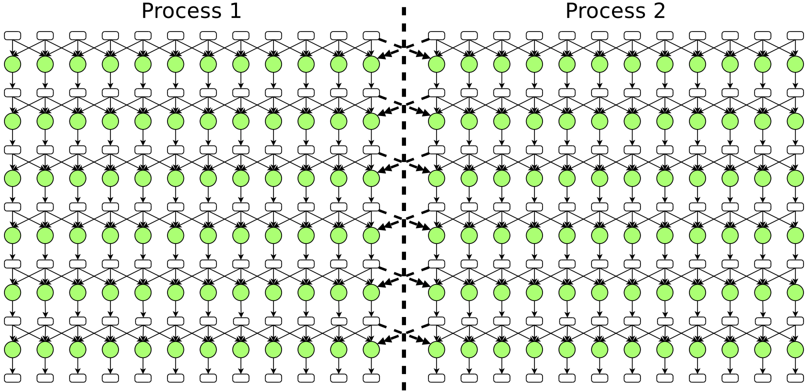
Local re-ordering

Default execution order



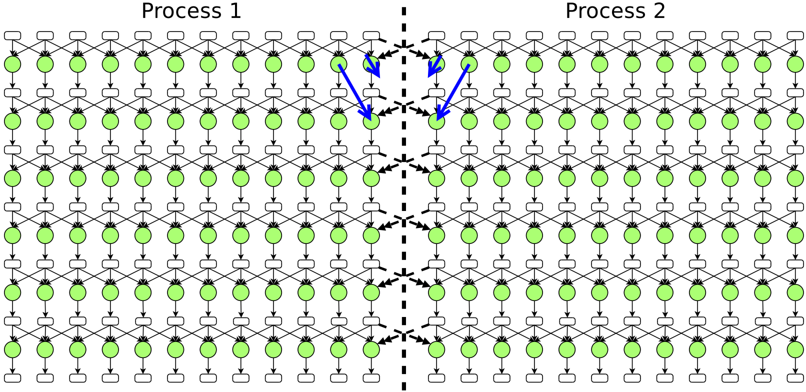
Local re-ordering

With local re-ordering



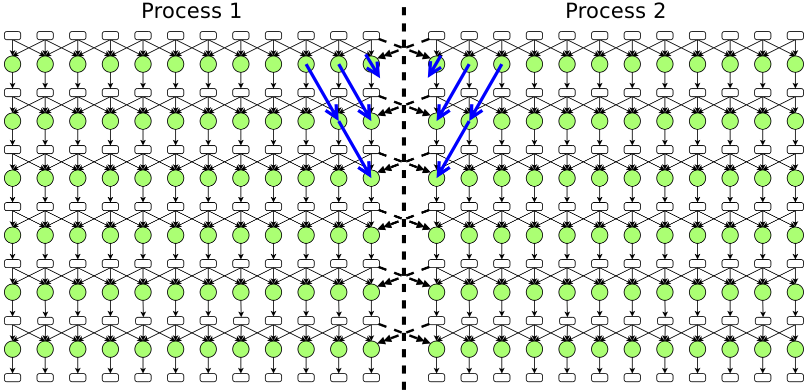
Local re-ordering

With local re-ordering



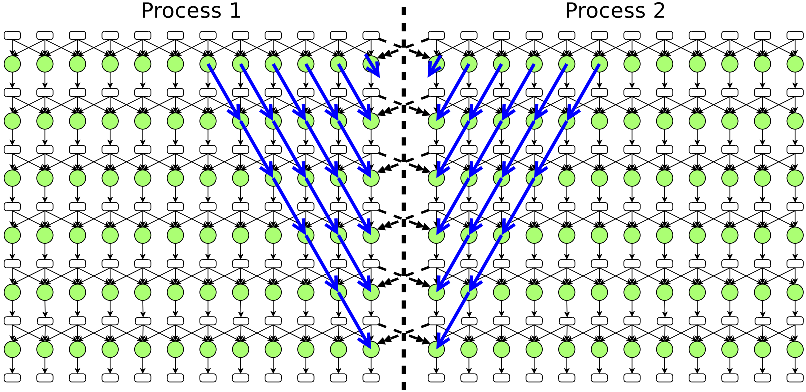
Local re-ordering

With local re-ordering



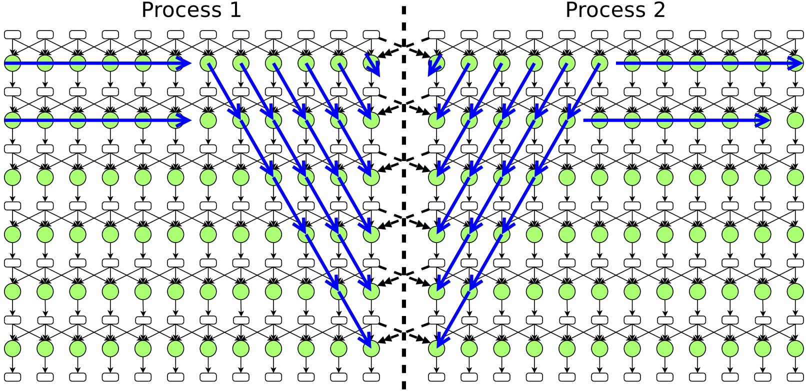
Local re-ordering

With local re-ordering



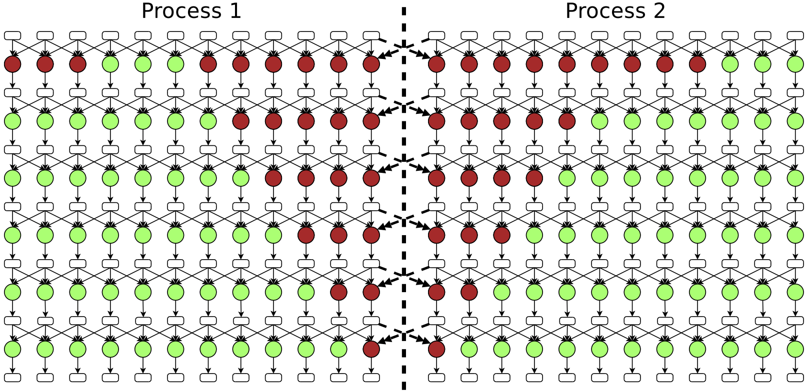
Local re-ordering

With local re-ordering



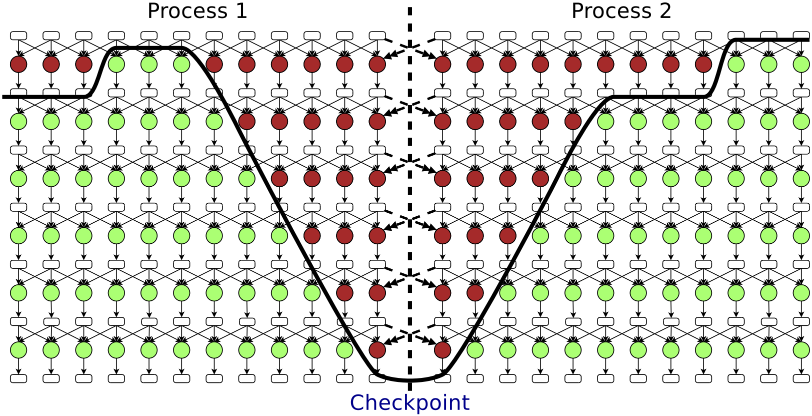
Local re-ordering

With local re-ordering

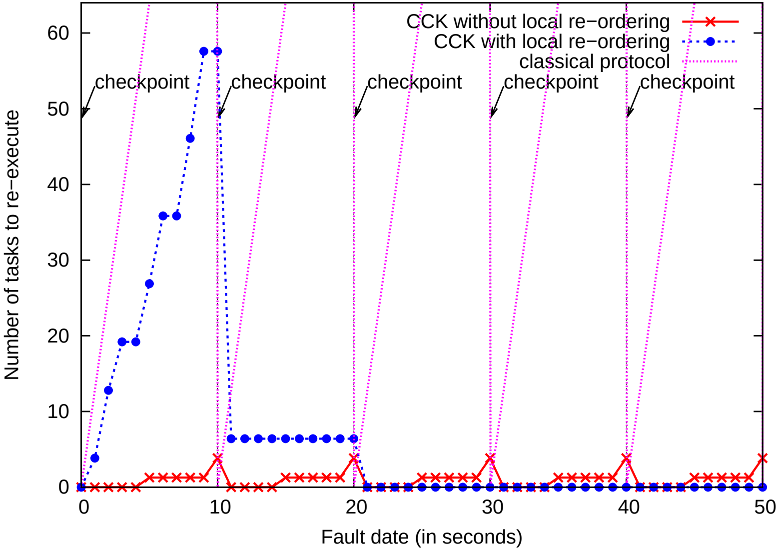


Local re-ordering

With local re-ordering



CCK restart: local re-ordering influence



Outline

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?

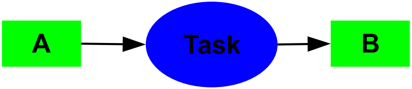
Kaapi parallel programming model

The application is described as a data flow graph.

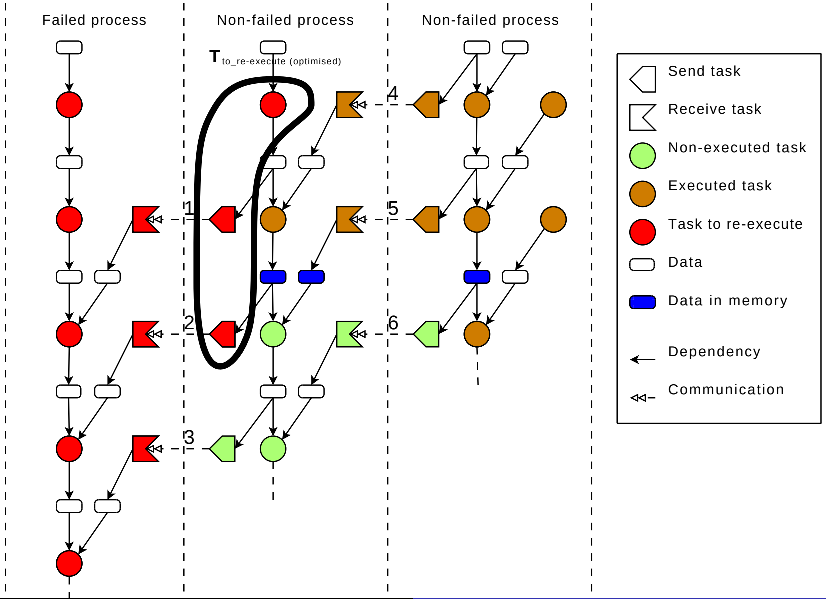
API

- 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

```
Shared<Matrix> A;
Shared<double> B;
Fork<Task>() (A,B);
```



Optimized CCK restart



First experiments: 3D-domain decomposition

Preliminary results, Kaapi vs MPICH: