Research Project Proposal: Fault mitigation and tolerance for MPI applications

Roberto Rocco
roberto2.rocco@mail.polimi.it
CSE Track

POLITECNICO MILANO 1863

HONOURS PROGRAMME HP-SR in Information Technology

Overview

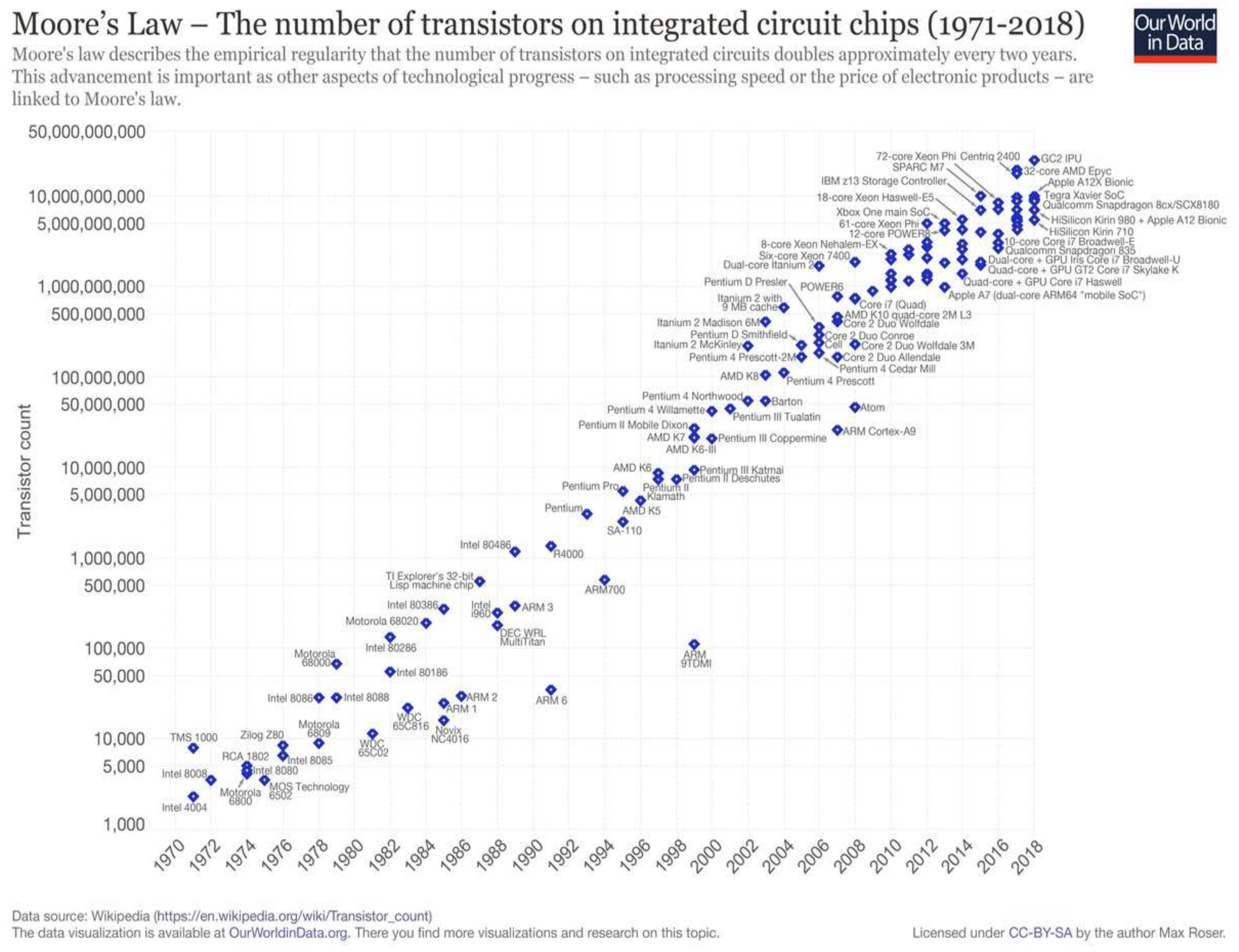
- Introduction to the research topics
- Problem analysis
- Previous efforts
- Proposed approaches
- Evaluation of the research

High Performance Computing

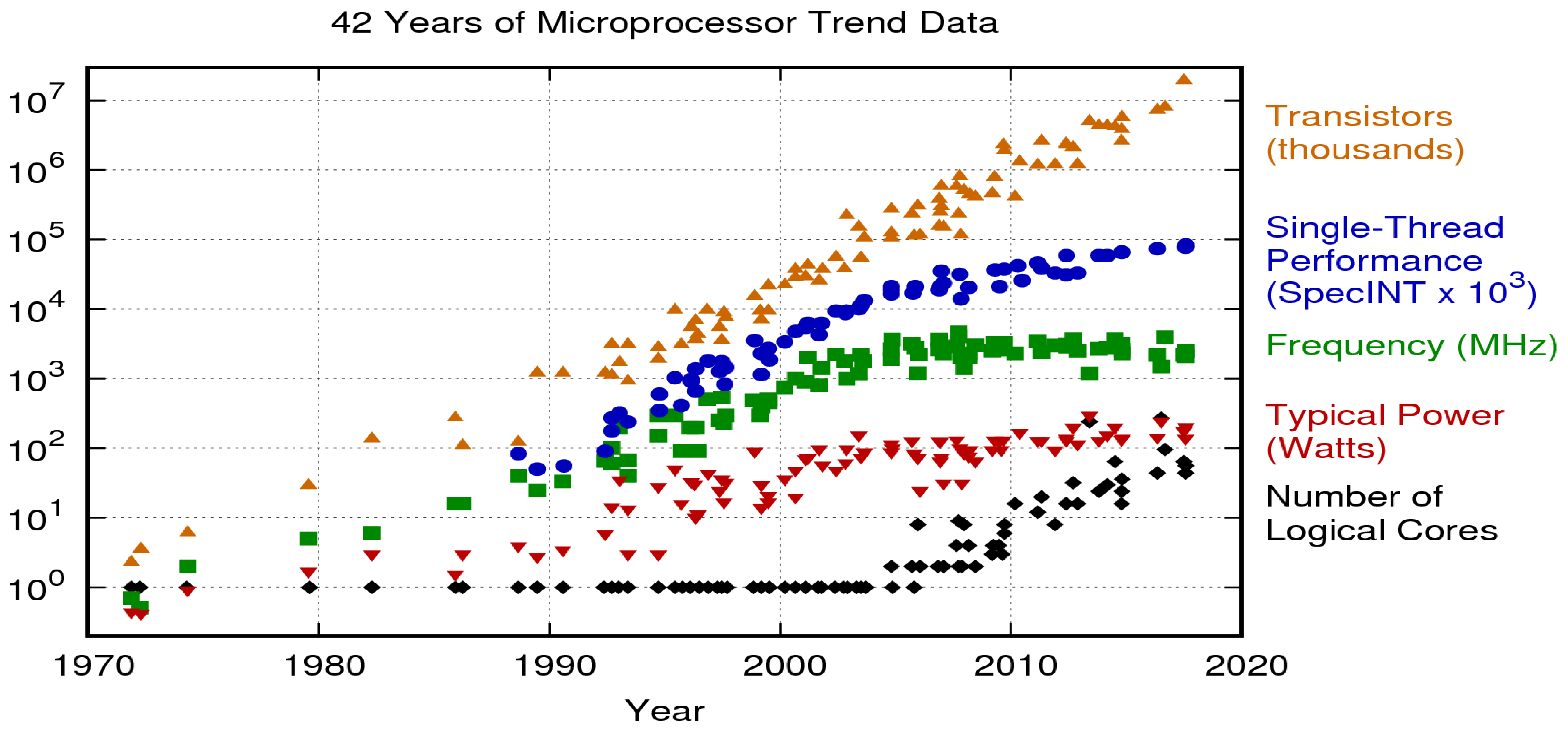
- Field of computer architectures aimed at reaching the highest computation capabilities.
- Performance is core, no trade-offs with power consumption, space, costs.
- Continuous evolution.

Moore’s Law

- *The number of transistors incorporated in a chip will approximately double every 24 months*
- Empirical relationship, used to define evolution of computation capabilities over the years.
- Adopted by manufacturers as target for the production



Moore’s Law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Parallelization

- 1 Flop/s = 1 floating point operation per second;

- From single core
  Intel Pentium 4 - 1 core -~3 GFlop/s (f = 3GHz)

- Multi cores in the same CPU chip
  Intel i9 9980XE –18 cores -~5-10 TFlop/s
  Intel Xeon PHI 7290 –72 cores –3 TFlop/s (f = 1.5-1.7 GHz)

- Multi computer (clusters)
  Summit –IBM –2,397,824 cores –200,794 TFlop/s

An example

- **Exascalate4Cov**: systematic analysis of proteins that allow virus replication to virtually test pharmaceutical molecules to stop virus propagation.
- Born from a collaboration between 18 research centers (including PoliMi) in 7 different nations.
- Speeds up validation steps, earlier drug production.
- On a normal computer it would take 4 months per protein; database contains half billion of them.
- The HPC realization is able to evaluate 3 million proteins per second.

Communication

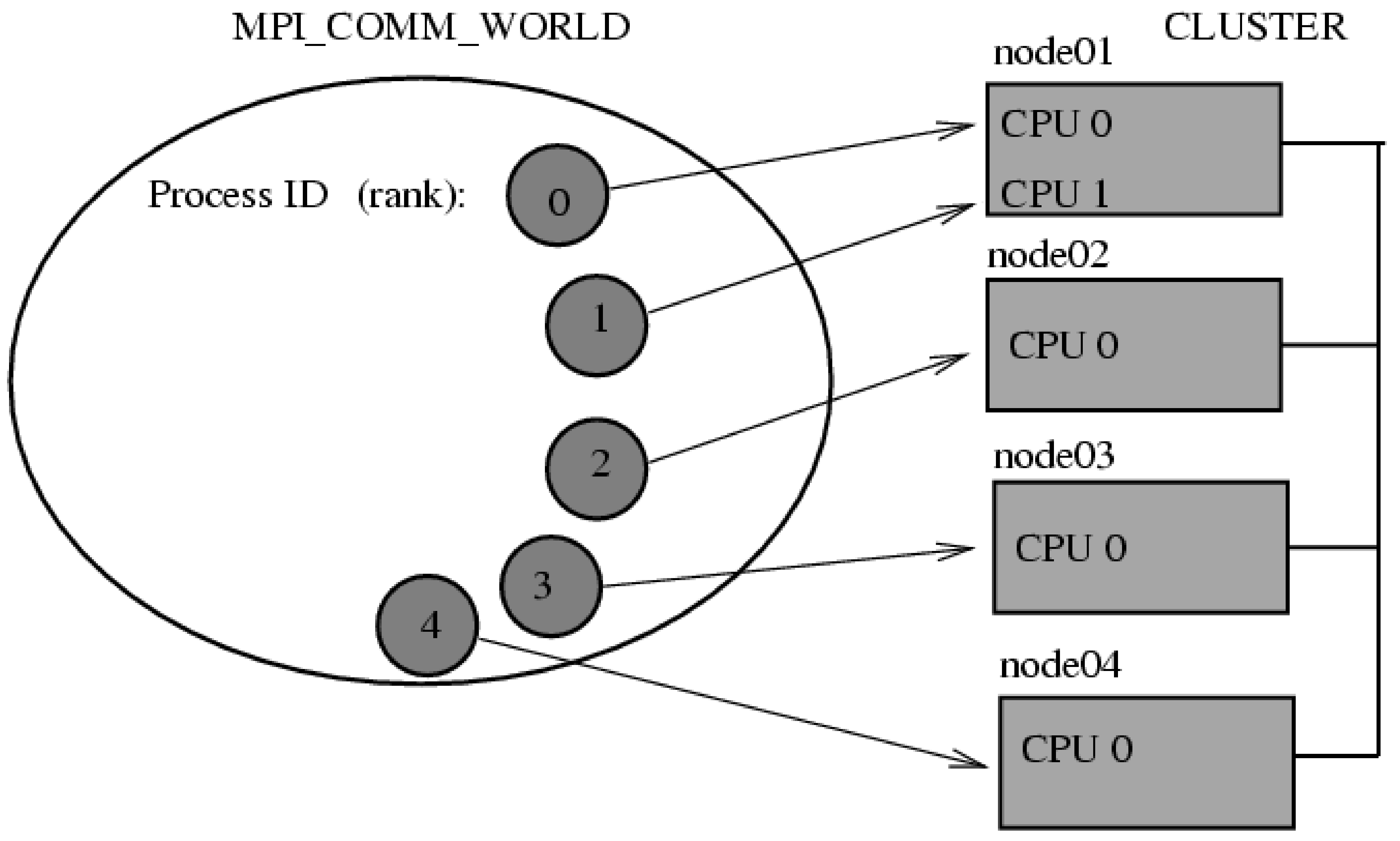
- Running code on multiple computers requires communication:
  - Data exchange
  - Code exchange
  - Coordination

- Message Passing Interface (MPI) is the de-facto standard for intra-process communication in HPC environment.

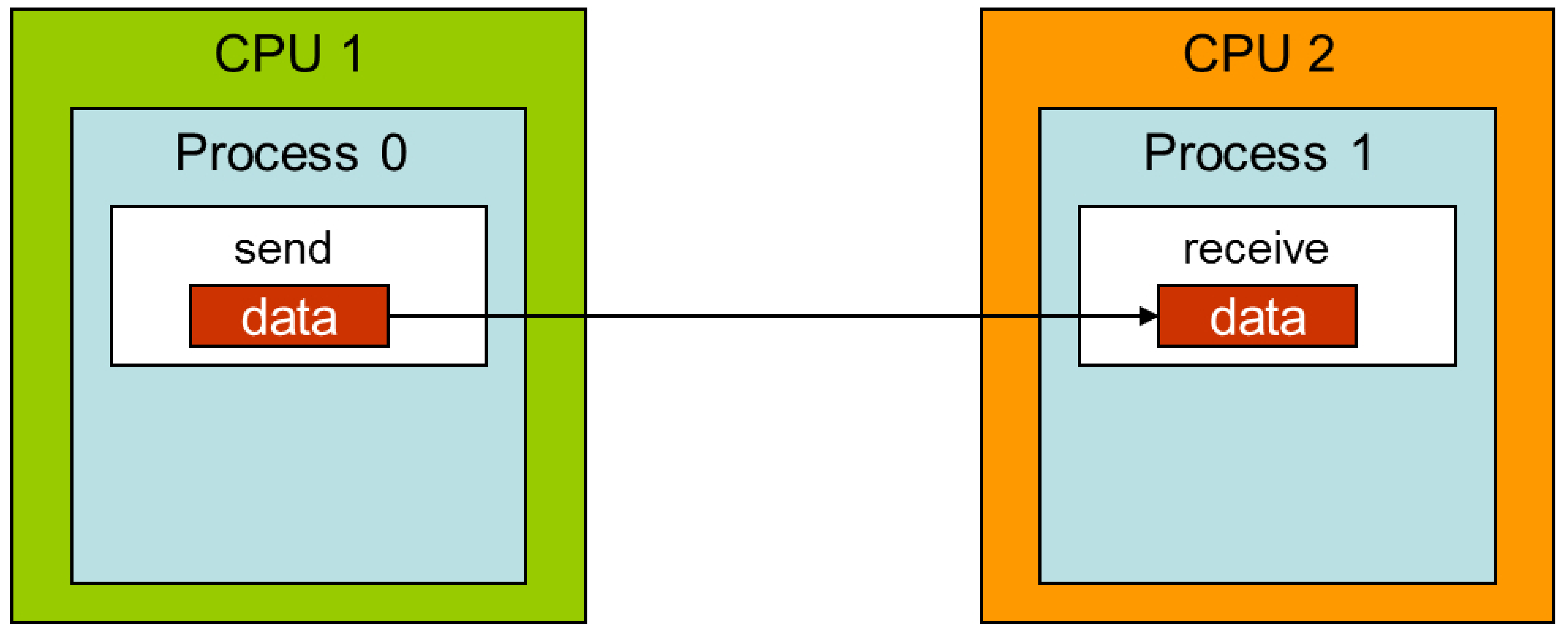
- Rather simple w.r.t. other communication middleware: no underlying framework, just library.

MPI basics

- MPI communication is based on communicators
- Each process within a communicator has a rank
- Ranks go from 0 to size-1

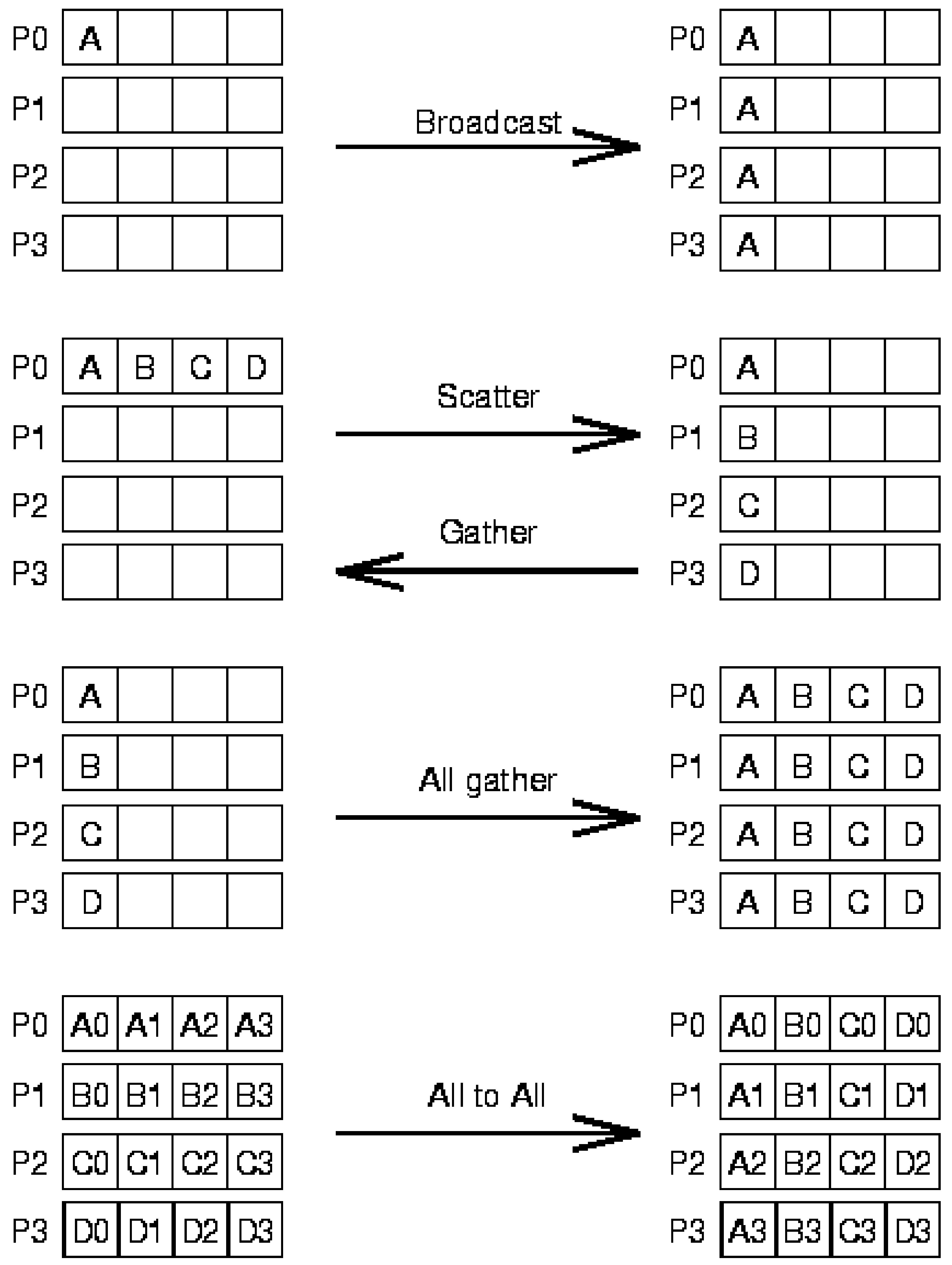


MPI communication



- Point to point

- Collective



The problem

- MPI provides communication with good performance but lacks features.
- Many features introduced during the years
- <u>Fault tolerance is still missing.</u>

- *Predefined error handlers: [default] causes the program to abort on all executing processes; [other] has no effect other than returning the error code to the user.*
- ***After an error is detected, the state of MPI is undefined****. [The standard] does not necessarily allow the user to continue to use MPI after an error is detected.*
  - <u>MPI 3.1 standard</u>

The problem causes

- The most important MPI programs are well tested.
- A single fault can stop the entire execution, unlikely to happen even if they do, computation can restart from the beginning.
- So why loose performance to bother with fault tolerance?

What is missing

- We can model a cluster as a collection of components connected in series, since a single failure can stop the entire job.
- Let’s suppose each processor core has a MTTF of a century (876000 h).
- Summit cluster has 2,397,824 cores, the MTTF of the cluster is

$$\frac{MTTF}{n\_cores} = \frac{876000\,h}{2397824} = 0.365\,h \ \approx 21\,min$$

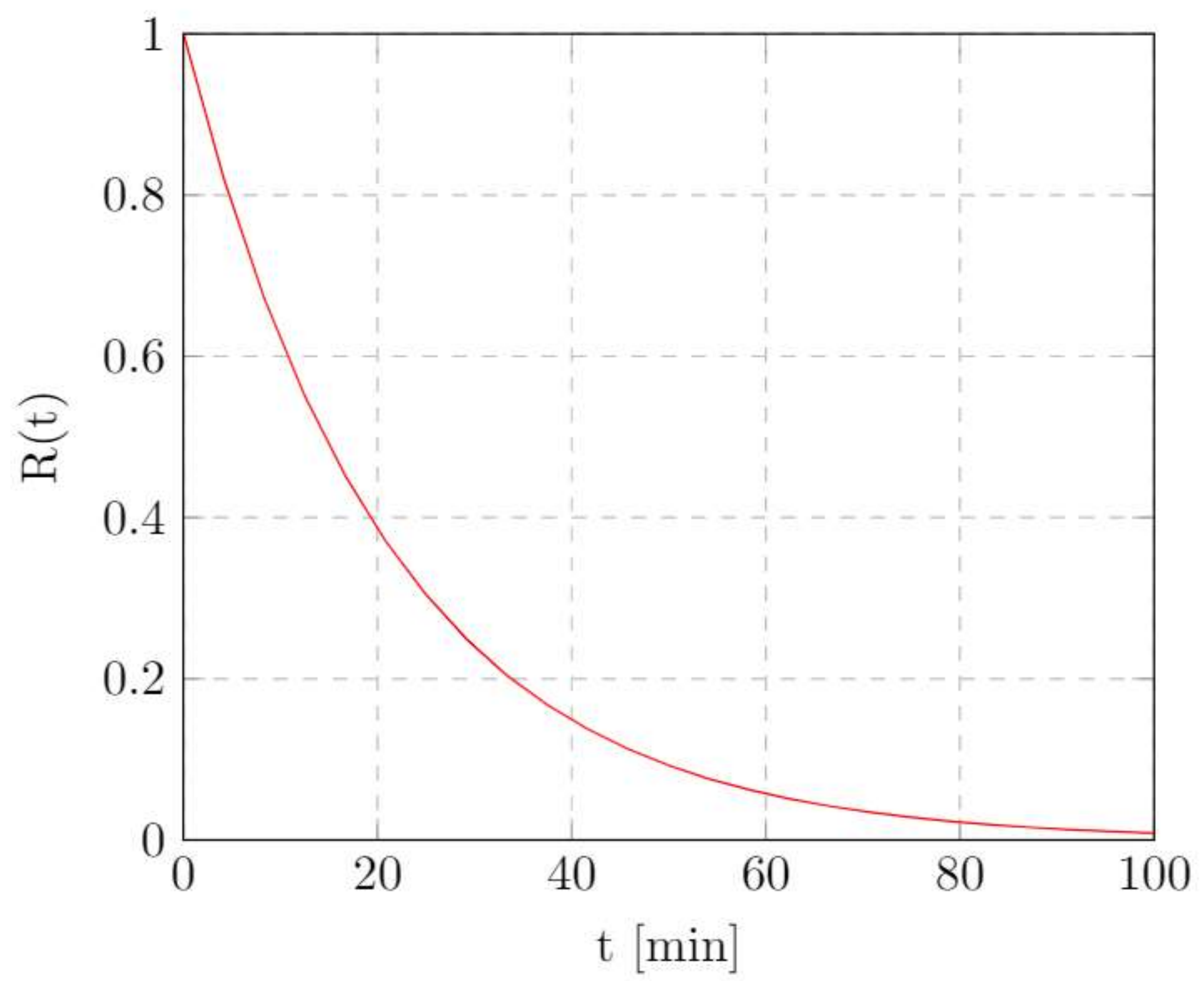
- From this value we can compute other metrics.

Reliability analysis

- R(t): probability that the system will operate correctly in a specified operating environment up until time t.

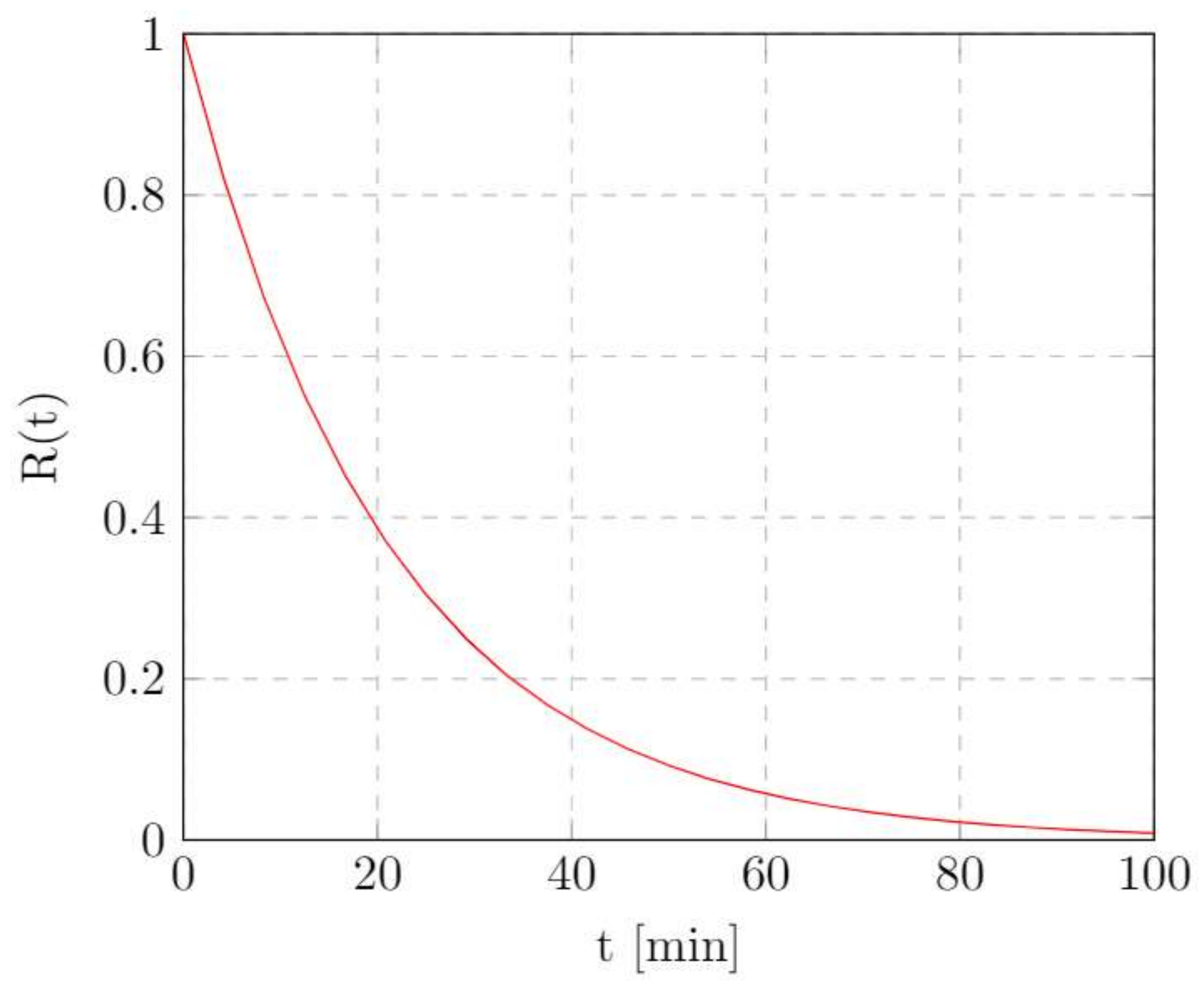
$$R(t)\colon P(no\ faults\ in\ [0,t])\ = e^{-\frac{1}{MTTF}t}$$

- Plotting the curve we get the graph on the right.
- The probability of terminating execution of an application requiring t minutes to complete without faults is equal to R(t).

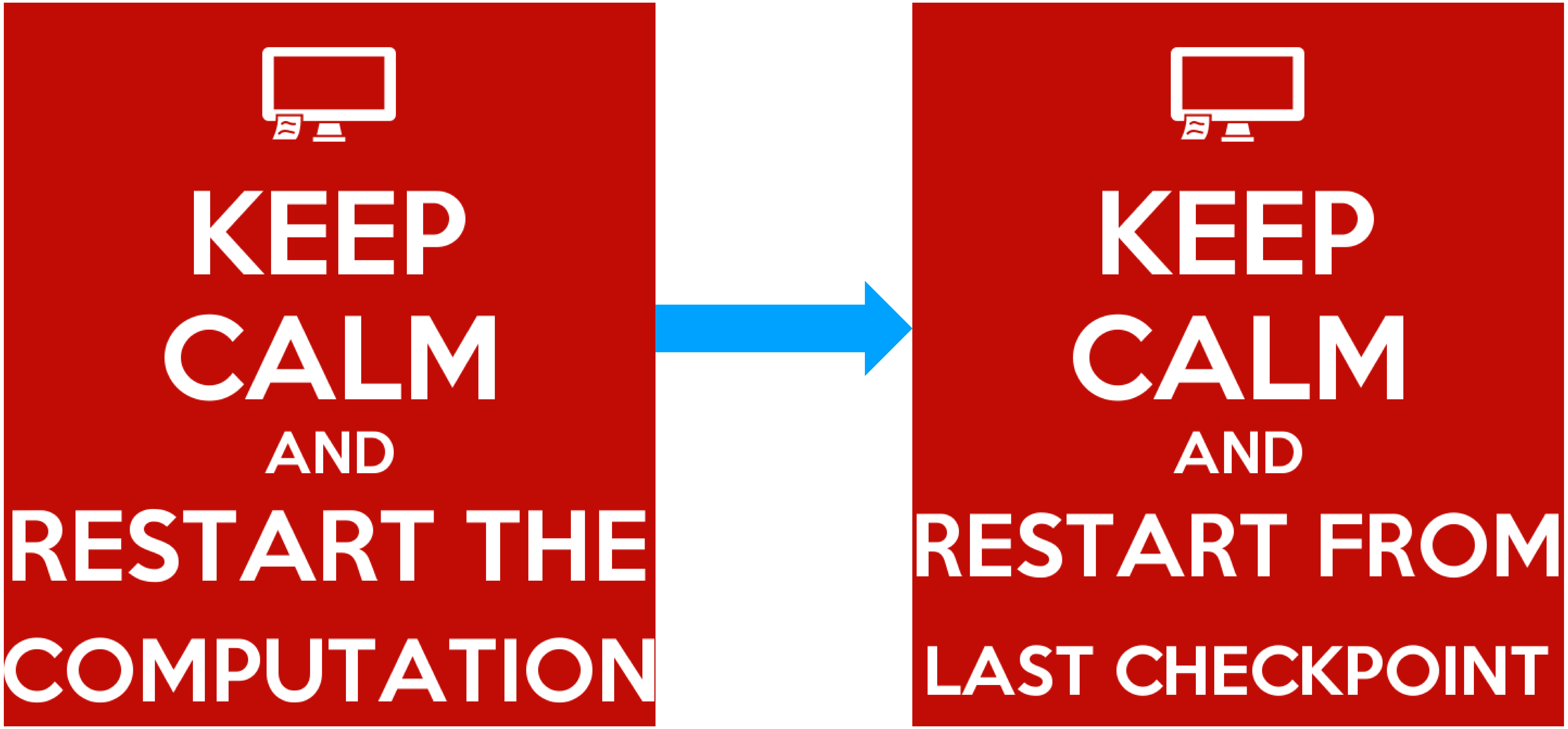


Reliability analysis

- If a program needs 20 minutes to complete, it will have to be run $^1/_{R(20)} \cong {}^1/_{0.4} = 2.5$ times on average;
- If a program needs an hour to complete, it will have to be run $^1/_{R(60)} \cong {}^1/_{0.057} \cong 17.54$ times on average;
- Exascalate4Cov would need about 46 hours to search in the entire database: about 3.5*10e58 times on average
- It’s easy to see that faults are the bottleneck of the system.



First solution: C/R


KEEP CALM AND RESTART THE COMPUTATION
KEEP CALM AND RESTART FROM LAST CHECKPOINT

Better solution

- ~~Let fault stop the program~~
- Let the program handle the fault and continue its execution past it.
- Many advantages, but harder to implement: needs communication and coordination between the nodes.
- Many efforts tried to solve the problem this way.

What is needed

- What can be useful to program for introducing fault tolerance?
  - Get which processes failed;
  - Propagate errors on the network;
  - Eliminate faulty processes from the network;
  - Agreement algorithm.
- User Level Failure Mitigation (ULFM) provides all of these.

ULFM

- Helps with fault tolerance, doesn’t implement it.
- It’s focused toward user level usage, not system level.
- Many efforts based on it to create automatic tools
  - Fenix
  - CPPC

Fenix

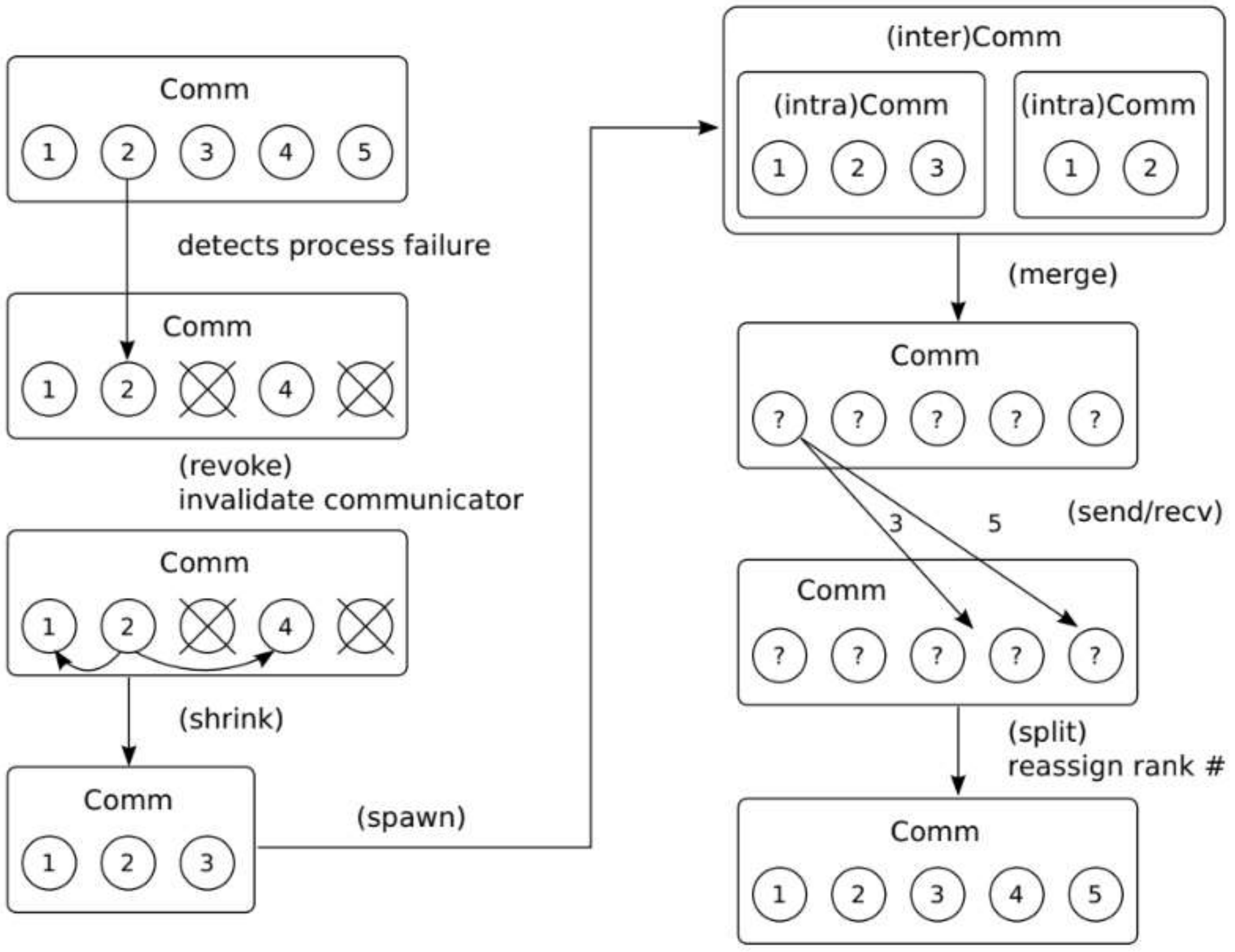


Fig. 1. Communicator recovery in Fenix by spawning new processes. The recovery process when using a process pool is similar.

CPPC

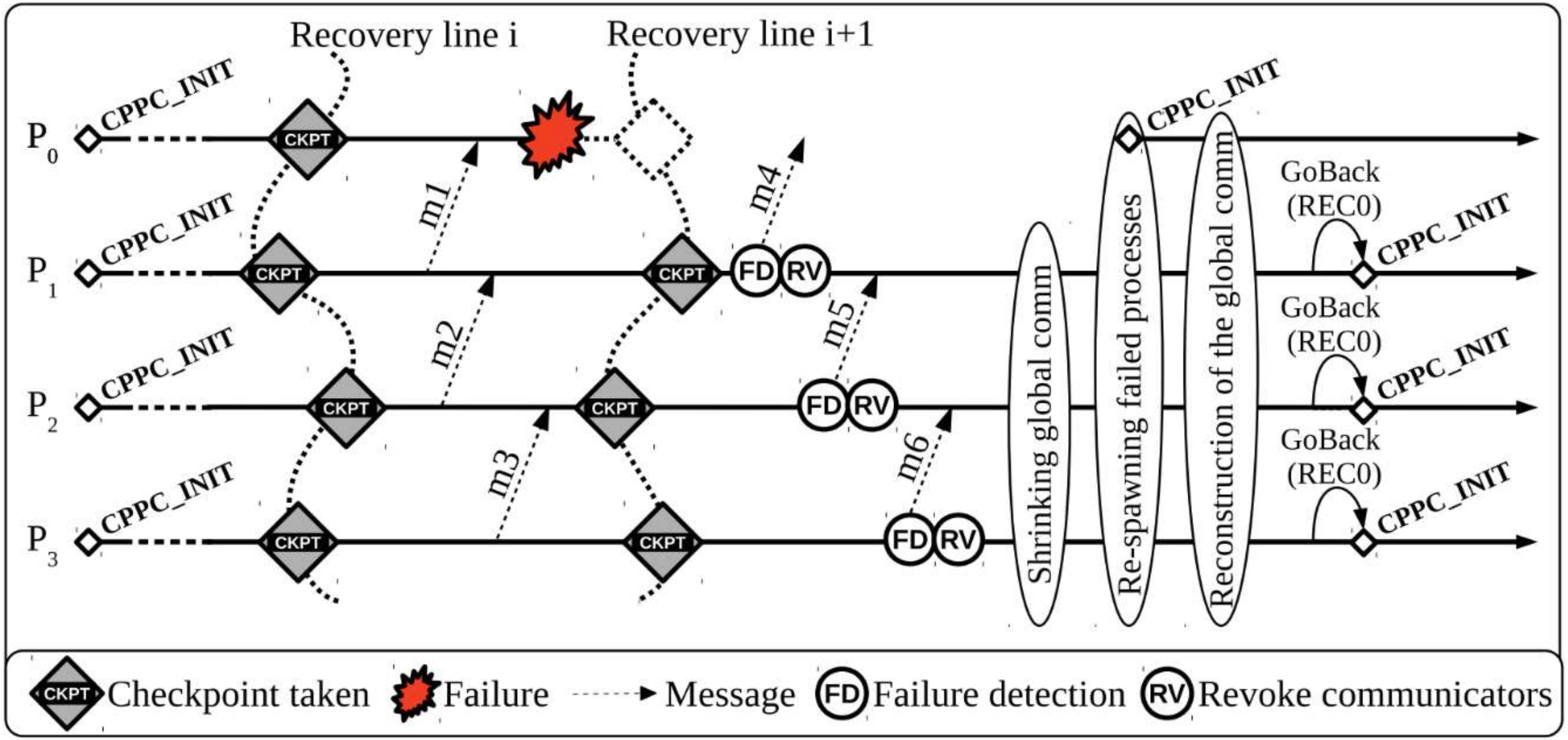


Figure 1. Global recovery strategy.

Weak points

1. All processes must collaborate in the recovery process, even non-faulty ones;

2. Both need few changes in the code in order to be functional: recovery is performed by loading a checkpoint, but the user (programmer) needs to choose when and what is saved;

3. All process restart from last checkpoint, even non-faulty ones (**global recovery**)

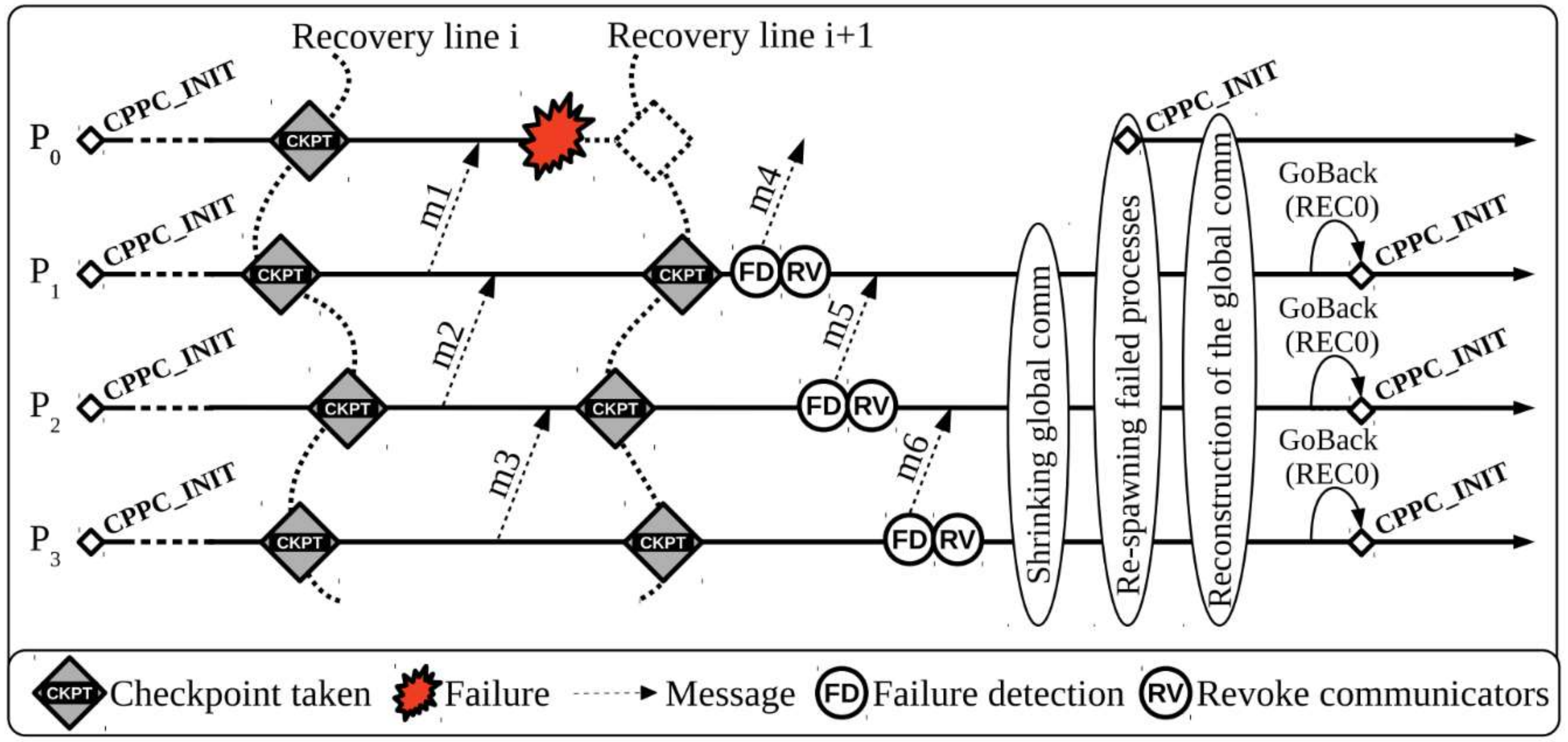


Figure 1. Global recovery strategy.

Weak point analysis

- - **First**: ULFM heritage
- - **Second**: <u>application-aware</u> approach, achieve fault tolerance without loosing too much performance.
- - **Third**: solved in a further effort
  (in Fenix ULFM wasn’t used because of the first weak point).

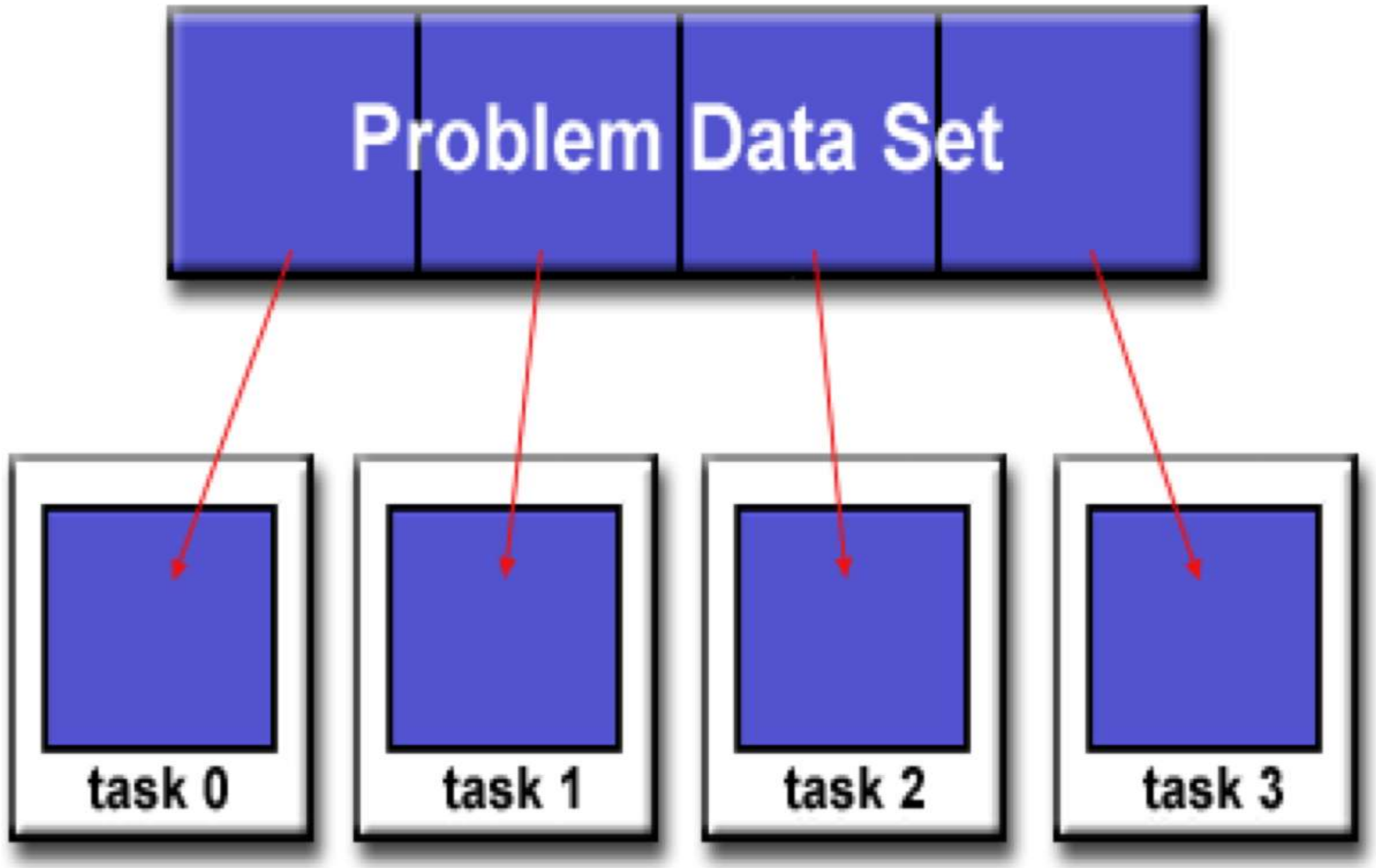
Inter-layer communication

- **Explicit communication**
- Requires changes of code (Fenix & CPPC way)
- Viable but sub-optimal
- Is there any alternative?

- **Implicit communication**
- Fault Tolerance layer operates assuming the application behaves in a certain way
- No code changes -> Transparency
- FT layer is application-aware
  not portable
- FT layer is characteristic-aware
  some degree of portability

Data parallelism

- Focus on the distribution of the data across different nodes, which will operate on it in parallel.
- Good scaling, exascale ready
- Almost no communication between the processes.
- The absence of communication is exploitable.
- Even further, a failure has local impact, making graceful degradation possible



Proposed approach

Application-aware	Characteristic-aware

Proposed approach

All processes must collaborate in the recovery process, even non-faulty ones;	Same since it’s ULFM based
Application-aware	Characteristic-aware

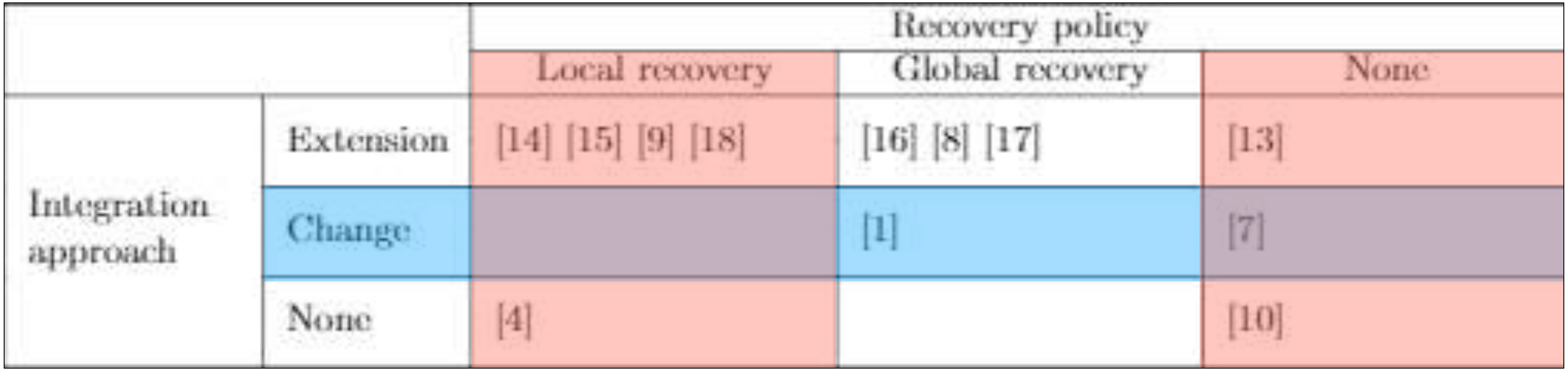
Proposed approach

All processes must collaborate in the recovery process, even non-faulty ones;	Same since it’s ULFM based
Need for few changes in the code in order to be functional: recovery is performed by loading a checkpoint, but the user (programmer) needs to choose when and what is saved;	Transparency: no changes in code, lower level structure
Application-aware	Characteristic-aware

Proposed approach

All processes must collaborate in the recovery process, even non-faulty ones;	Same since it's ULFM based
Need for few changes in the code in order to be functional: recovery is performed by loading a checkpoint, but the user (programmer) needs to choose when and what is saved;	Transparency: no changes in code, lower level structure
All process restart from last checkpoint, even non-faulty ones (global recovery)	Recovery is optional, if done it doesn't impact non-faulty processes (local recovery)
Application-aware	Characteristic-aware

Scenario view



		Recovery policy		
		Local recovery	Global recovery	None
Integration approach	Extension	[14] [15] [9] [18]	[16] [8] [17]	[13]
	Change		[1]	[7]
	None	[4]		[10]

Analysis of 3 possible approaches:

Direct mitigation	Hierarchical mitigation	C/R fault tolerance
[DM]	[HM]	[C/R]

Evaluation of the research

- To be compliant with HPC standards, these approaches shall achieve fault tolerance with low overheads in terms of performance. This is the most important metric.
- The artefacts produced must be scalable, since they target scalable applications.
- Configurability of the produced artefacts is also important to adapt their behaviour to the needs of the user.