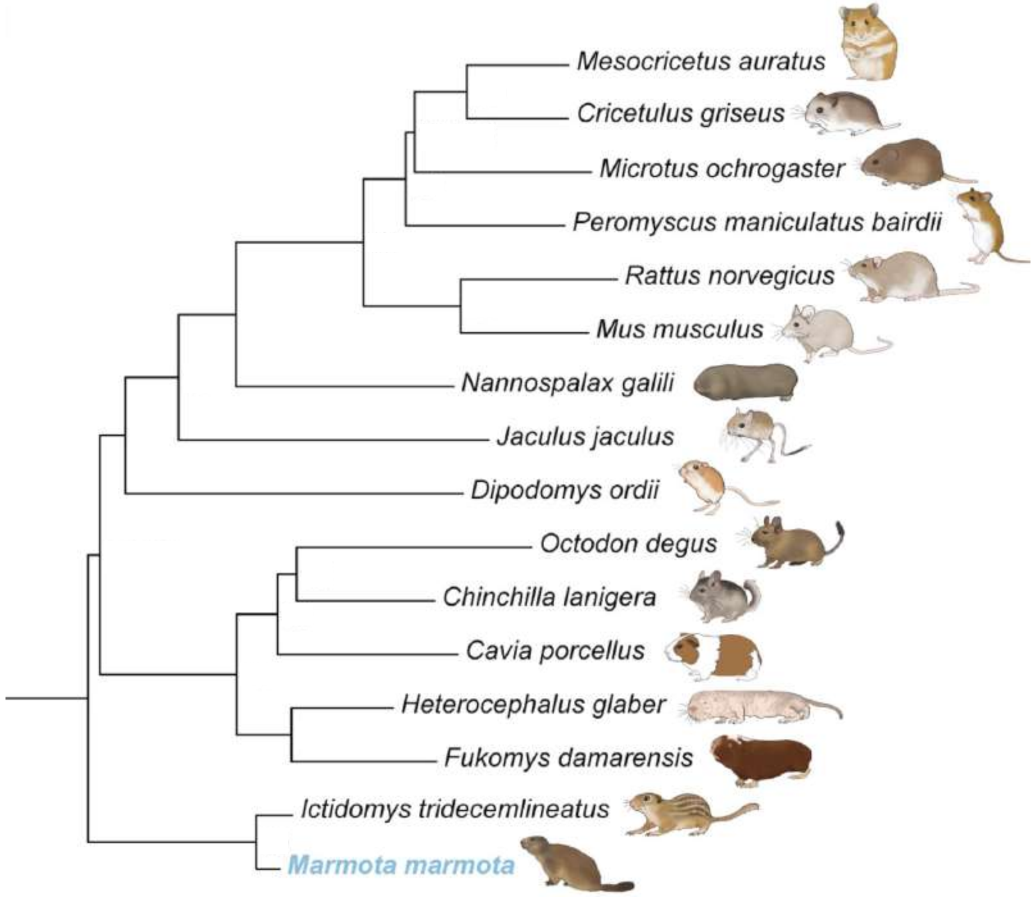
Future-Proofing Bioinformatic Applications

Handling CPU-failures, abstracting MPI & reproducible experiments · 2024-10-30

Lukas Hübner

Phylogenetics

> Nothing in biology makes sense except in the light of evolution.
>
> *Theodosius Dobzhansky*



Phylogenetics

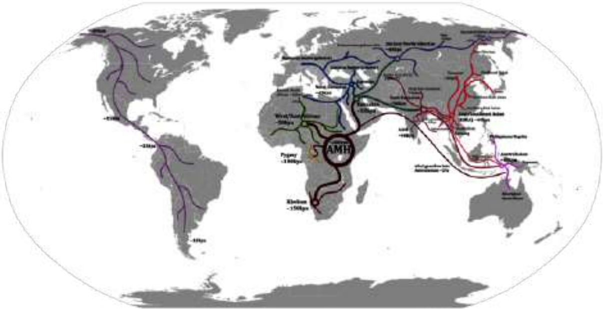
Describe evolutionary history among species using trees

Phylogenetics: Applications

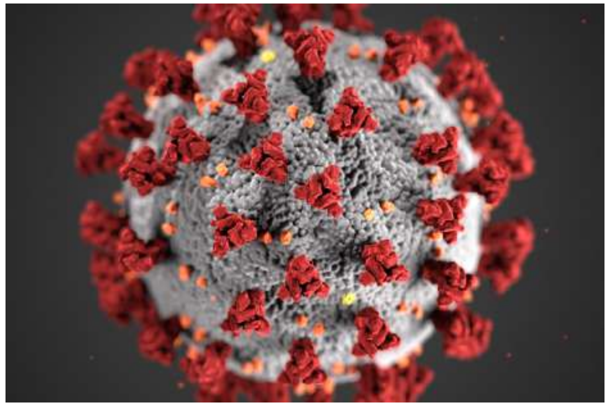
understanding evolution

host-parasite interaction

wildlife conservation



human migration patters

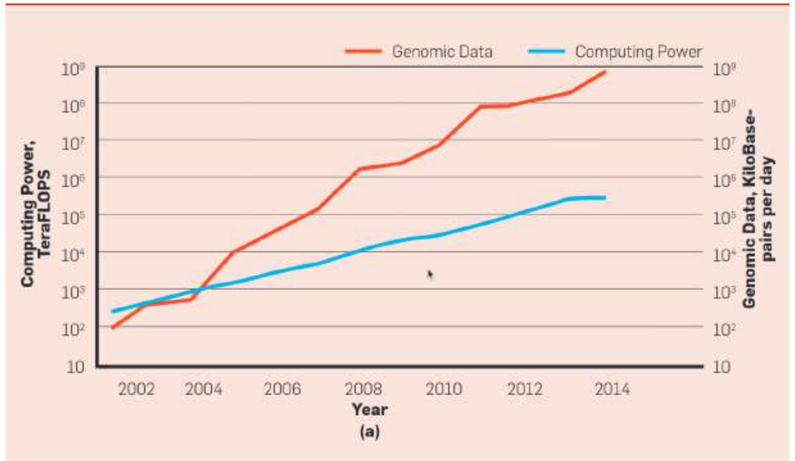


pathogen spread

forensics

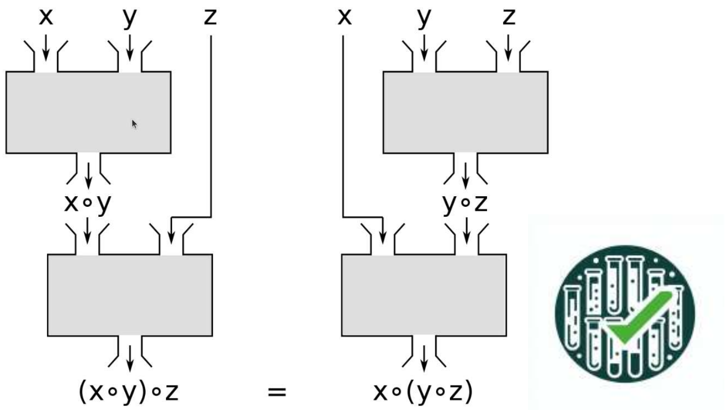
Scalability Challenges

amount of genomic data grows faster than Moore's Law



using more CPUs increases frequency of hardware failures

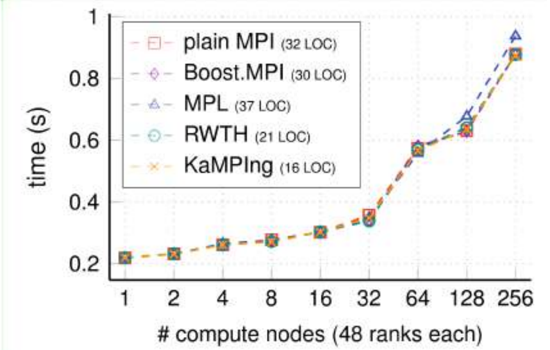
distributed software must yield reproducible results



abstractions needed for distributed-memory development

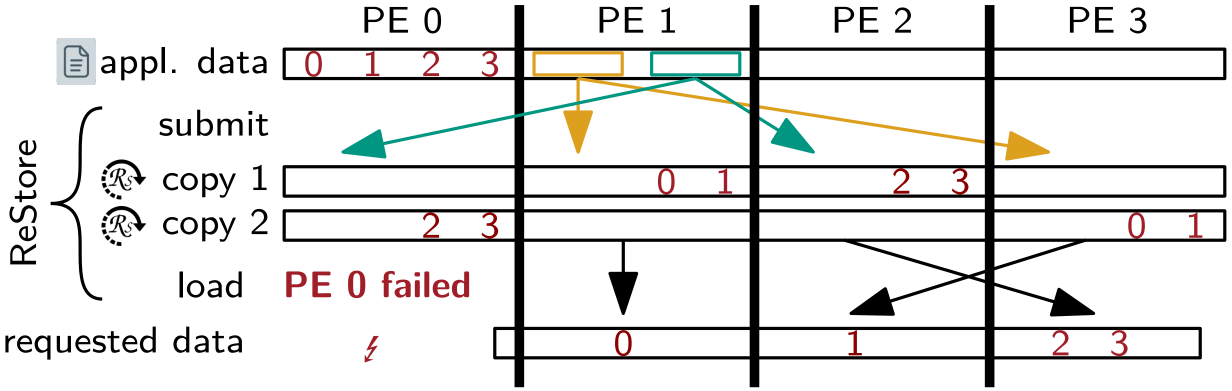
Overview

zero-overhead C++ MPI wrapper and distributed toolbox [SC24]

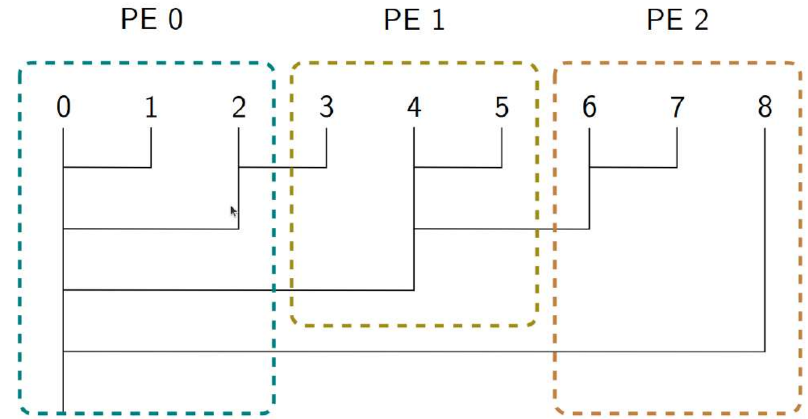


```
recv_buf = comm.allgatherv(send_buf(v_local));
```

replicated storage for rapid recovery after CPU failure [FTXS22]

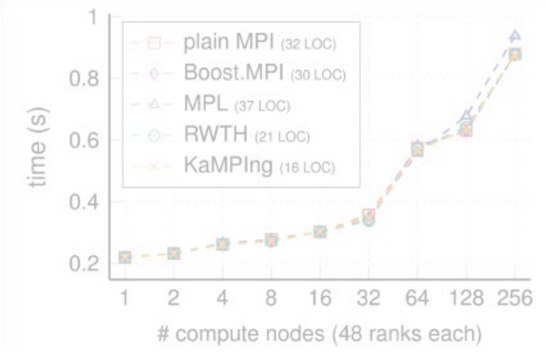


reproducible distributed memory reduction



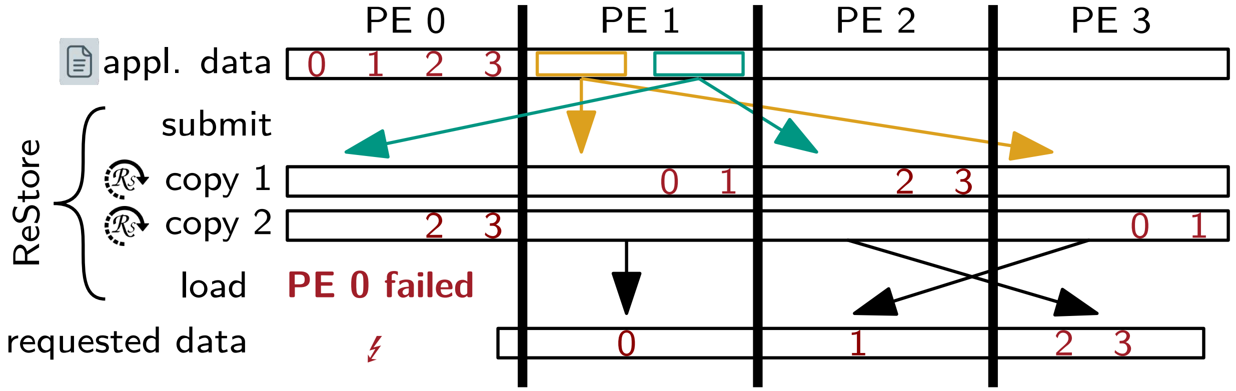
Overview

zero-overhead C++ MPI wrapper and distributed toolbox [SC24]

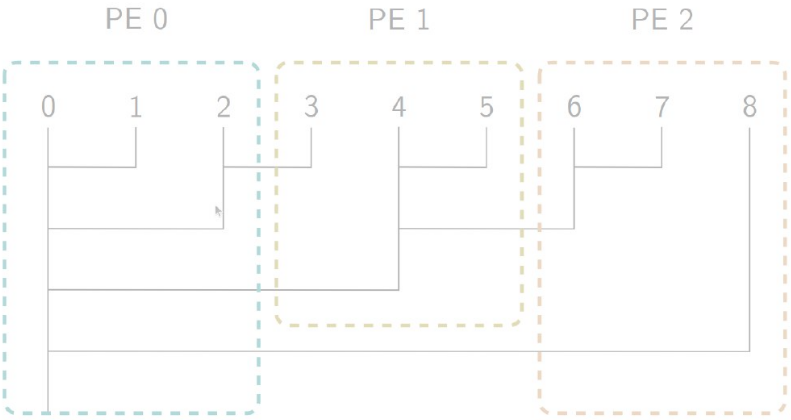


```
recv_buf = comm.allgatherv(send_buf(v_local));
```

replicated storage for rapid recovery after CPU failure [FTXS22]

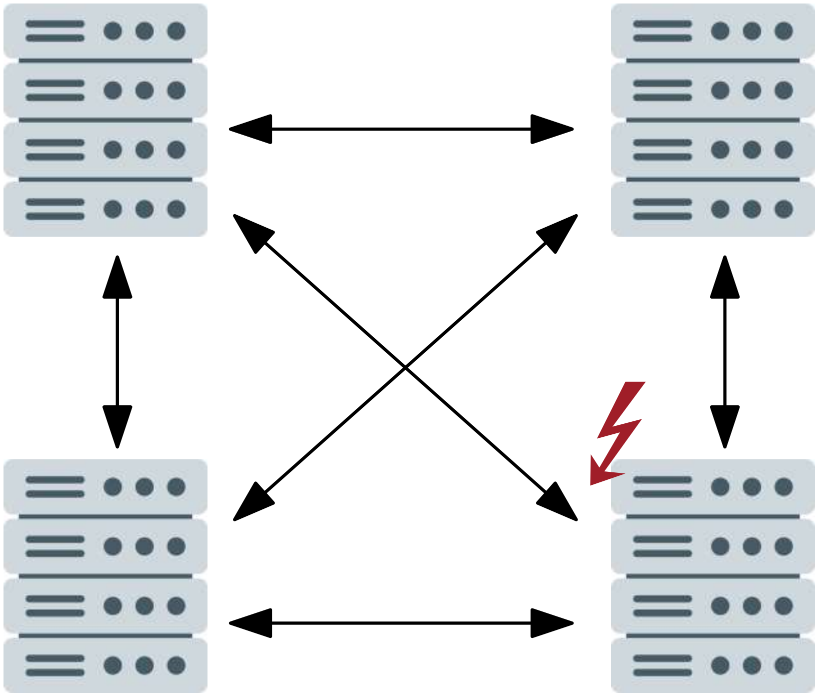


reproducible distributed memory reduction

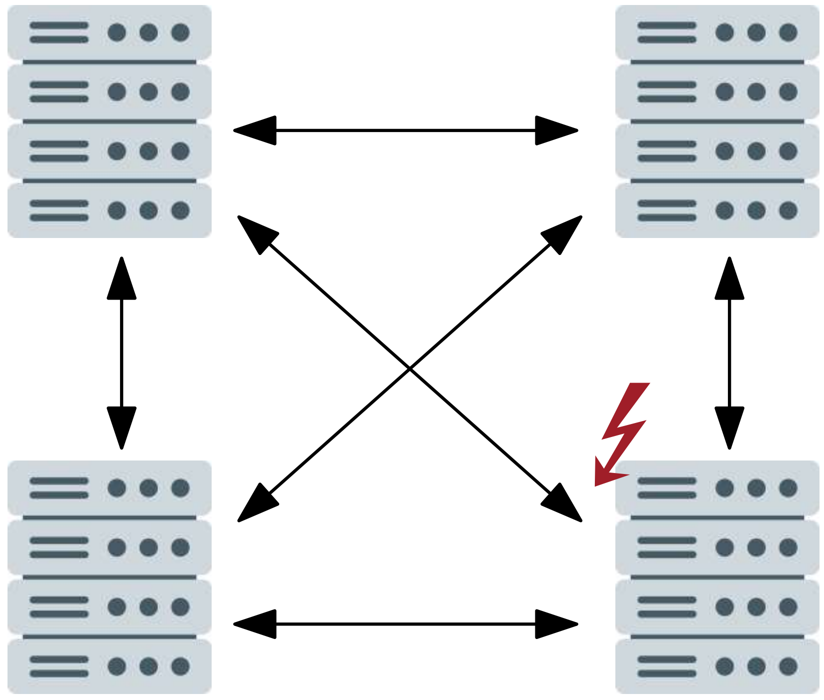


Fault-Tolerance

- Using more CPUs → **more frequent failures** → more recoveries
- Reports of **2 hardware failures per day**
- The parallel filesytem is a **bottleneck**



Fault-Tolerance

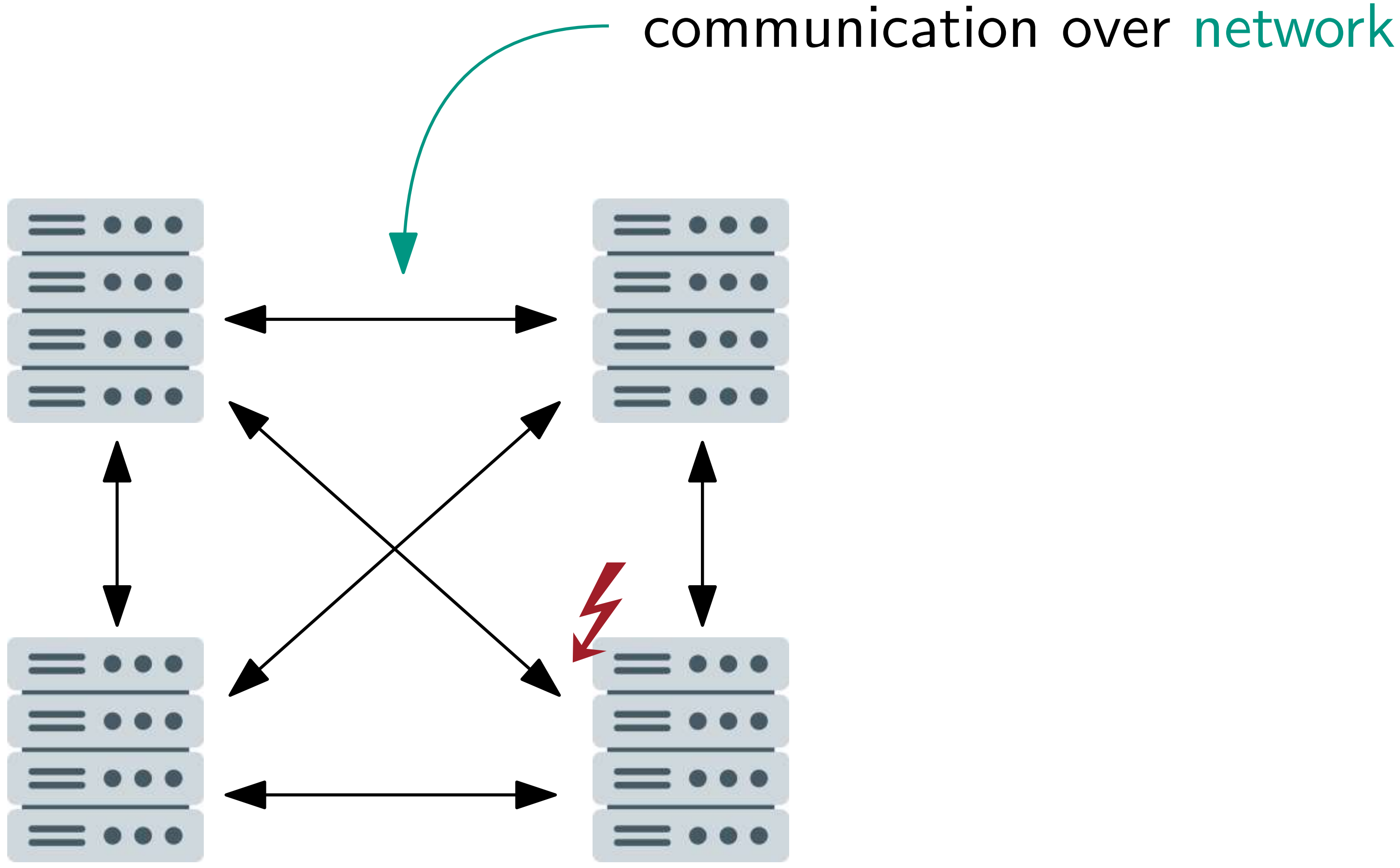


When I say "**CPU**", I mean:

- "compute node"
- "Processing Element"

simplyfing over multi-core and multi-socket architecture

Fault-Tolerance

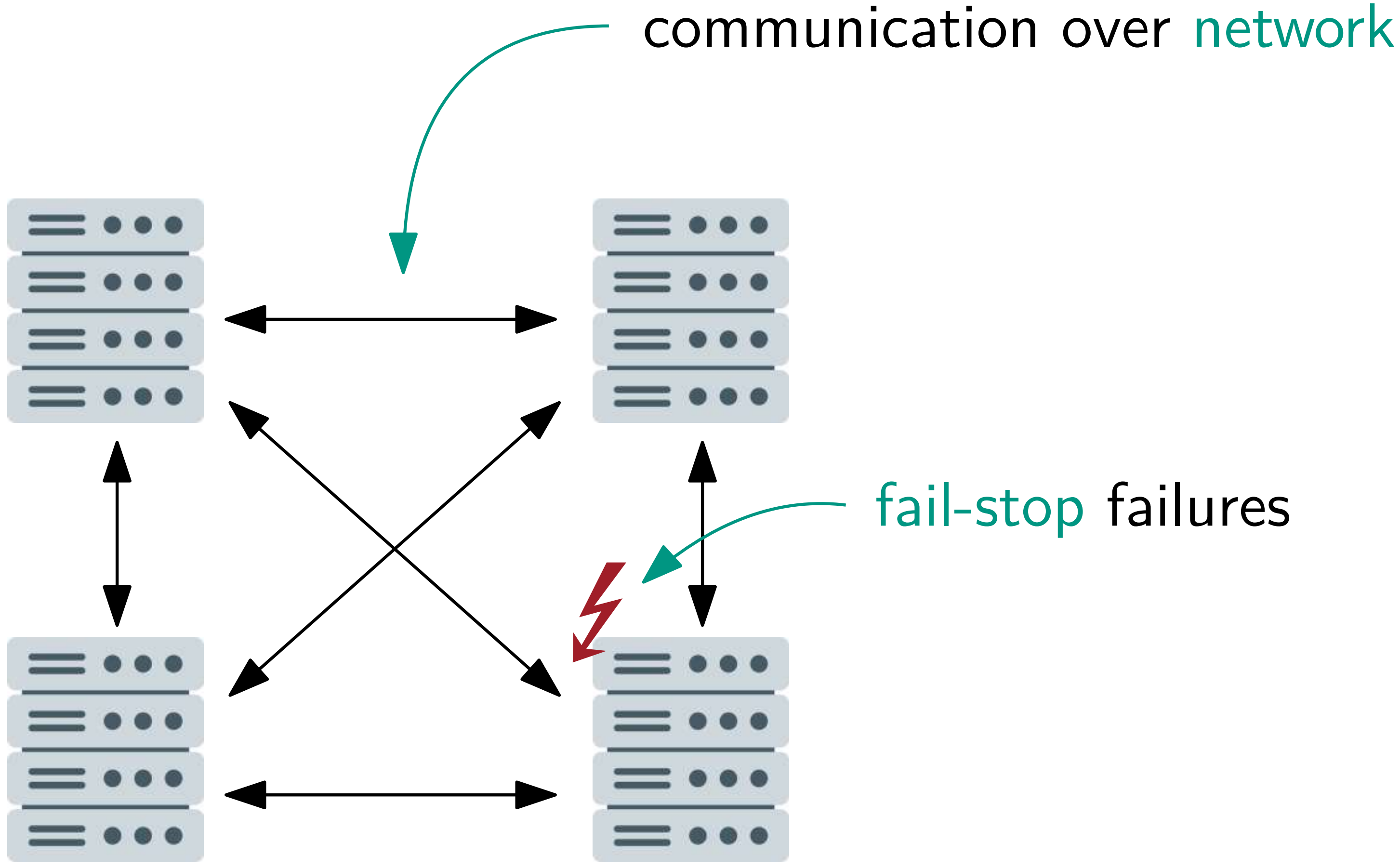


When I say “**CPU**”, I mean:

- “compute node”
- “Processing Element”

simplyfing over multi-core and multi-socket architecture

Fault-Tolerance



communication over network

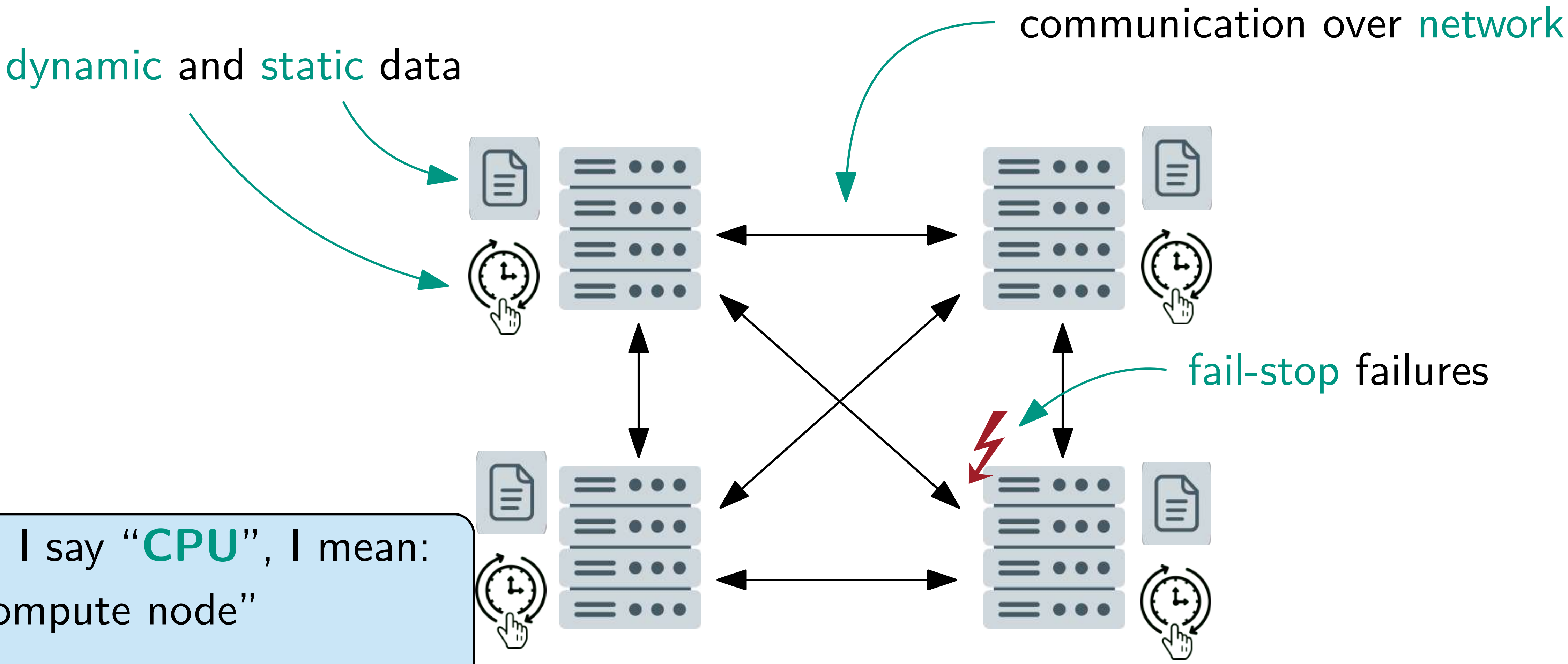
fail-stop failures

When I say "**CPU**", I mean:

- "compute node"
- "Processing Element"

simplyfing over multi-core and multi-socket architecture

Fault-Tolerance



dynamic and static data

communication over network

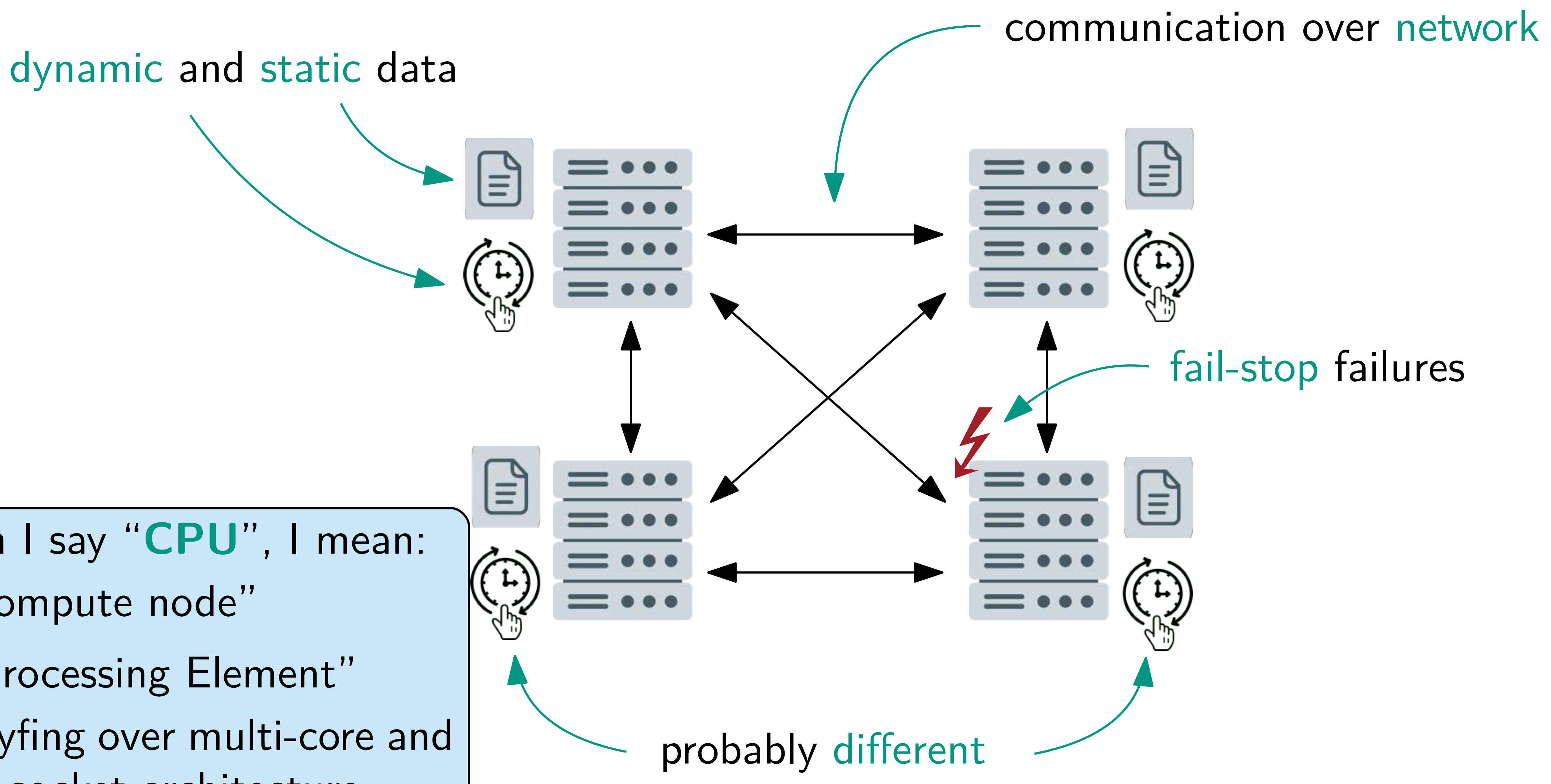
fail-stop failures

When I say “**CPU**”, I mean:

- “compute node”
- “Processing Element”

simplyfing over multi-core and multi-socket architecture

Fault-Tolerance

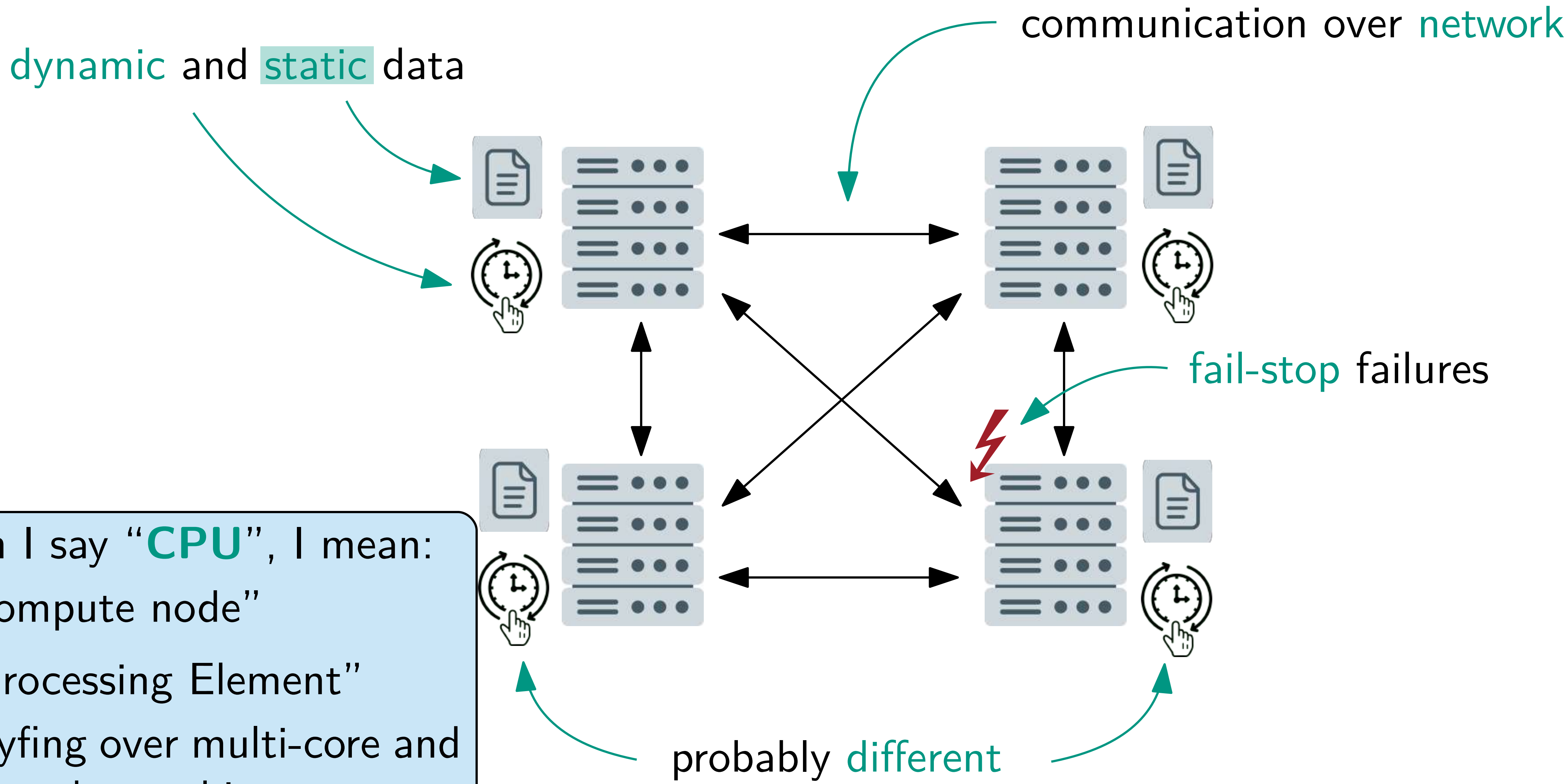


When I say “**CPU**”, I mean:

- “compute node”
- “Processing Element”

simplyfing over multi-core and multi-socket architecture

Fault-Tolerance

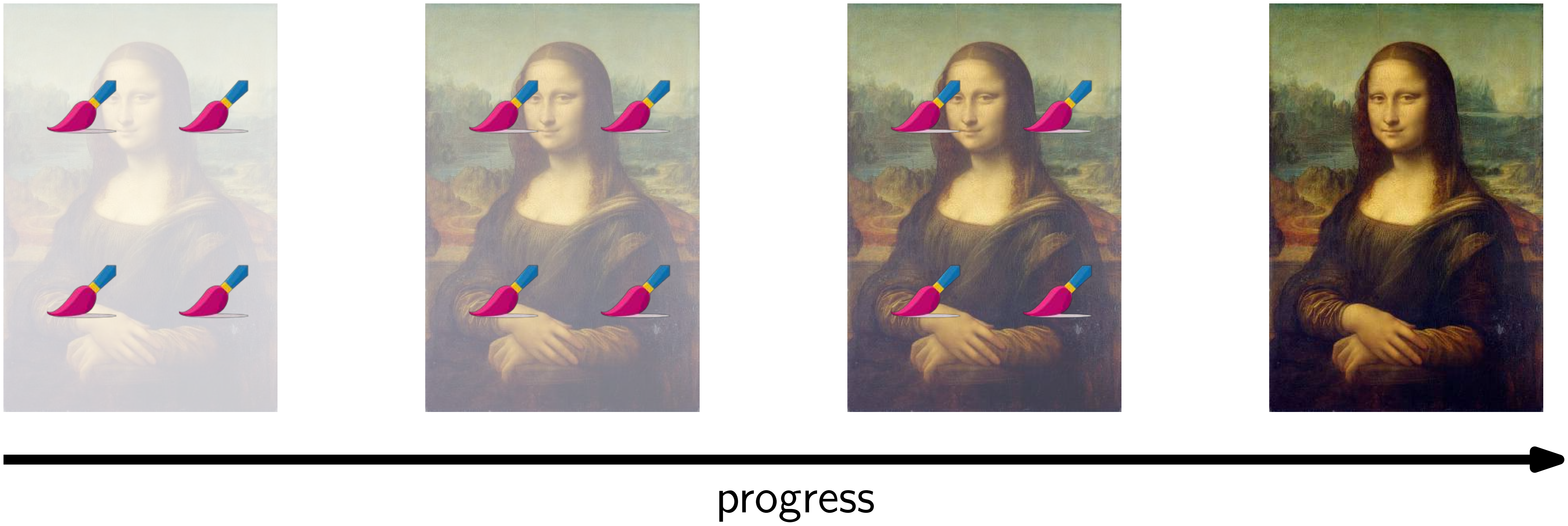


When I say "**CPU**", I mean:

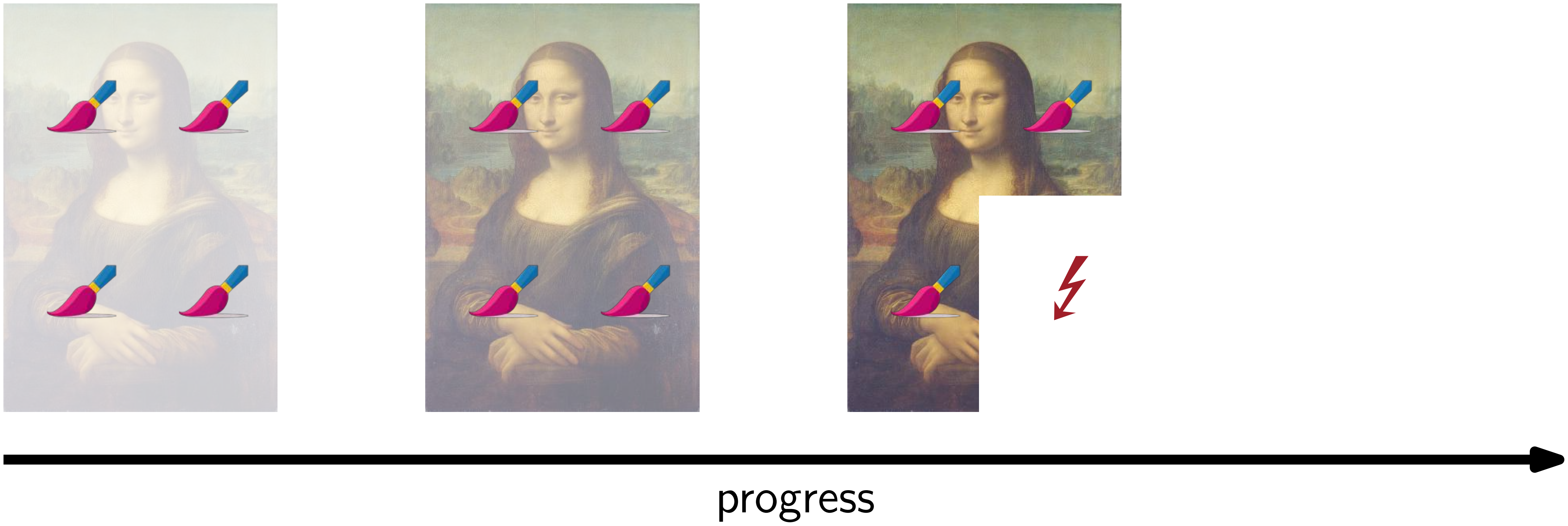
- "compute node"
- "Processing Element"

simplyfing over multi-core and multi-socket architecture

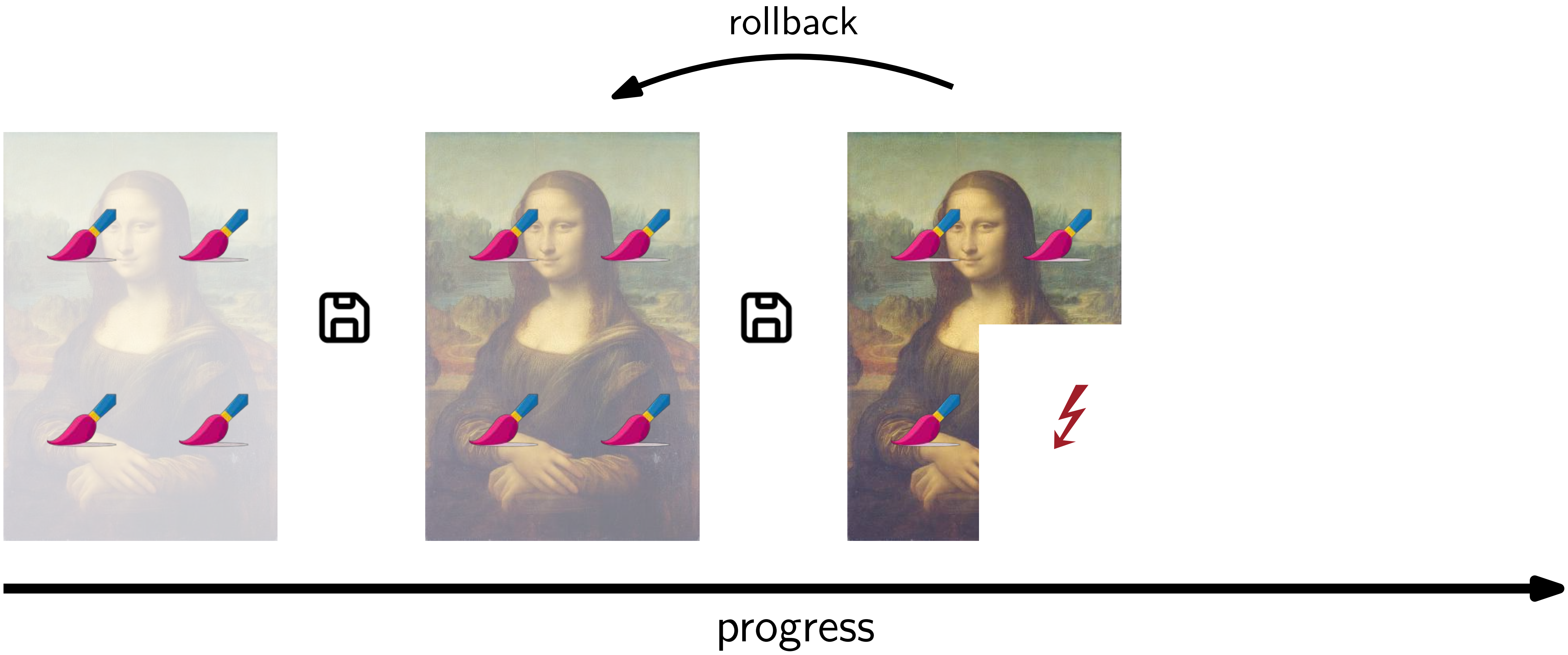
Checkpoints



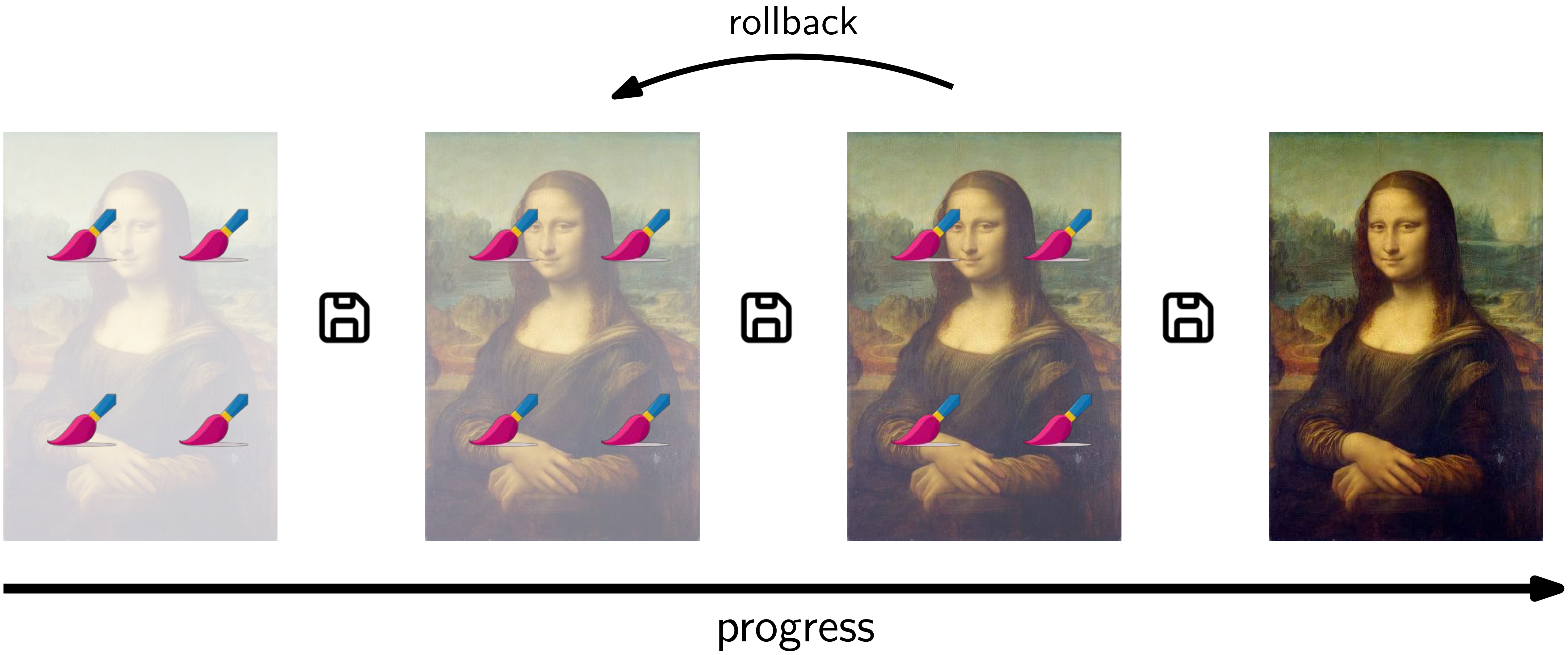
Checkpoints



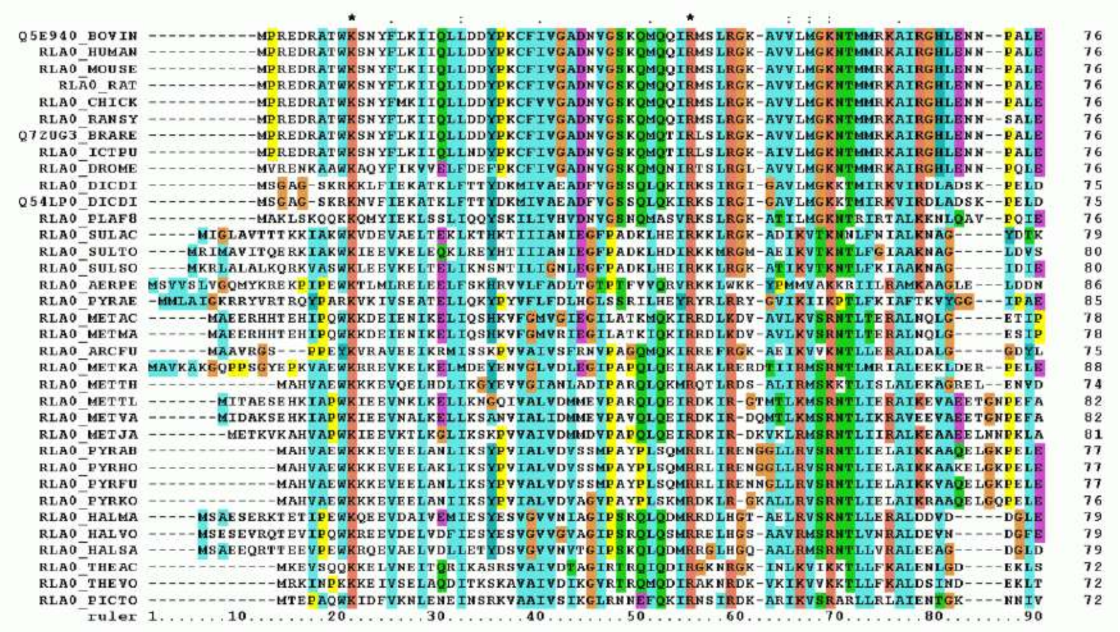
Checkpoints



Checkpoints



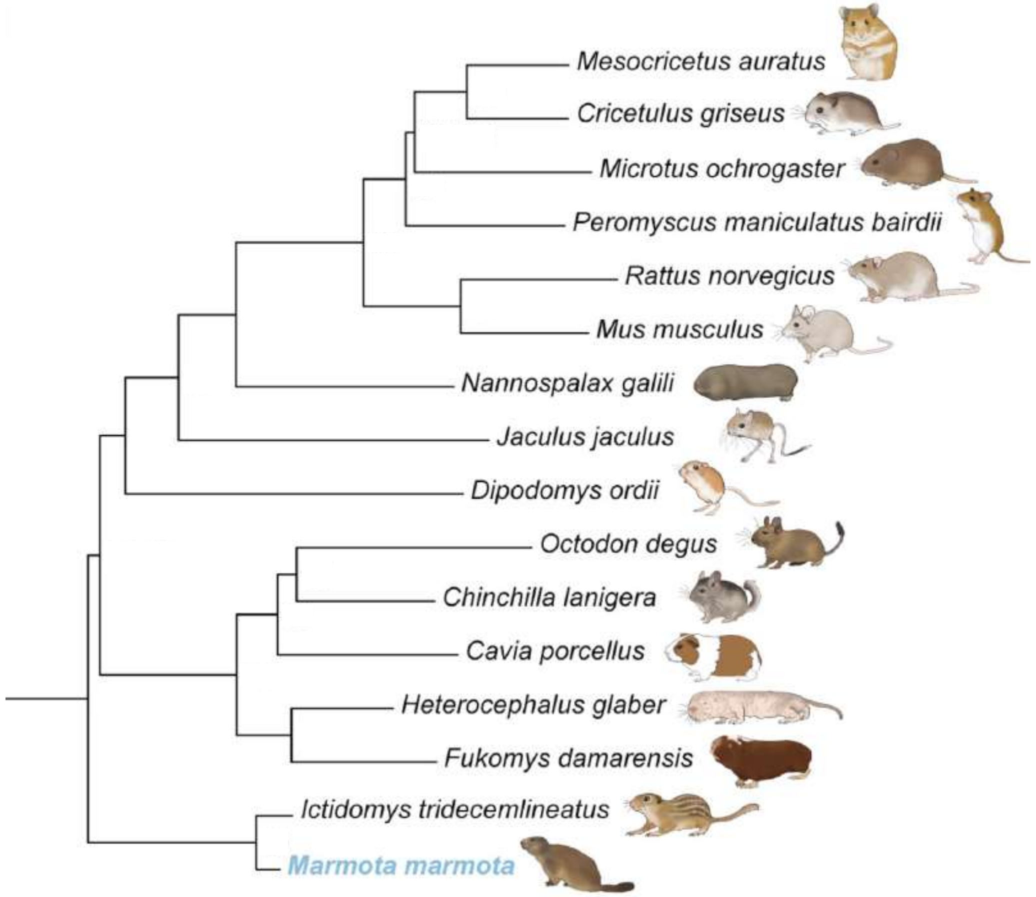
Phylogenetic Tree Search with RAxML-NG



——— different genomes

| | | | sites of the genome

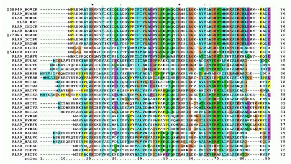
RAxML-NG →

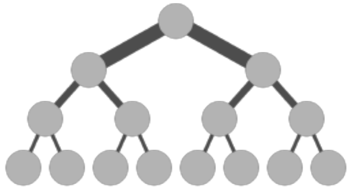
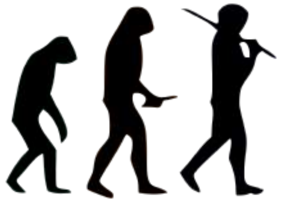


Multiple Sequence Alignment

Phylogenetic Tree

Phylogenetic Tree Search with RAxML-NG

Input:  genomic data of different species

Output: "best"  + 

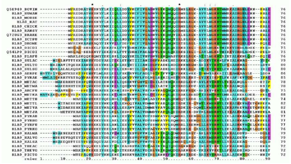
tree topology & branch lengths

evolutionary model

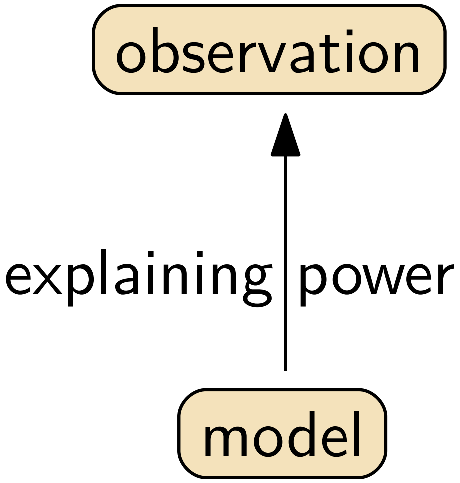
Algorithm:

Phylogenetic Tree Search with RAxML-NG

Input:



genomic data of different species

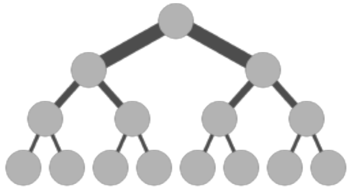


observation

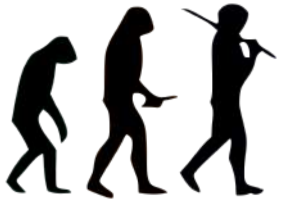
explaining power

model

Output: “best”



+

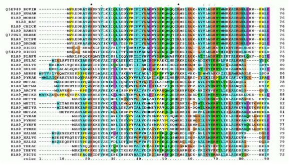


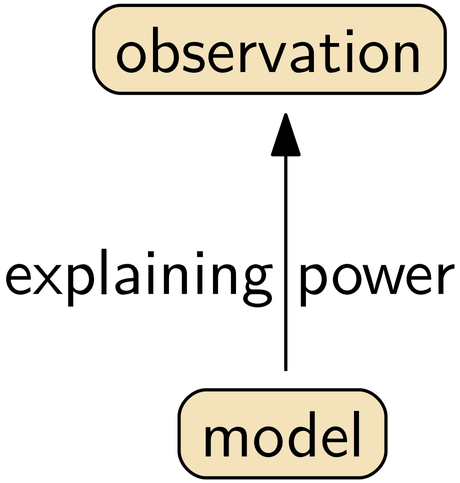
tree topology & branch lengths

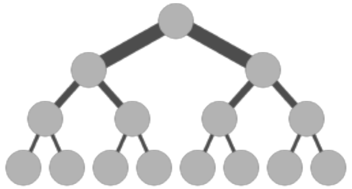
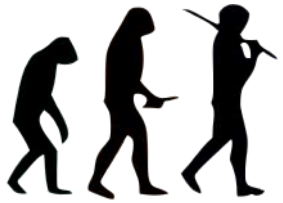
evolutionary model

Algorithm:

Phylogenetic Tree Search with RAxML-NG

Input:  genomic data of different species



Output: "best"  + 

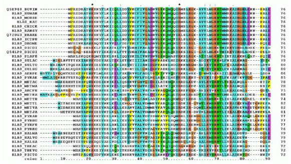
tree topology & branch lengths

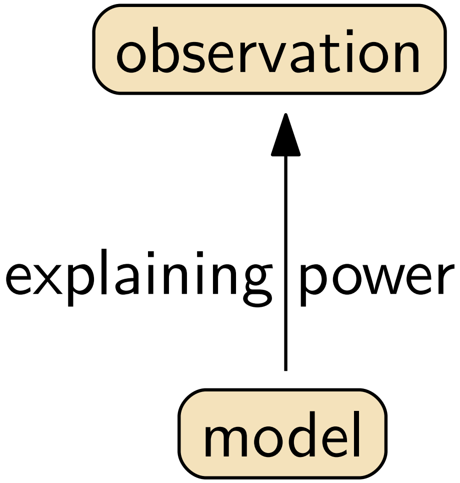
evolutionary model

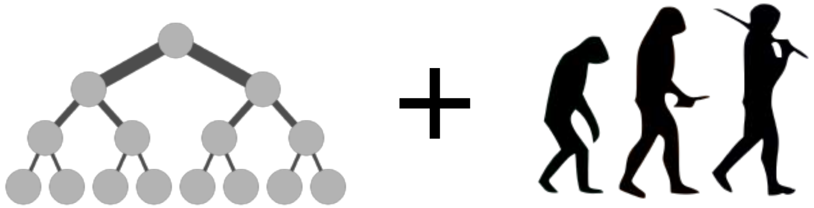
most likely *model* fitting the *observation*

Algorithm:

Phylogenetic Tree Search with RAxML-NG

Input:  genomic data of different species

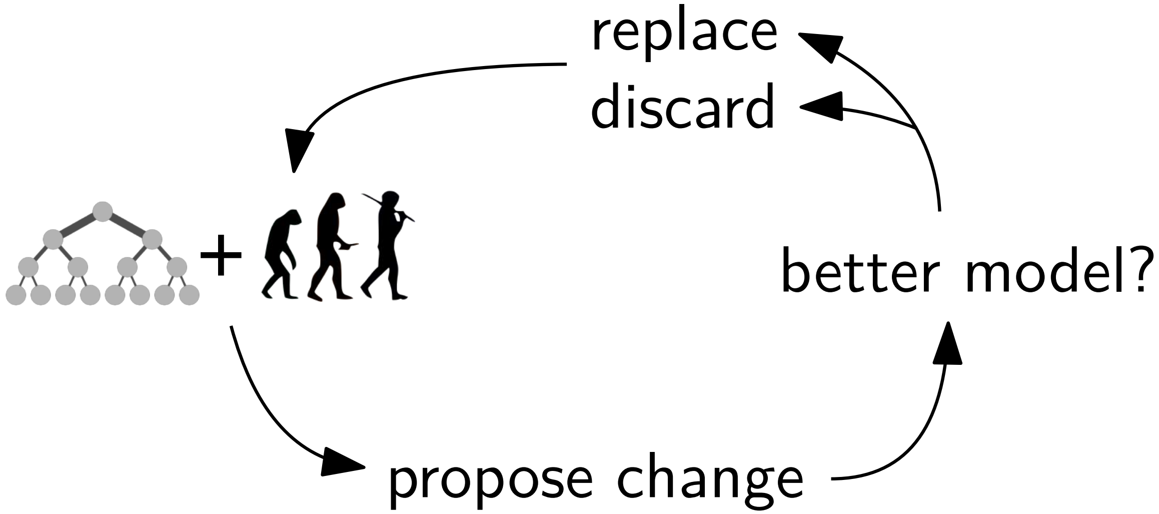


Output: “best” 

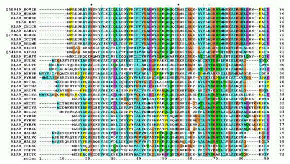
tree topology & branch lengths + evolutionary model

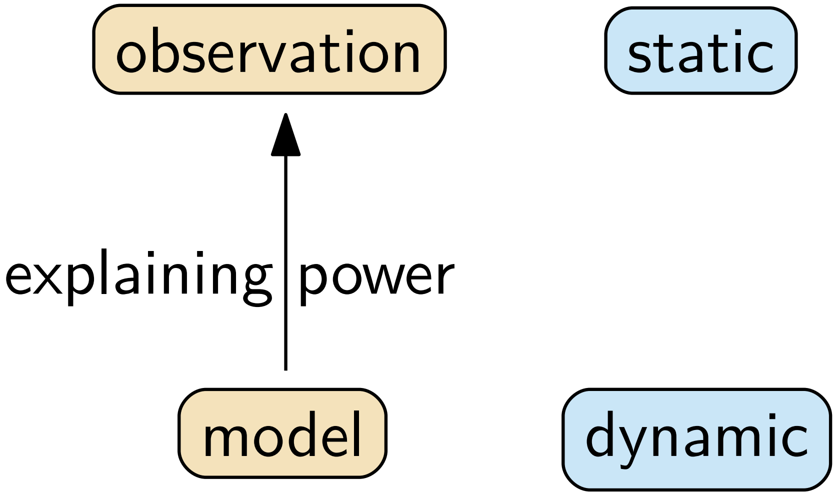
most likely *model* fitting the *observation*

Algorithm:



Phylogenetic Tree Search with RAxML-NG

Input:  genomic data of different species



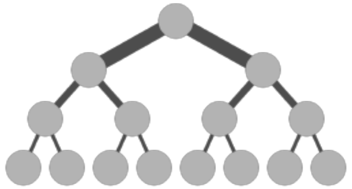
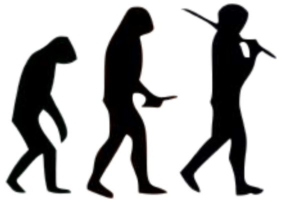
observation

static

explaining power

model

dynamic

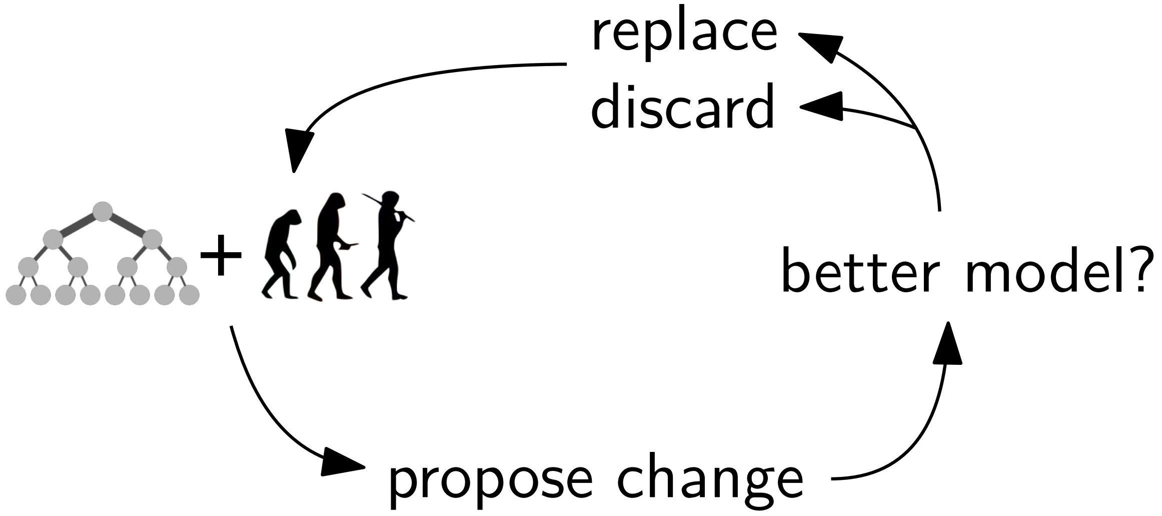
Output: “best”  + 

most likely *model* fitting the *observation*

tree topology & branch lengths

evolutionary model

Algorithm:



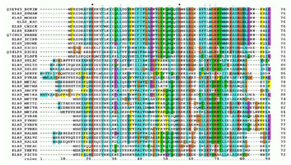
replace

discard

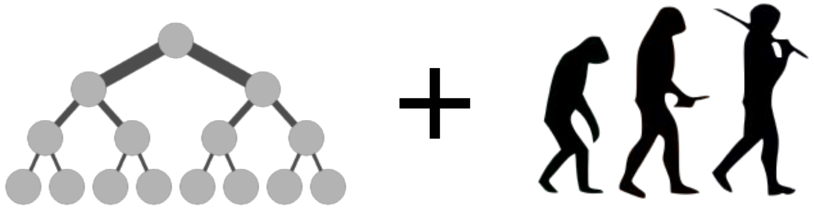
better model?

propose change

Phylogenetic Tree Search with RAxML-NG

Input:  genomic data of different species

Output: "best"



most likely *model* fitting the *observation*

tree topology & branch lengths + evolutionary model

observation

explaining power

model

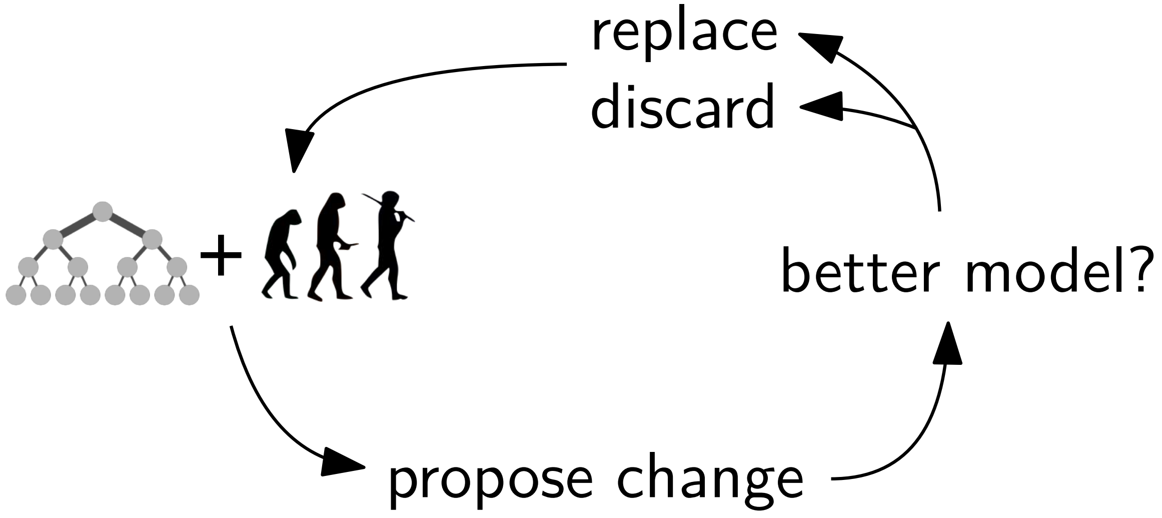
static

redistribute

dynamic

checkpoint & restore

Algorithm:



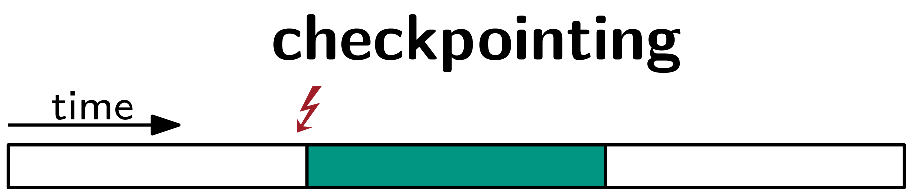
replace

discard

better model?

propose change

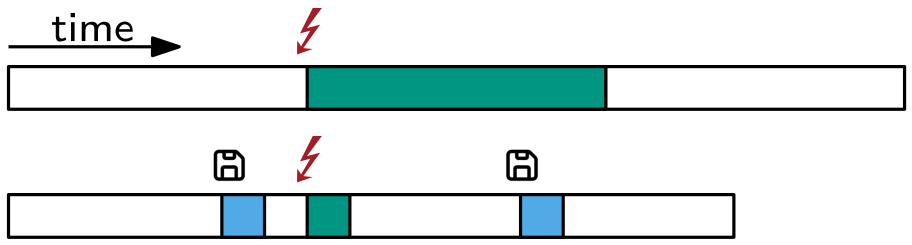
Checkpointing and Recovery Frequency



□	working	🟩	redundant work
🟦	checkpointing	🟪	recovering

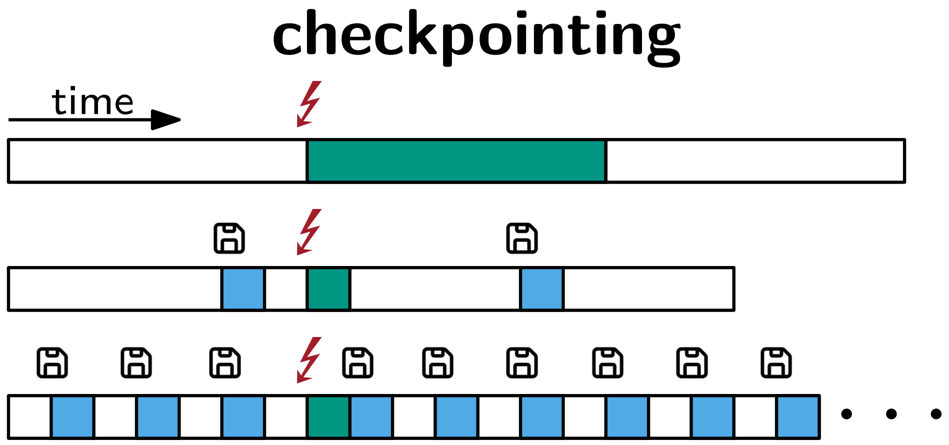
Checkpointing and Recovery Frequency

checkpointing



□	working	🟩	redundant work
🟦	checkpointing	🟪	recovering

Checkpointing and Recovery Frequency

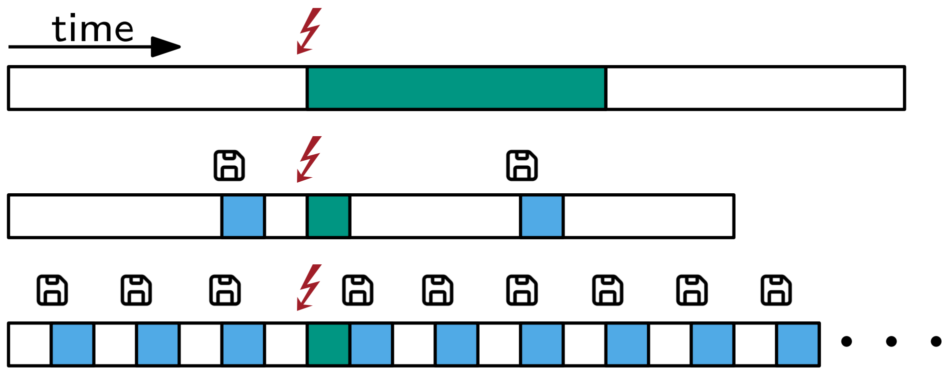


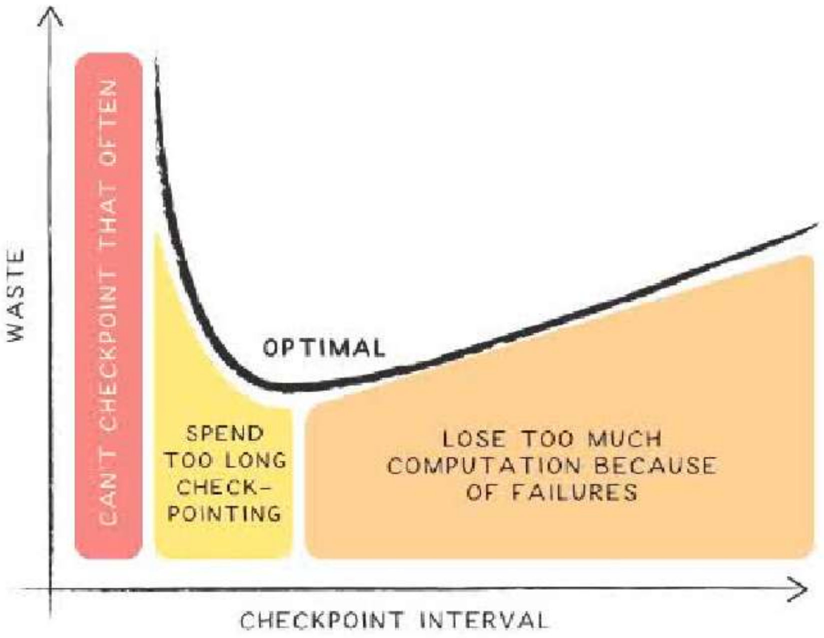
□ working 🟩 redundant work

🟦 checkpointing 🟪 recovering

Checkpointing and Recovery Frequency

checkpointing





Source: Benoit et al. *Checkpointing à la Young/Daly: An Overview*

□ working

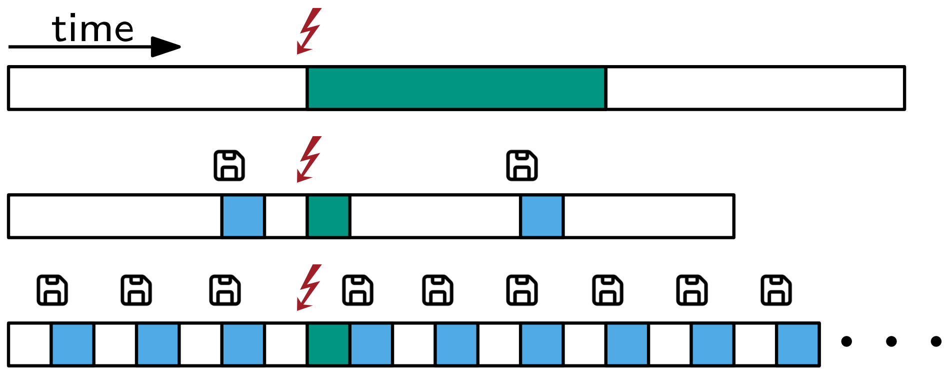
🟩 redundant work

🟦 checkpointing

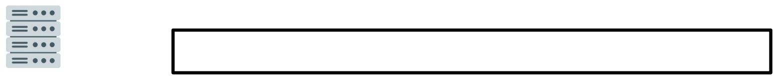
🟪 recovering

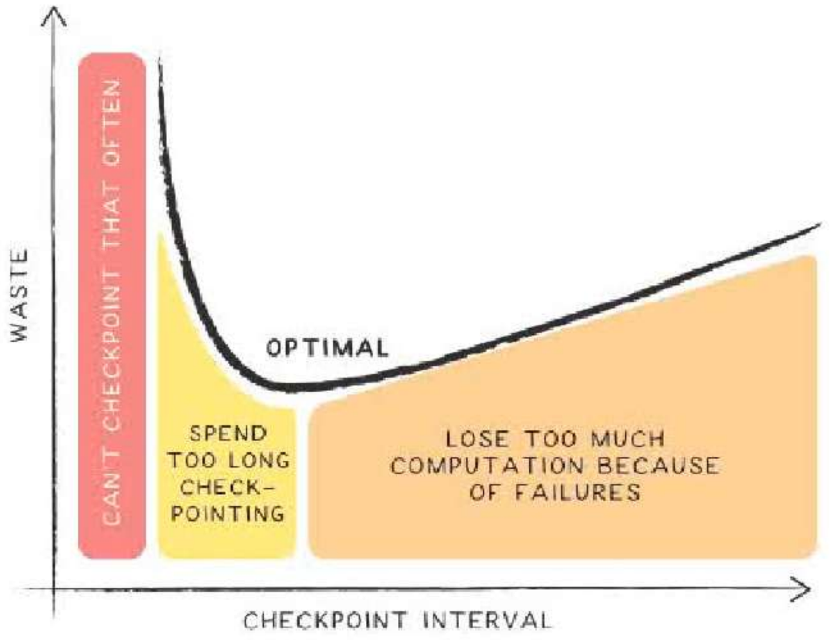
Checkpointing and Recovery Frequency

checkpointing



recovery




WASTE
CAN'T CHECKPOINT THAT OFTEN
OPTIMAL
SPEND TOO LONG CHECK-POINTING
LOSE TOO MUCH COMPUTATION BECAUSE OF FAILURES
CHECKPOINT INTERVAL

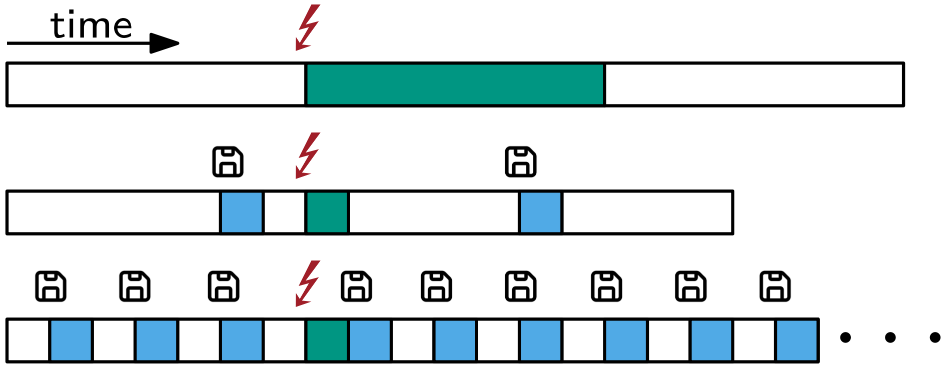
Source: Benoit et al. *Checkpointing à la Young/Daly: An Overview*

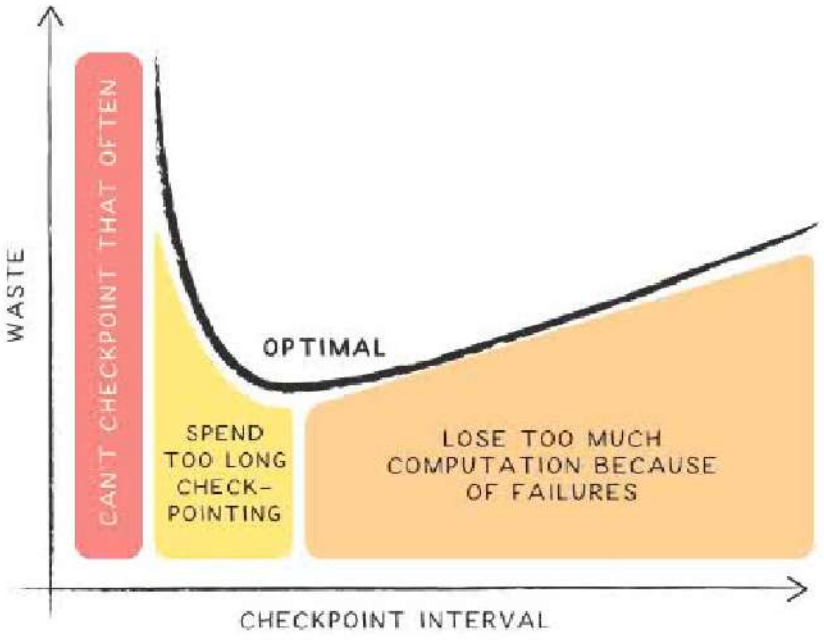
□ working 🟩 redundant work

🟦 checkpointing 🟪 recovering

Checkpointing and Recovery Frequency

checkpointing

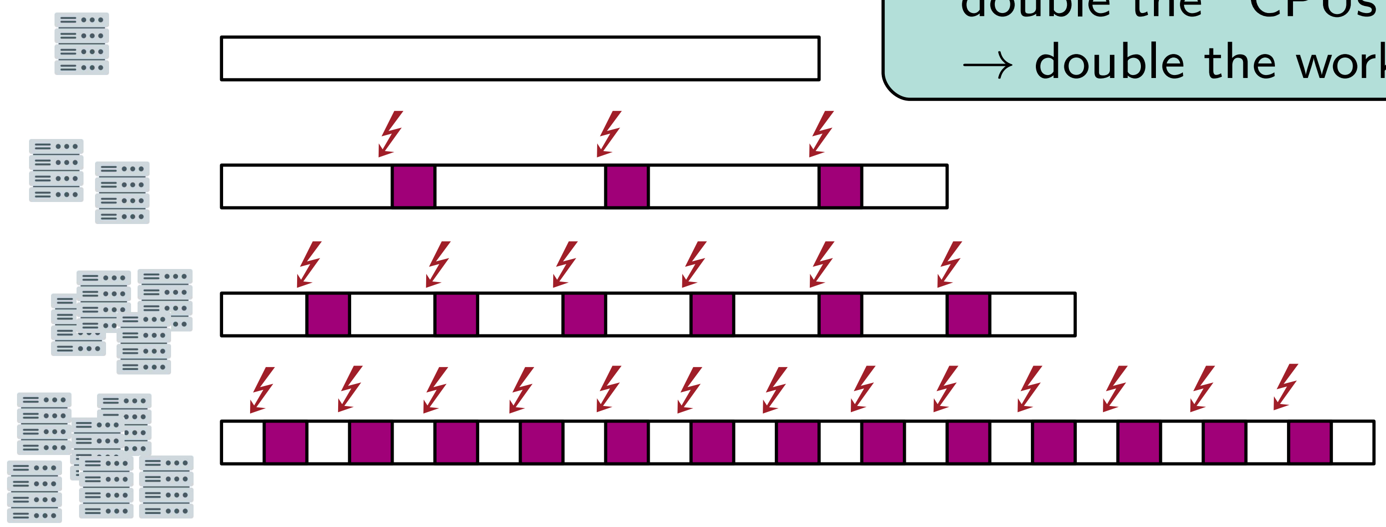




Source: Benoit et al. *Checkpointing à la Young/Daly: An Overview*

recovery

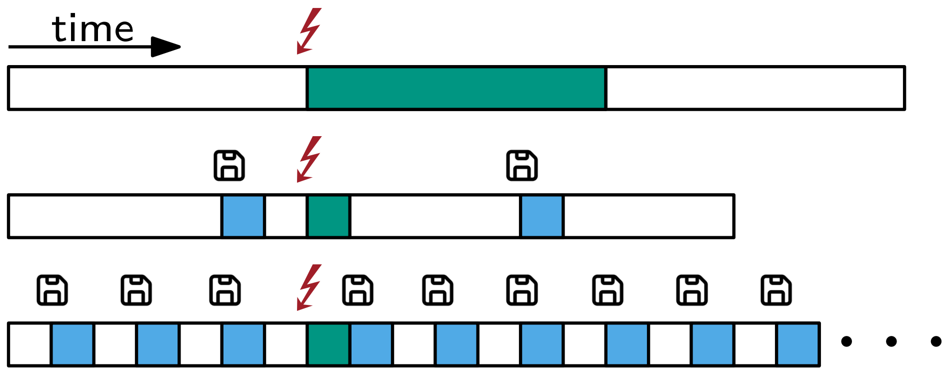
here: **weak scaling**
double the "CPUs"
→ double the work

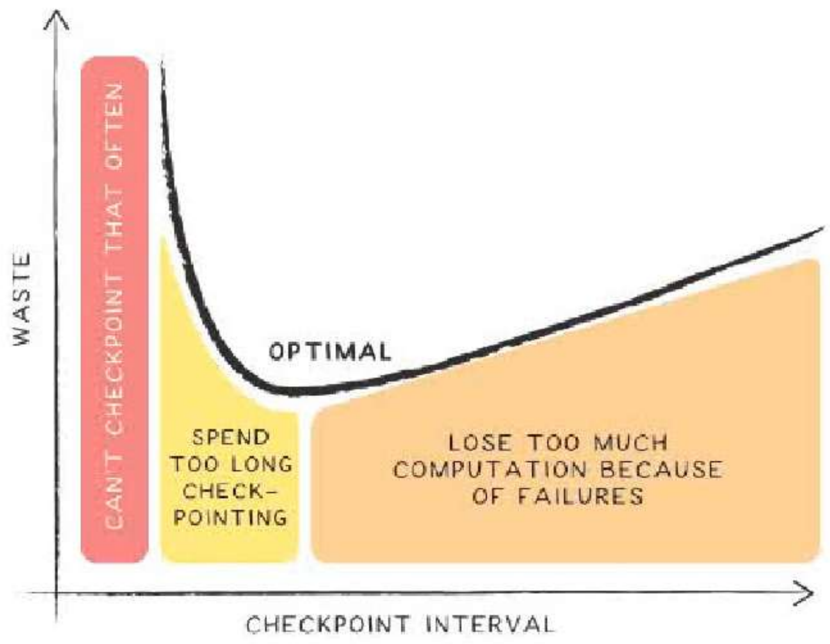


□ working
🟦 checkpointing
🟩 redundant work
🟪 recovering

Checkpointing and Recovery Frequency

checkpointing

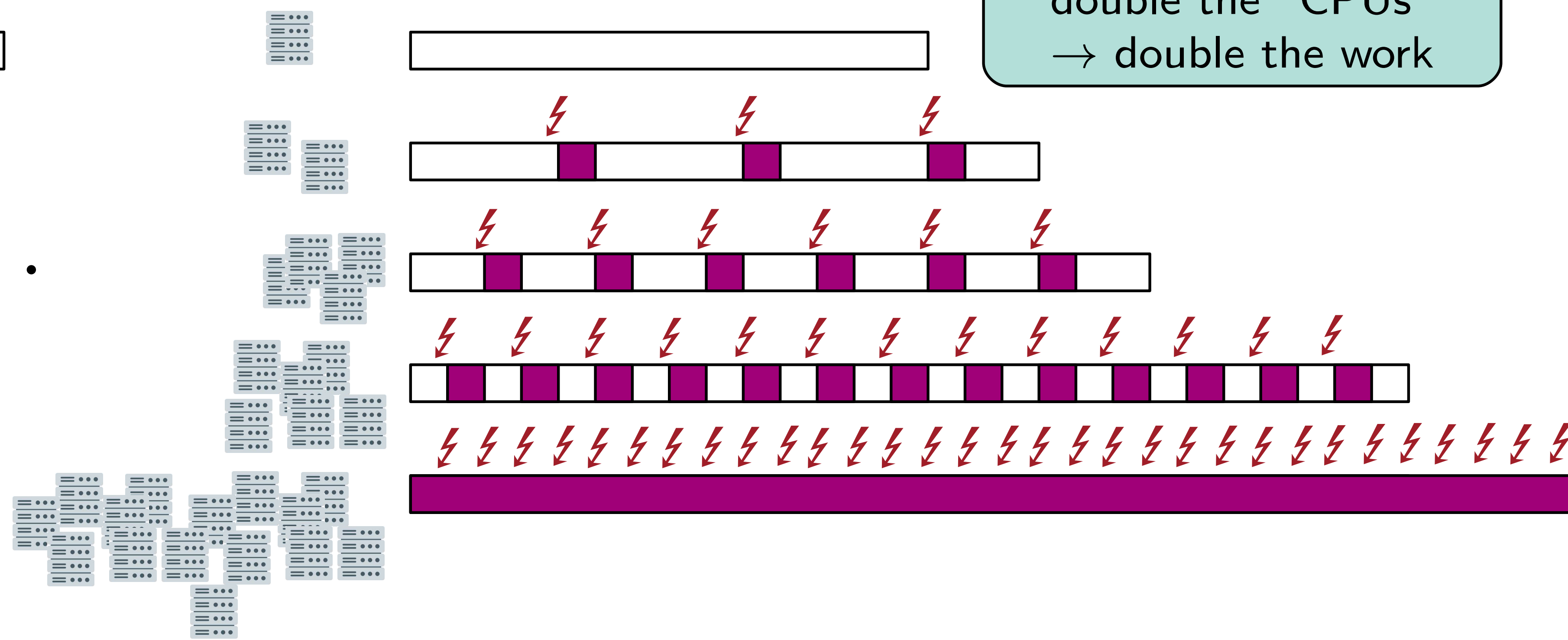




Source: Benoit et al. *Checkpointing à la Young/Daly: An Overview*

recovery

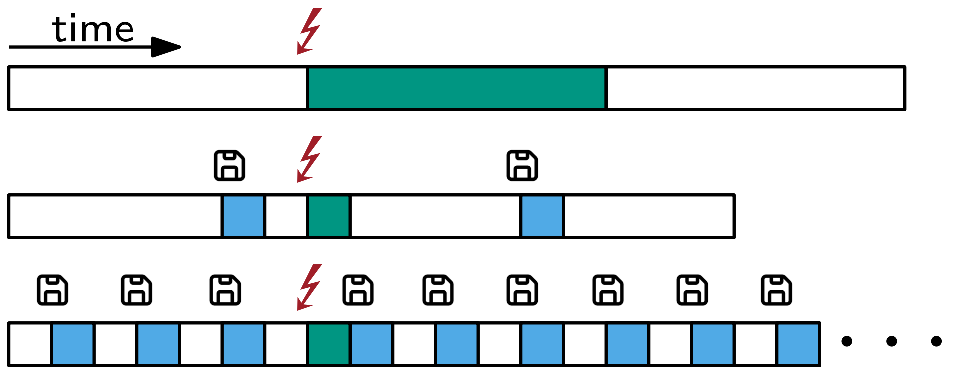
here: **weak scaling**
double the "CPUs"
→ double the work

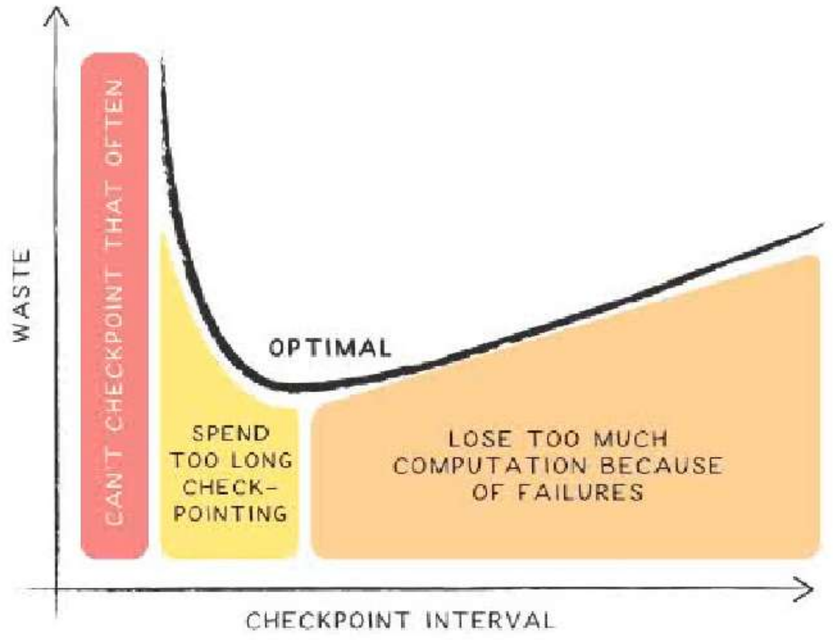


□ working
🟦 checkpointing
🟩 redundant work
🟪 recovering

Checkpointing and Recovery Frequency

checkpointing

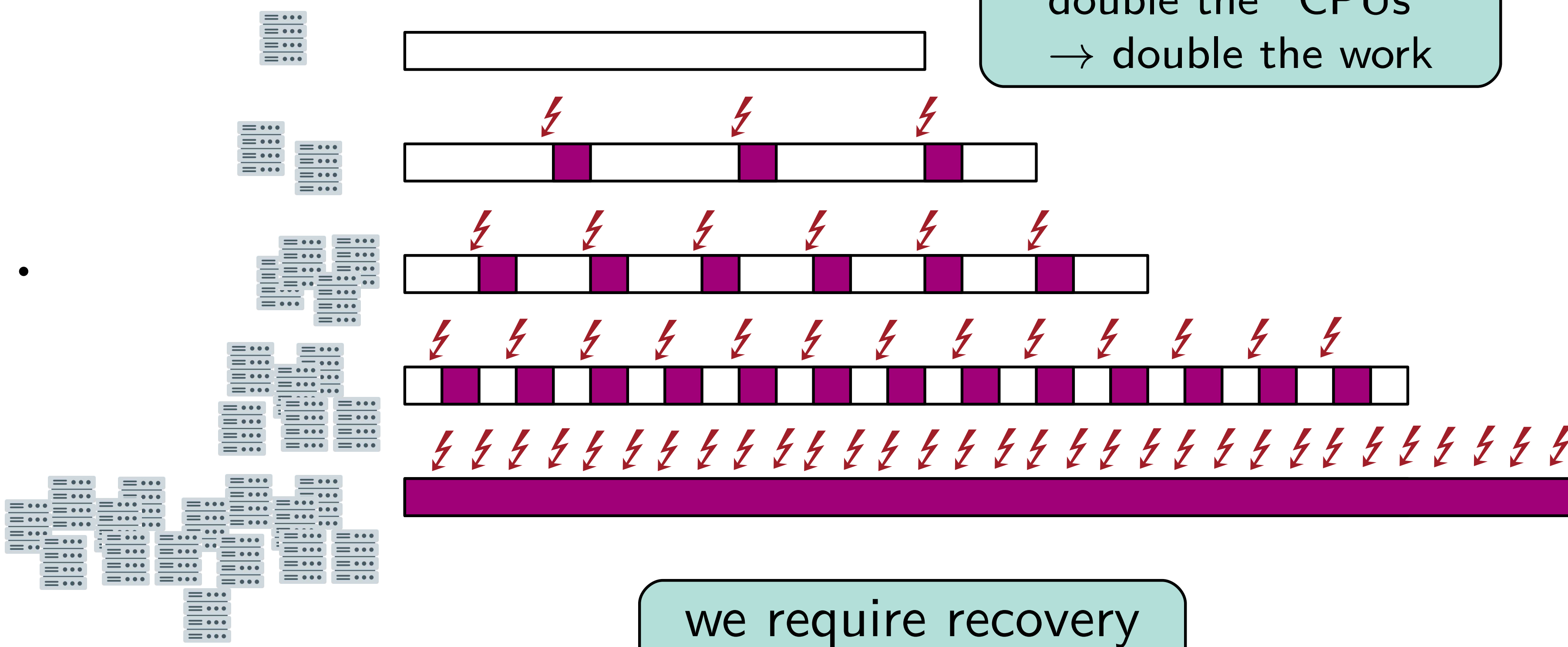




Source: Benoit et al. *Checkpointing à la Young/Daly: An Overview*

recovery

here: **weak scaling**
double the "CPUs"
→ double the work



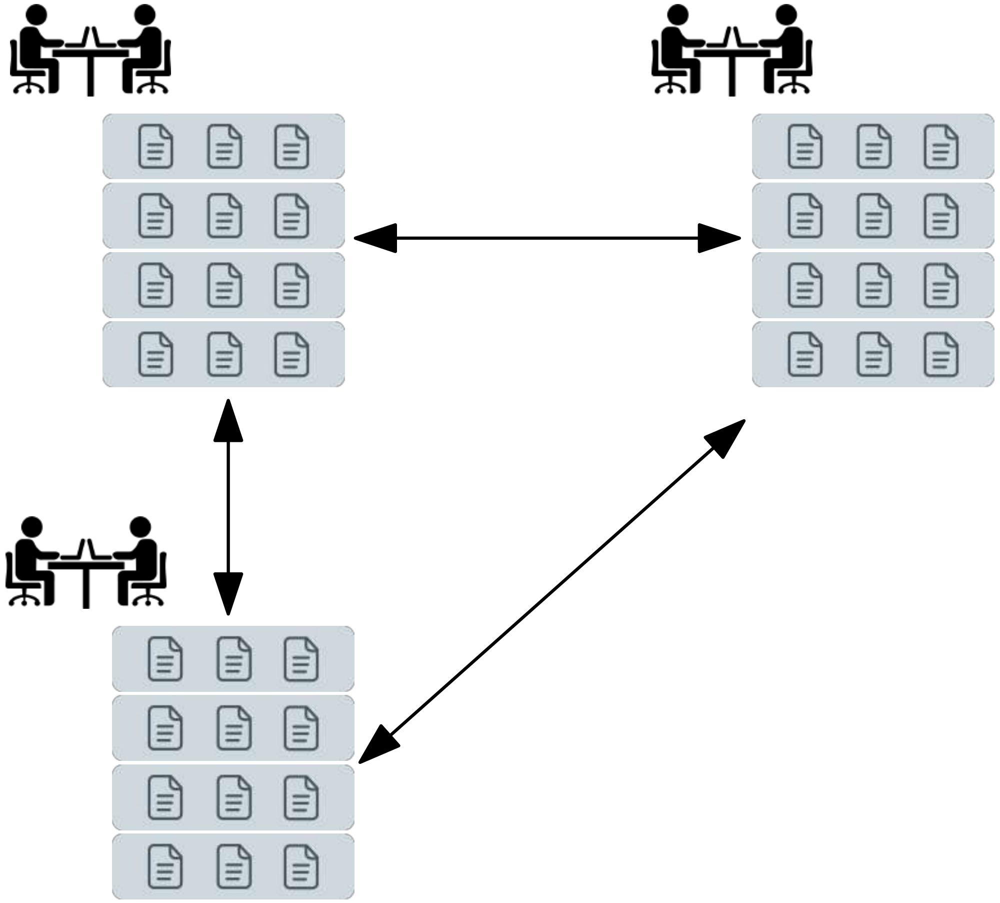
we require recovery in $\mathcal{O}(1/p)$ time

□ working 🟩 redundant work

🟦 checkpointing 🟪 recovering

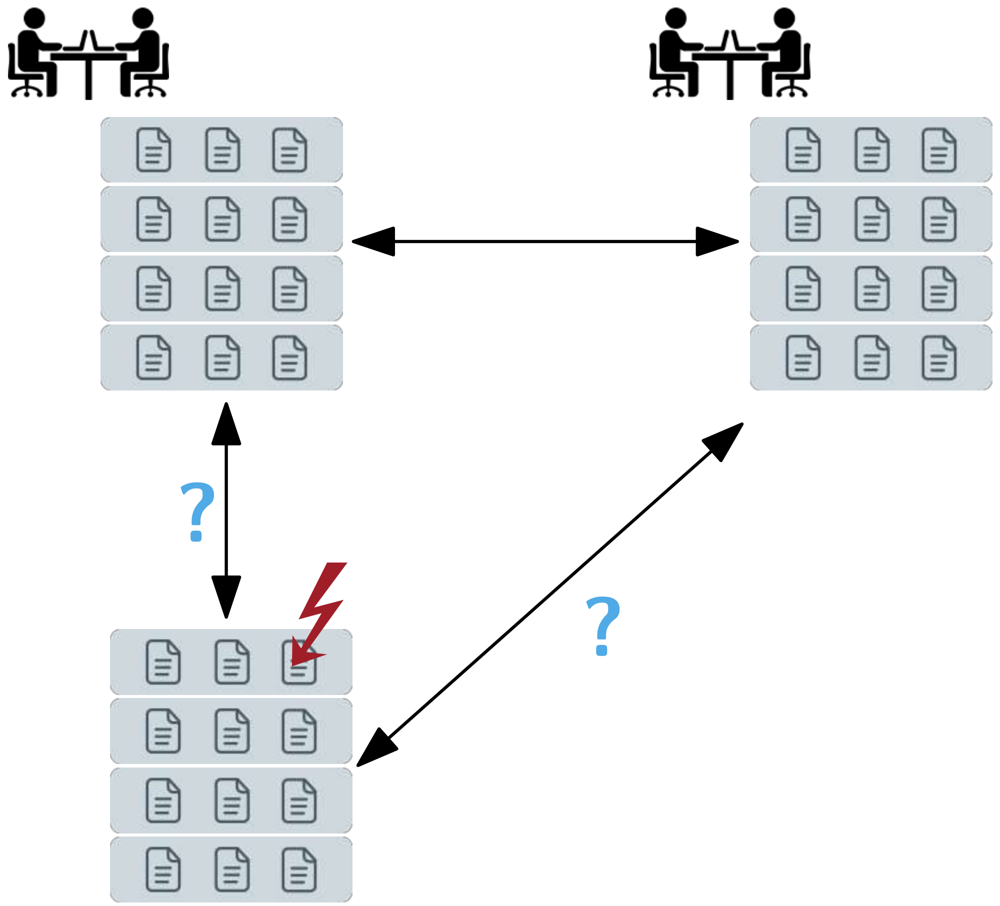
Detecing Node Failures

- User Level Failure Mitigation part of the (upcoming) MPI standard
- Already **implemented** in OpenMPI
- **Fail-stop** model



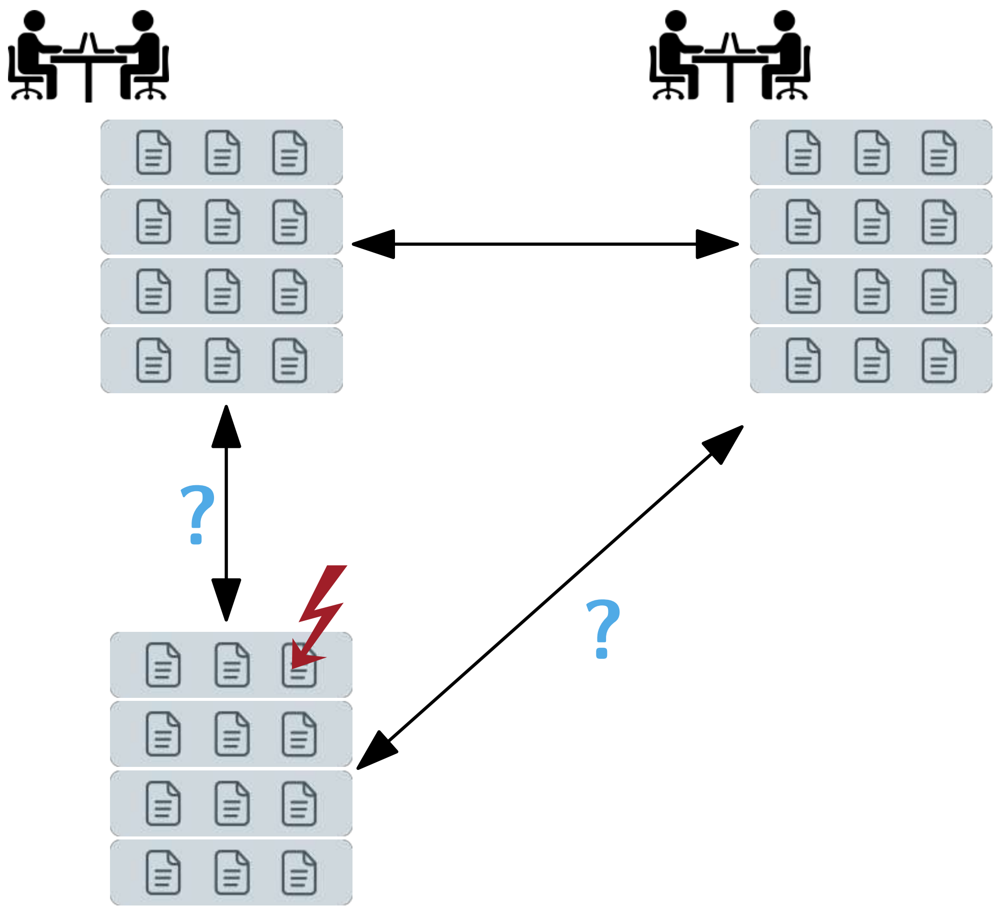
Detecing Node Failures

- User Level Failure Mitigation part of the (upcoming) MPI standard
- Already **implemented** in OpenMPI
- **Fail-stop** model



Detecing Node Failures

- User Level Failure Mitigation part of the (upcoming) MPI standard
- Already **implemented** in OpenMPI
- **Fail-stop** model

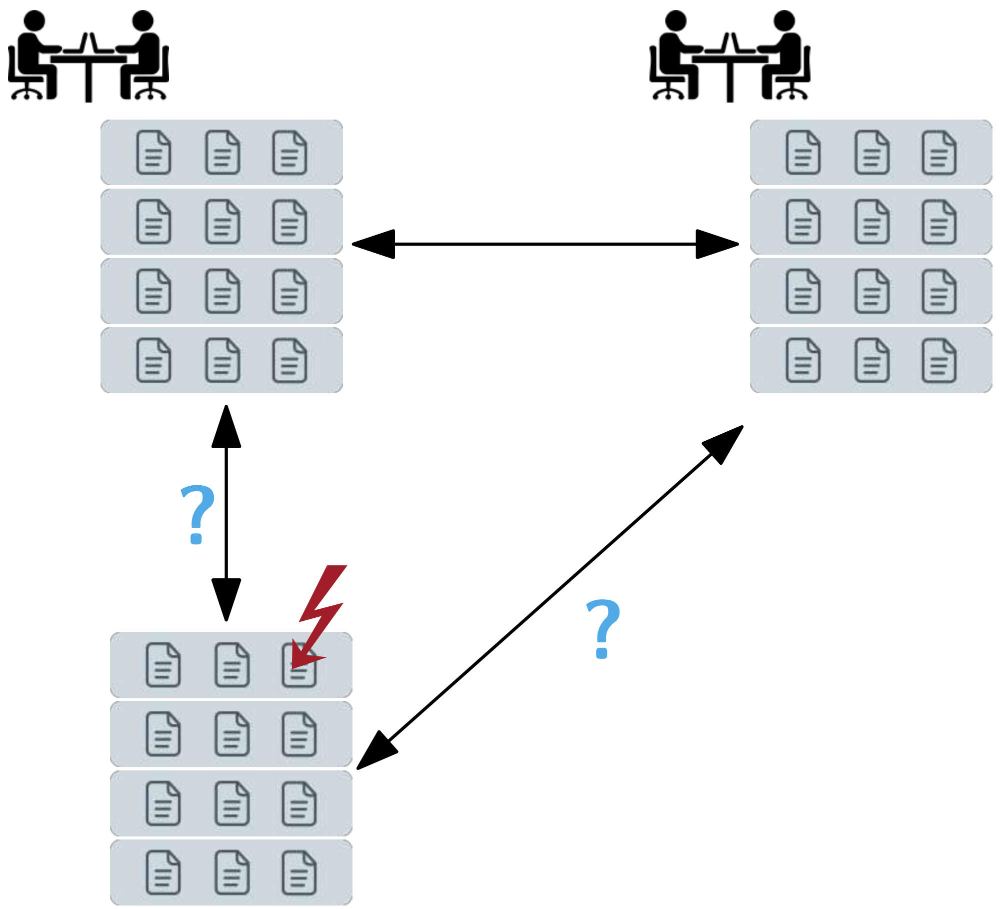


MPI will

- detect node-failures (heartbeat signals)
- repair the communicator
- (reluctantly) tell you which nodes failed

Detecing Node Failures

- User Level Failure Mitigation part of the (upcoming) MPI standard
- Already **implemented** in OpenMPI
- **Fail-stop** model



MPI will

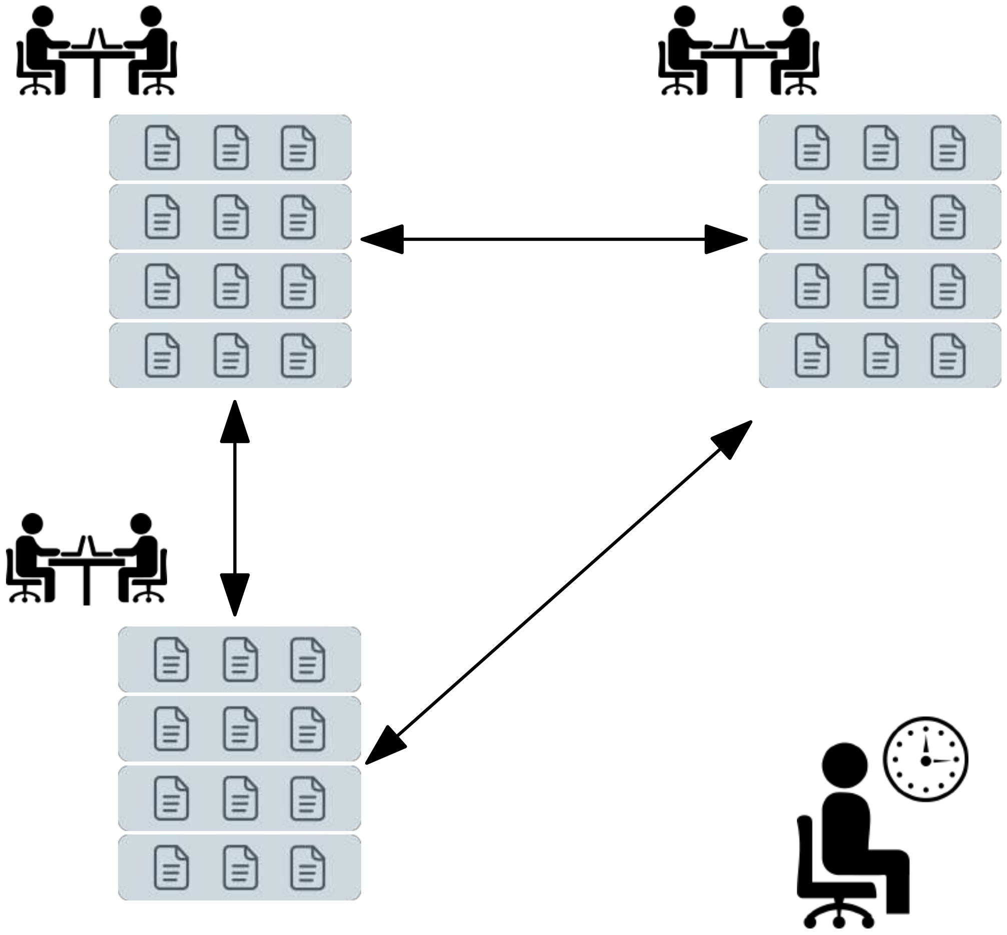
- detect node-failures (heartbeat signals)
- repair the communicator
- (reluctantly) tell you which nodes failed

You have to

- Recover data
- Roll-back you application
- Re-distribute work
- Aquire replacement nodes

Shrinking vs Substituting Recovery

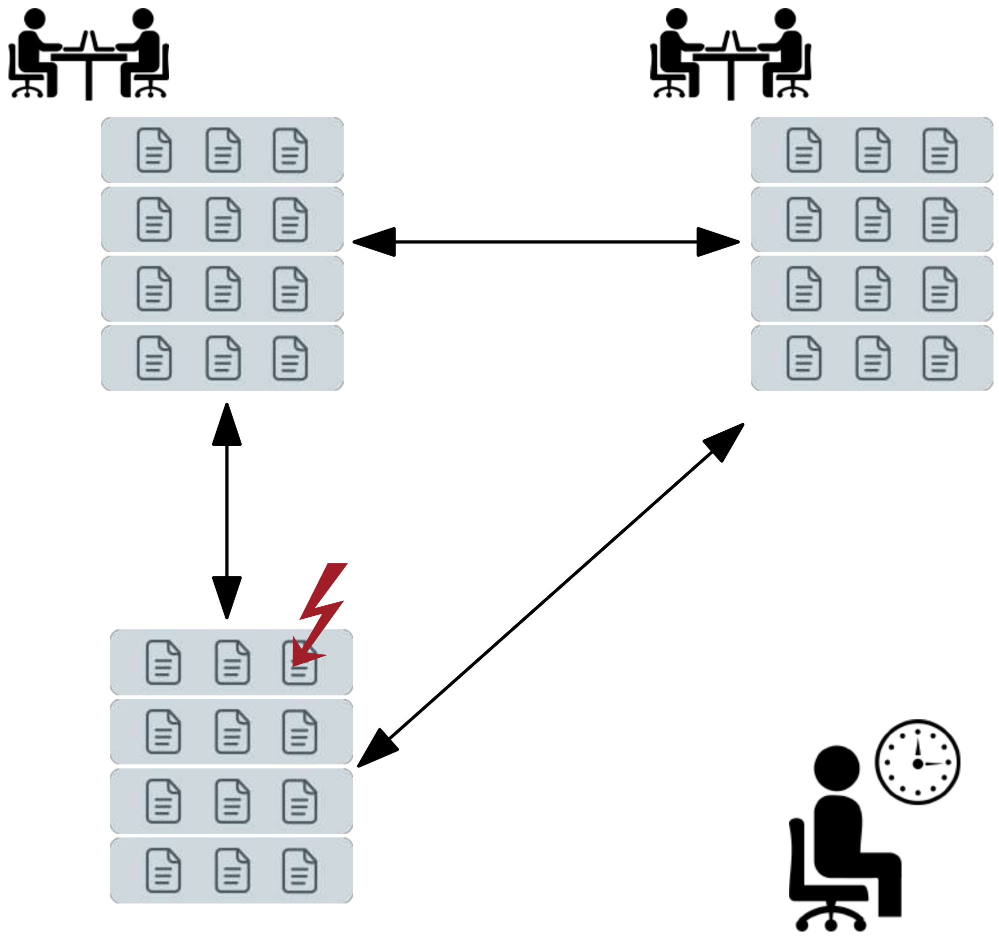
Substituting Recovery



Shrinking Recovery

Shrinking vs Substituting Recovery

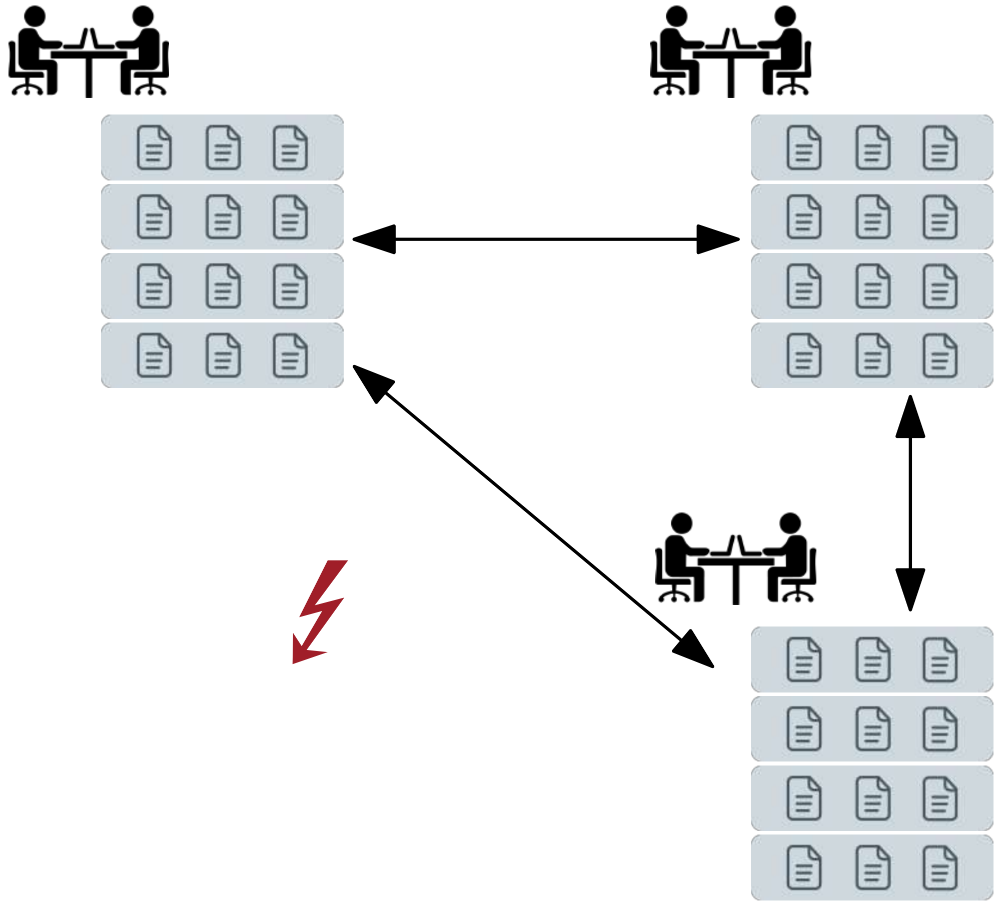
Substituting Recovery



Shrinking Recovery

Shrinking vs Substituting Recovery

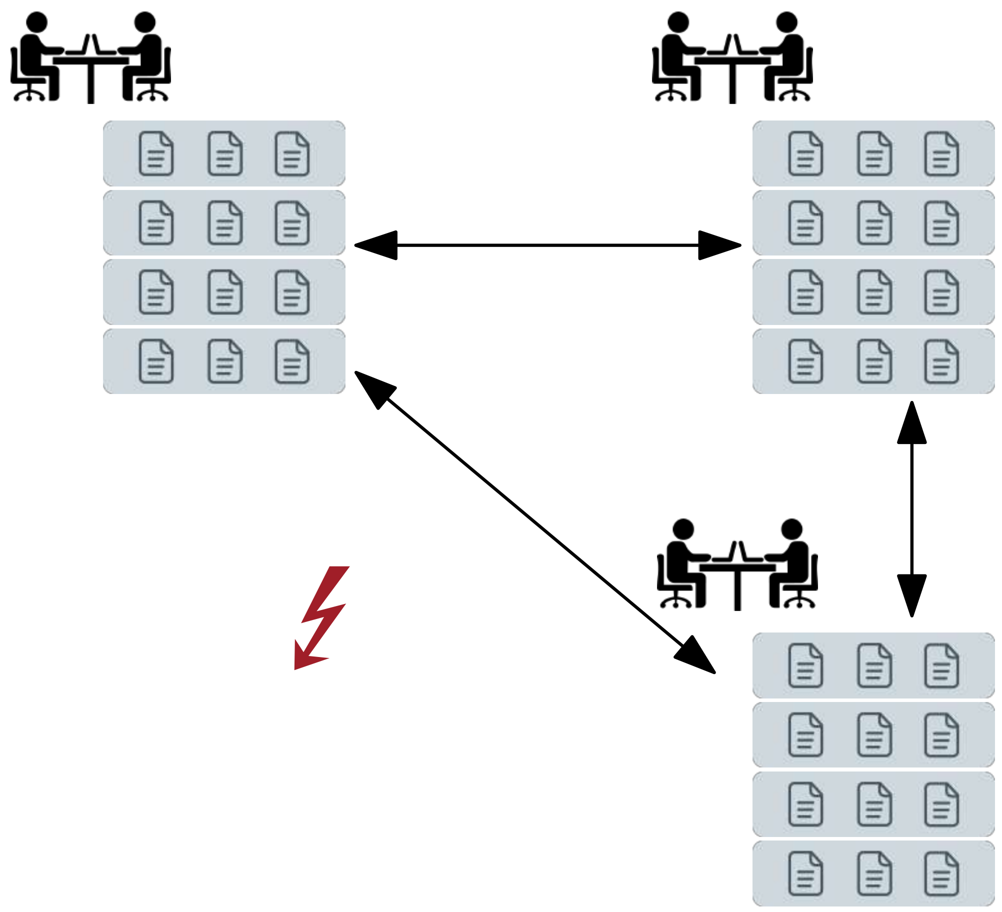
Substituting Recovery



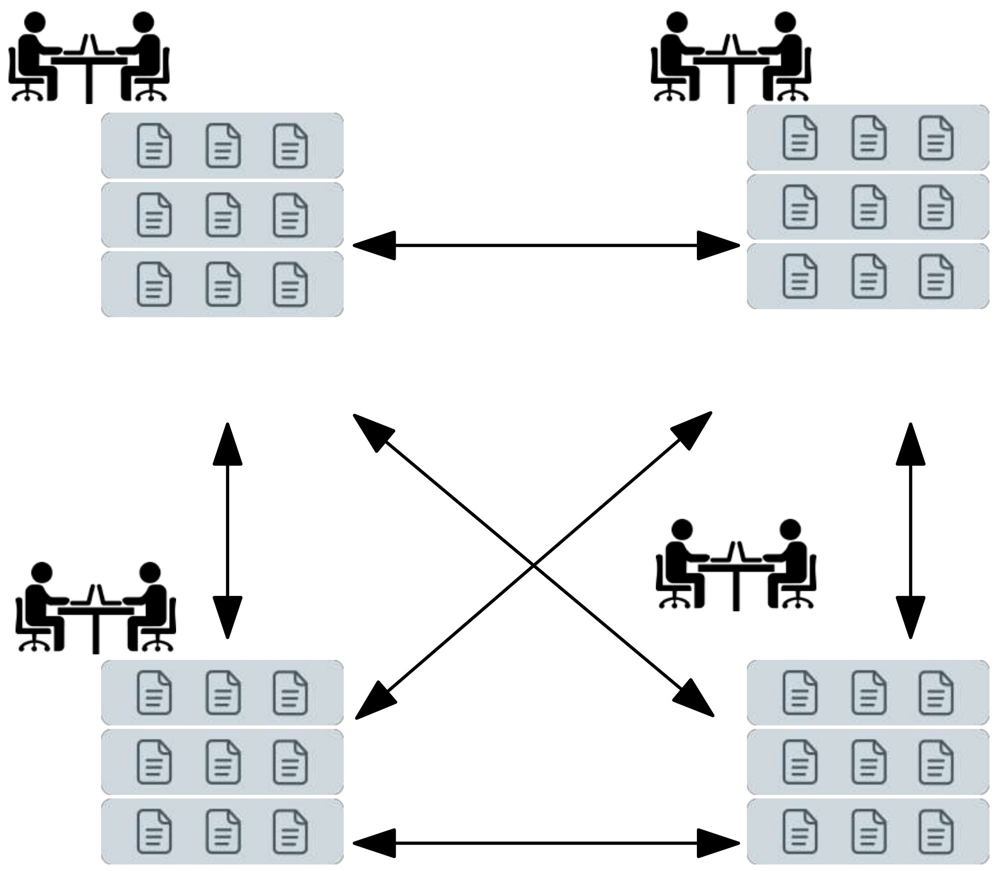
Shrinking Recovery

Shrinking vs Substituting Recovery

Substituting Recovery

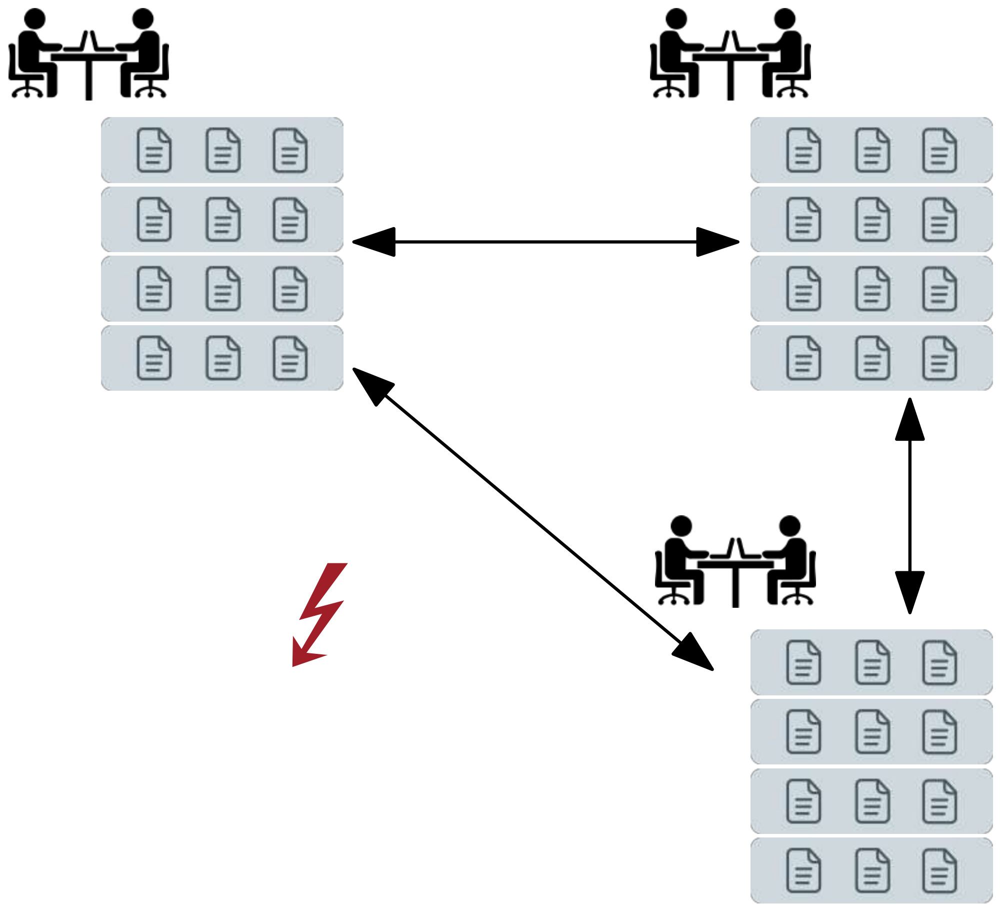


Shrinking Recovery

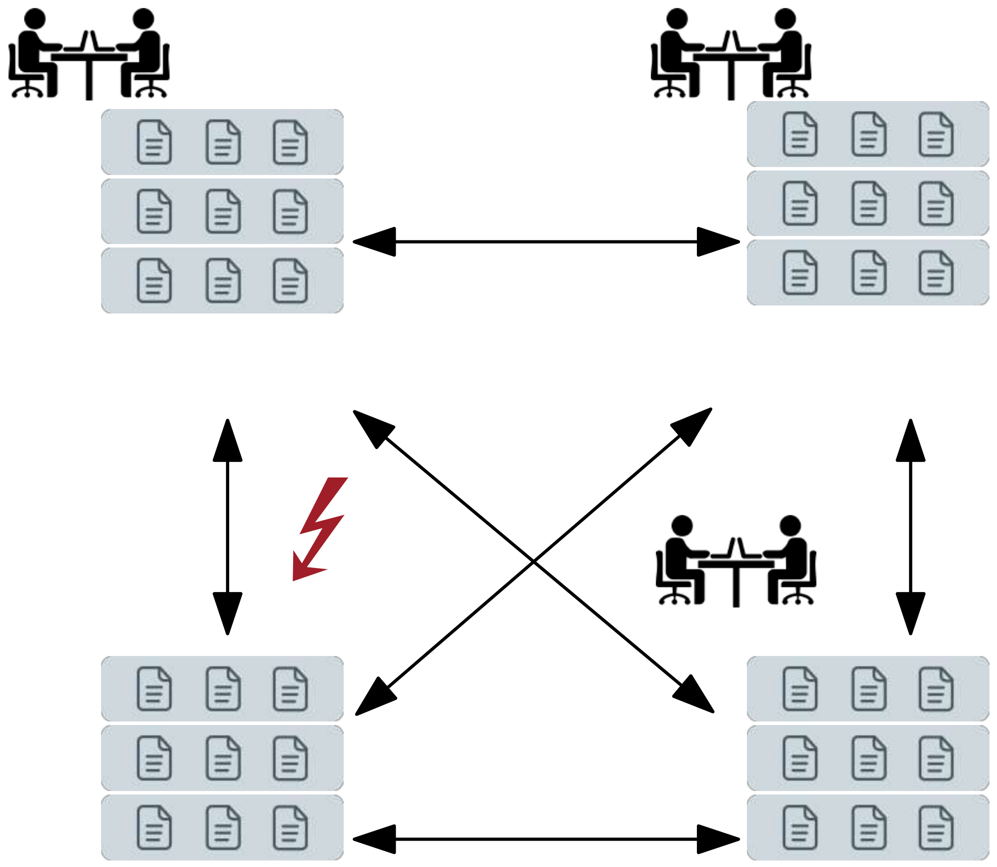


Shrinking vs Substituting Recovery

Substituting Recovery

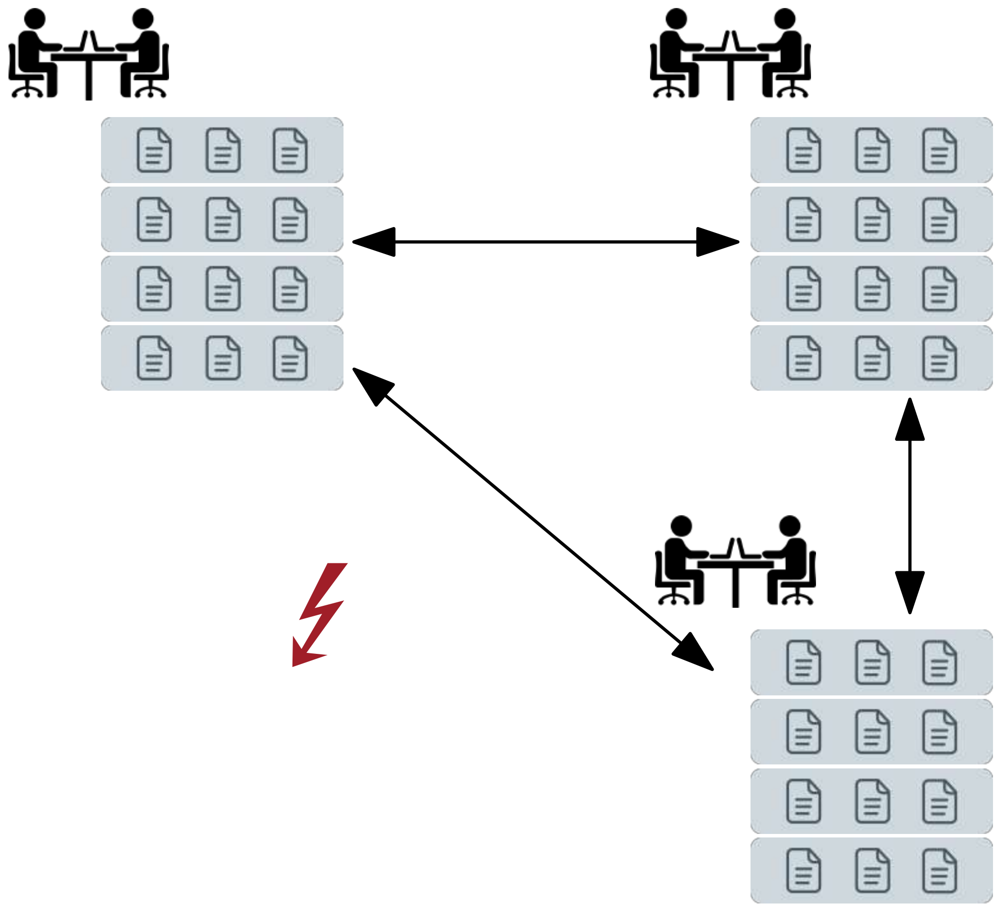


Shrinking Recovery

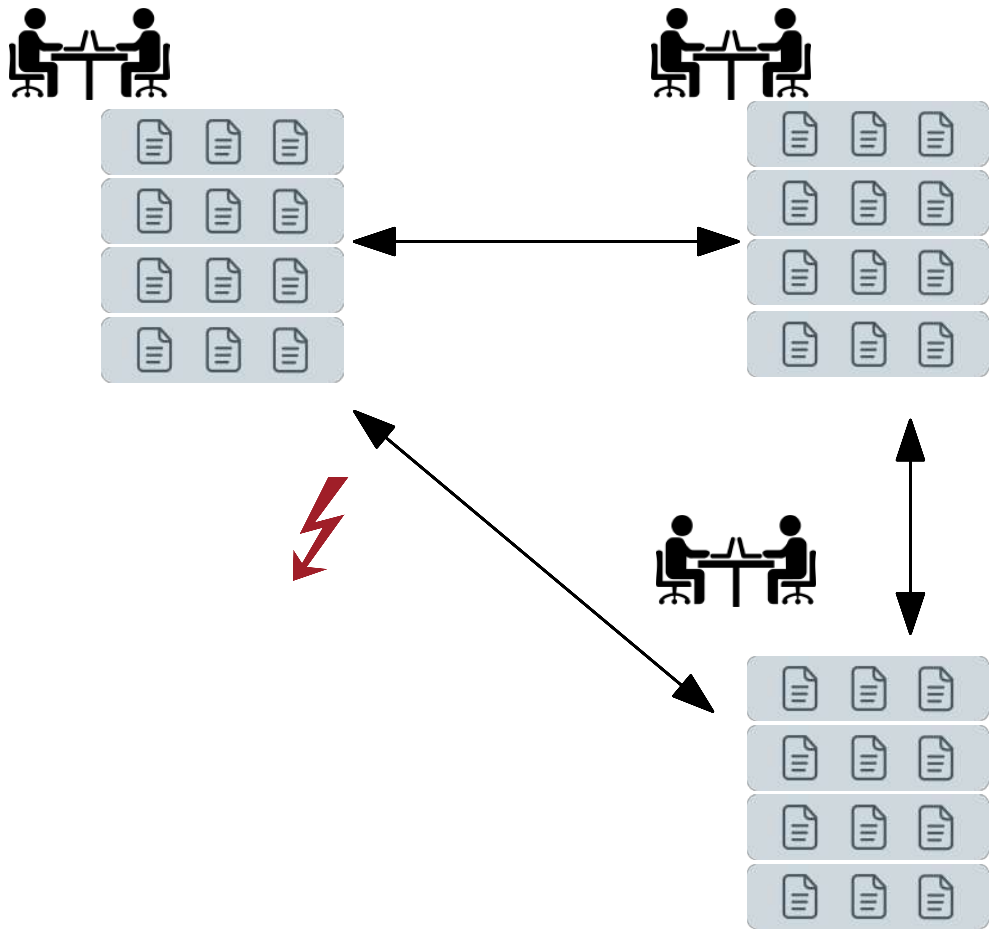


Shrinking vs Substituting Recovery

Substituting Recovery

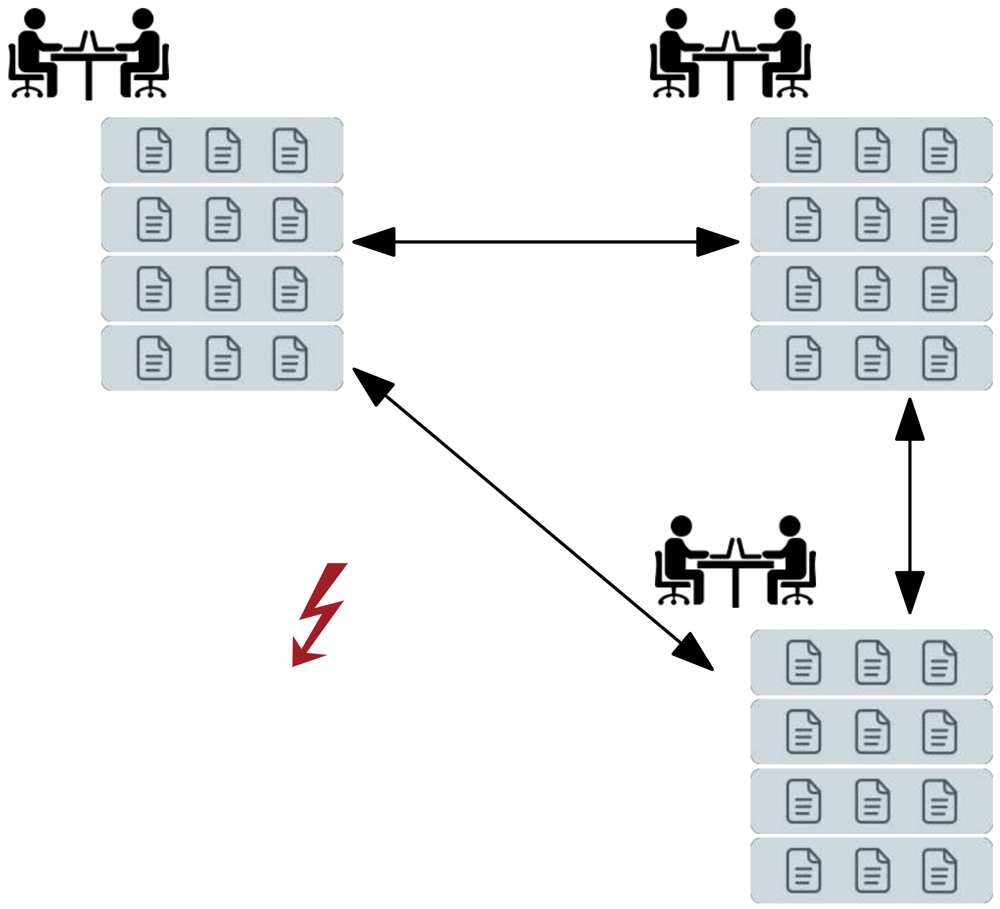


Shrinking Recovery



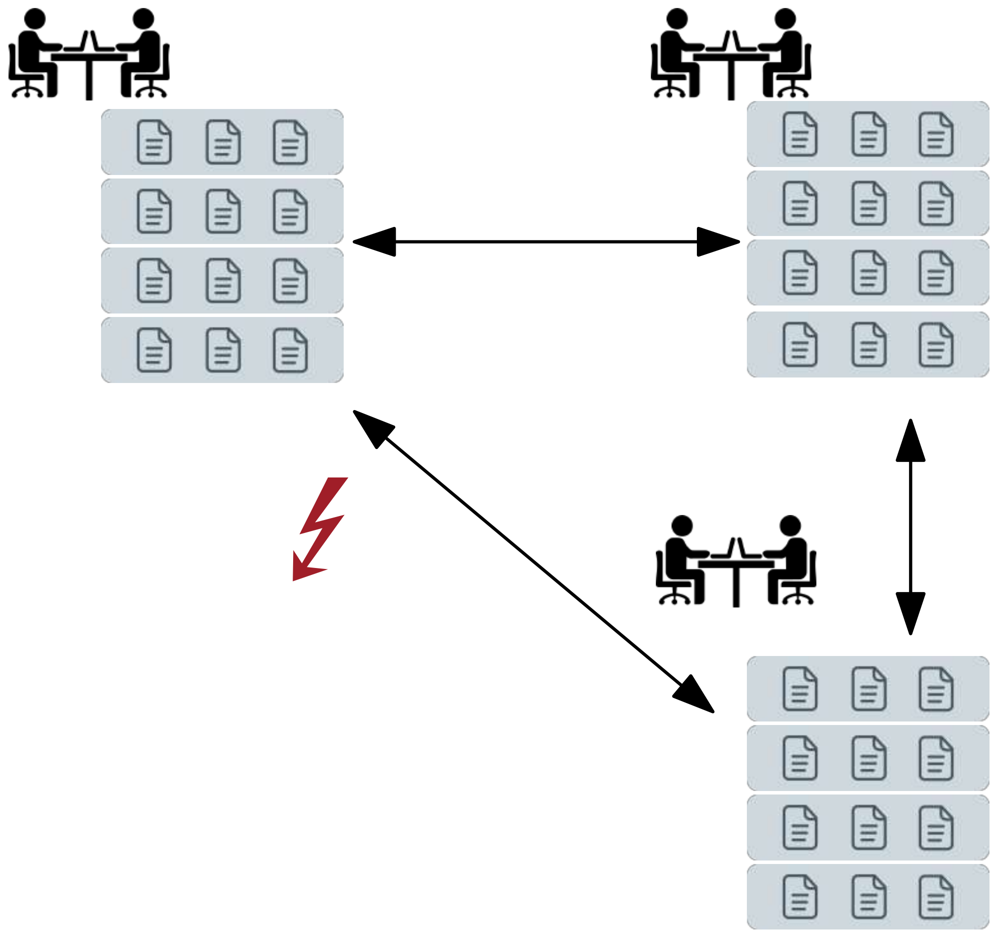
Shrinking vs Substituting Recovery

Substituting Recovery



- Up to $5\,\%$ of nodes idling
- Limited number of failures supported
- Recovery time does not scale

Shrinking Recovery

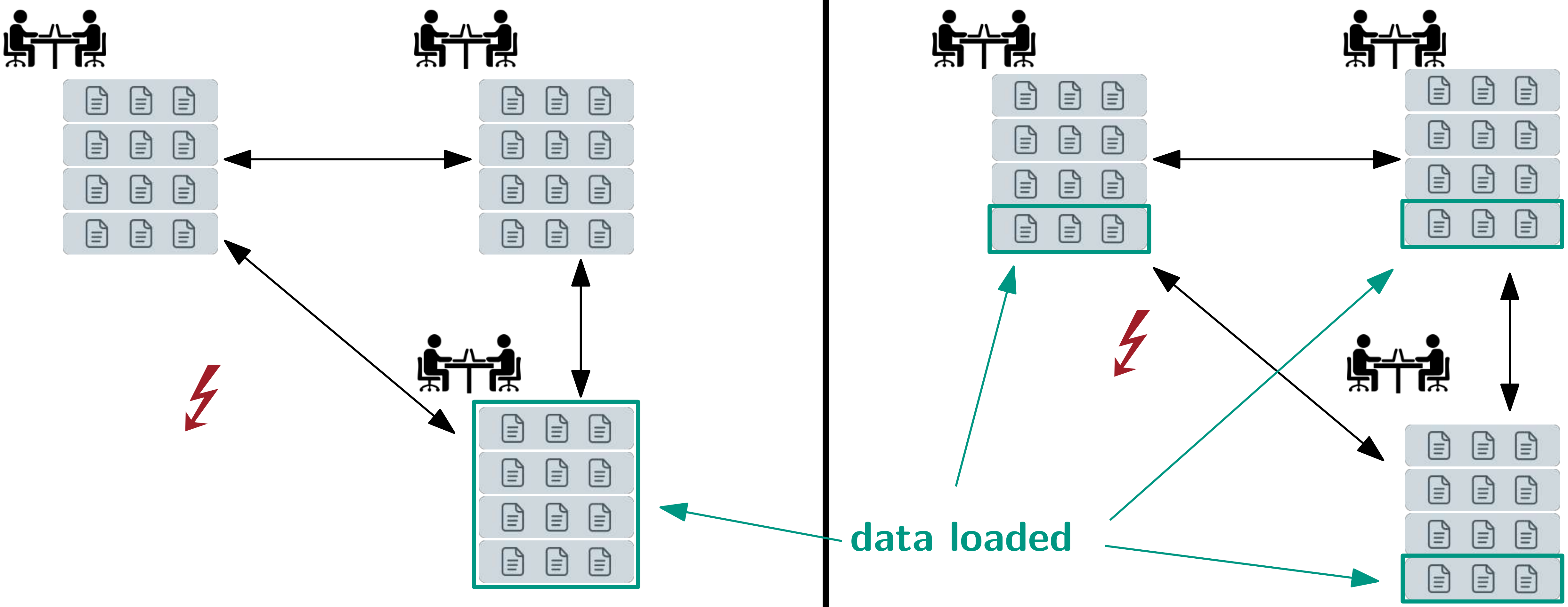


- All nodes participate in computation
- Unlimited number of failures supported
- Recovery time scales with $1/p$

Shrinking vs Substituting Recovery

Substituting Recovery

Shrinking Recovery



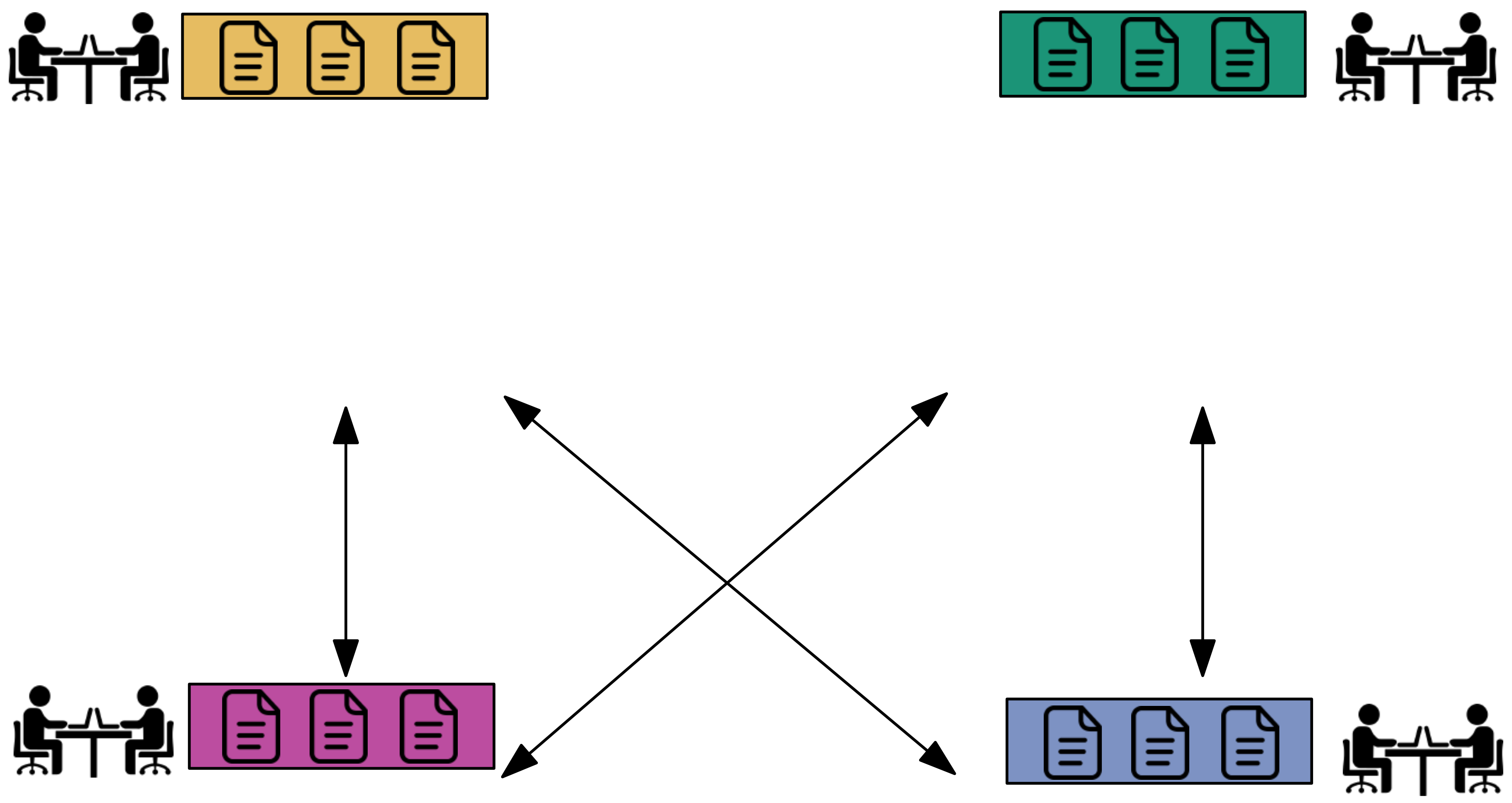
single node receives all messages
→ bottleneck

Design Goals

ReStore

in-memory	access to the parallel file system is a bottleneck
no spare nodes	spare nodes are wasted resources
no checkpointing nodes	checkpoint nodes are wasted resources
scalable recovery	$\in \mathcal{O}(1/p)$ time per failure
arbitrary replication level	more flexibility and robustness

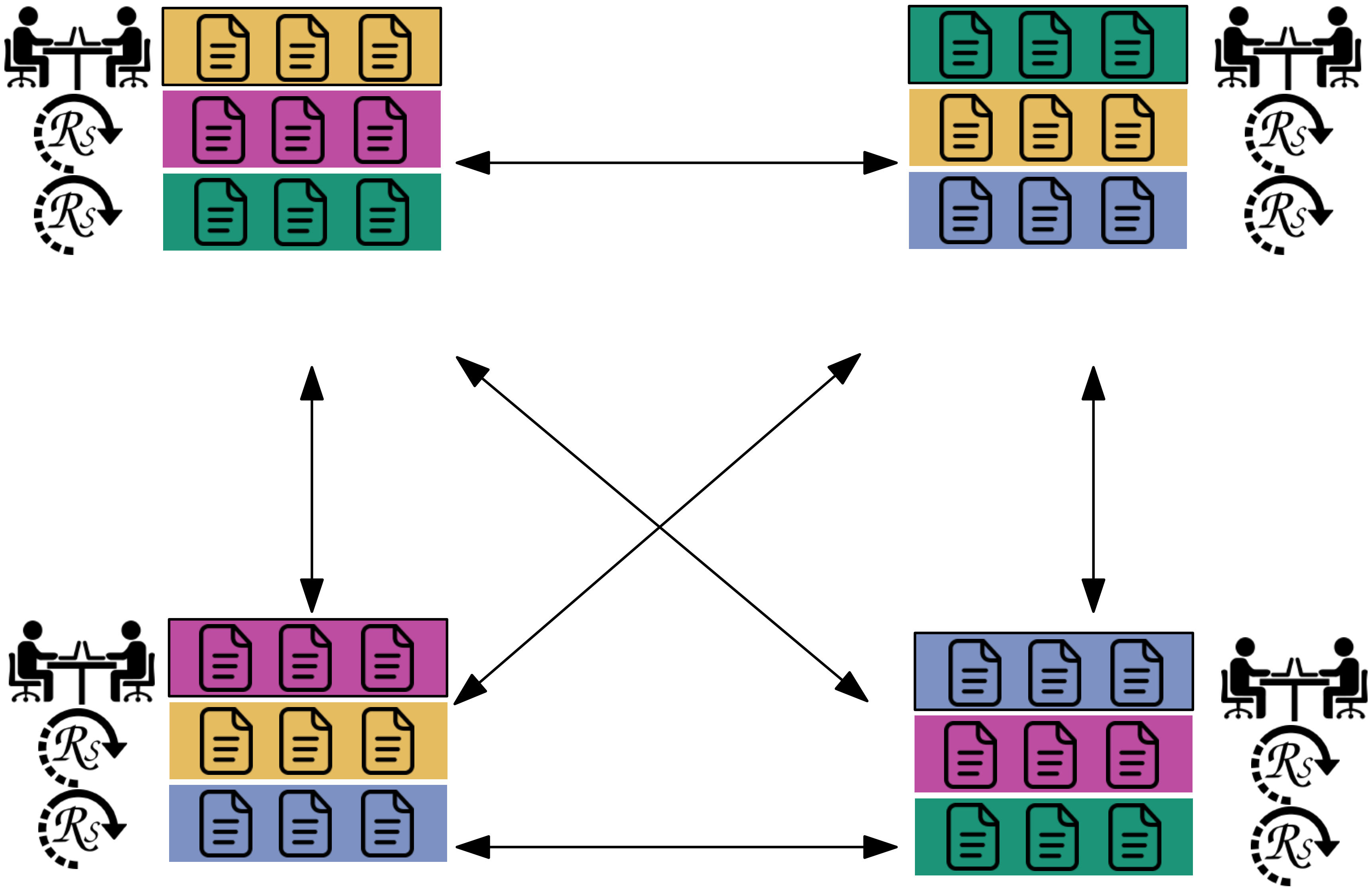
Naïve Data Distribution



- Data distributed across CPUs
- Data divided into blocks 🗎

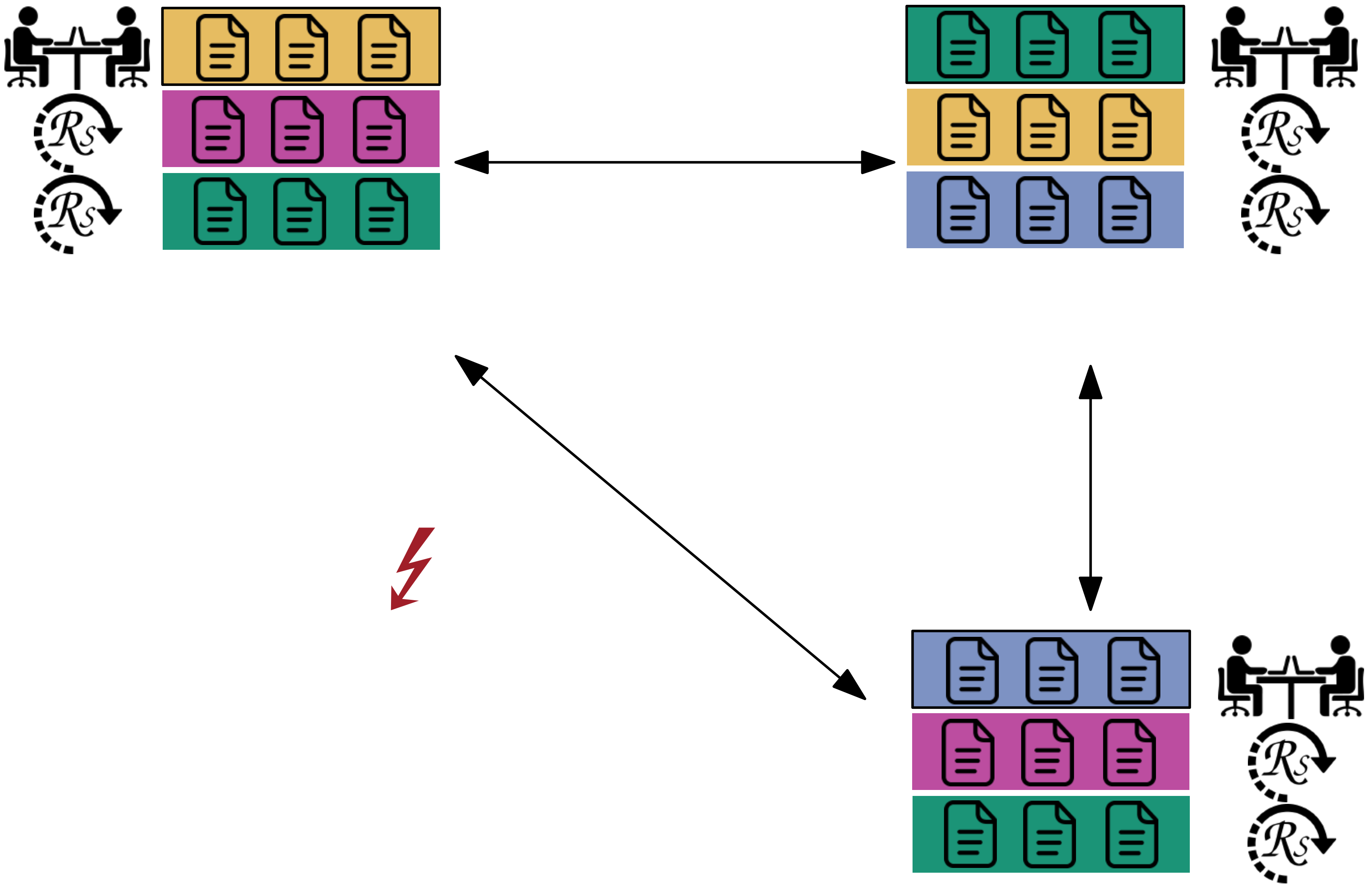
Naïve Data Distribution

- Data distributed across CPUs
- Data divided into blocks 🗎
- Additionally store replicas



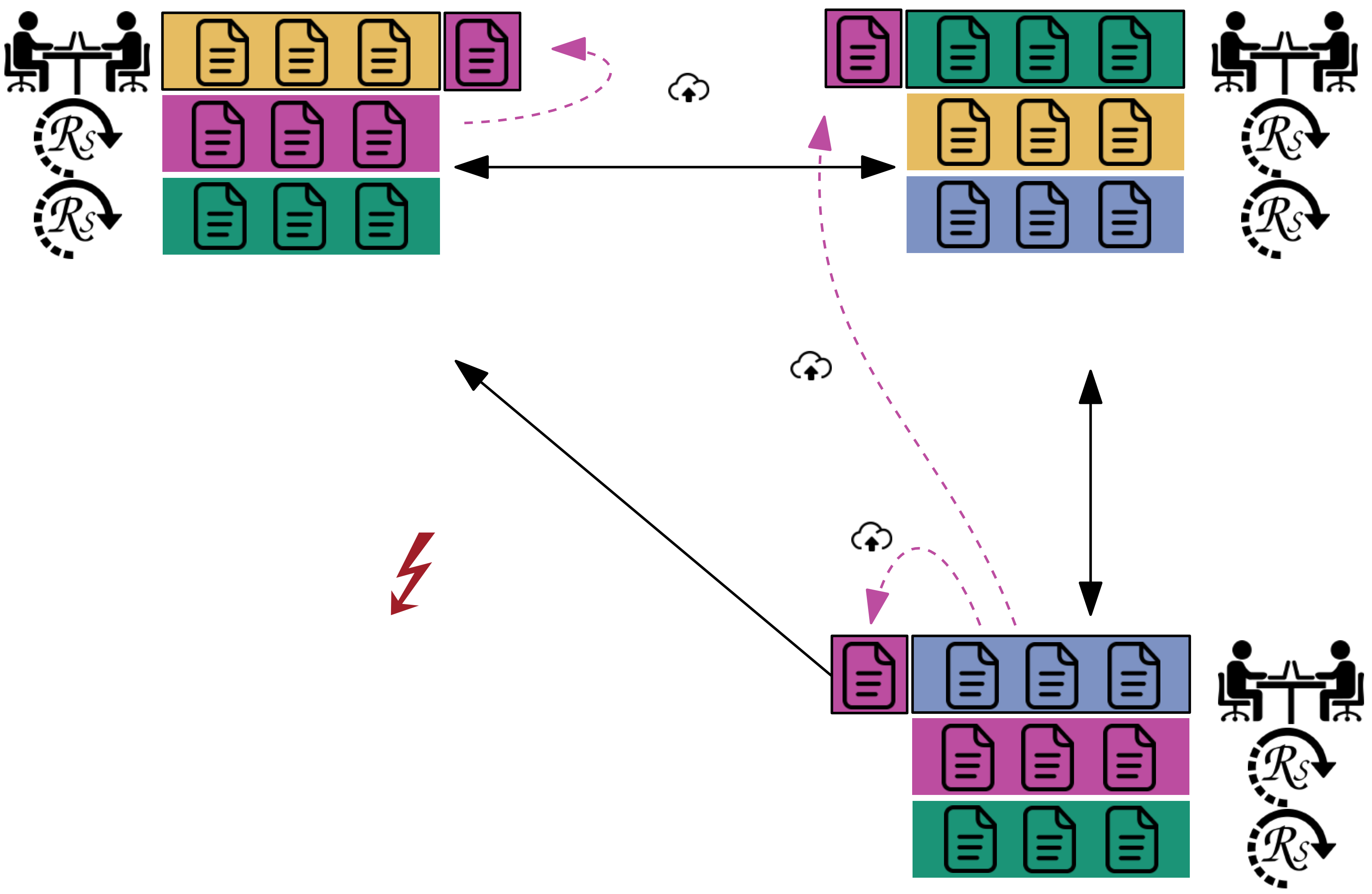
Naïve Data Distribution

- Data distributed across CPUs
- Data divided into blocks 🗎
- Additionally store replicas



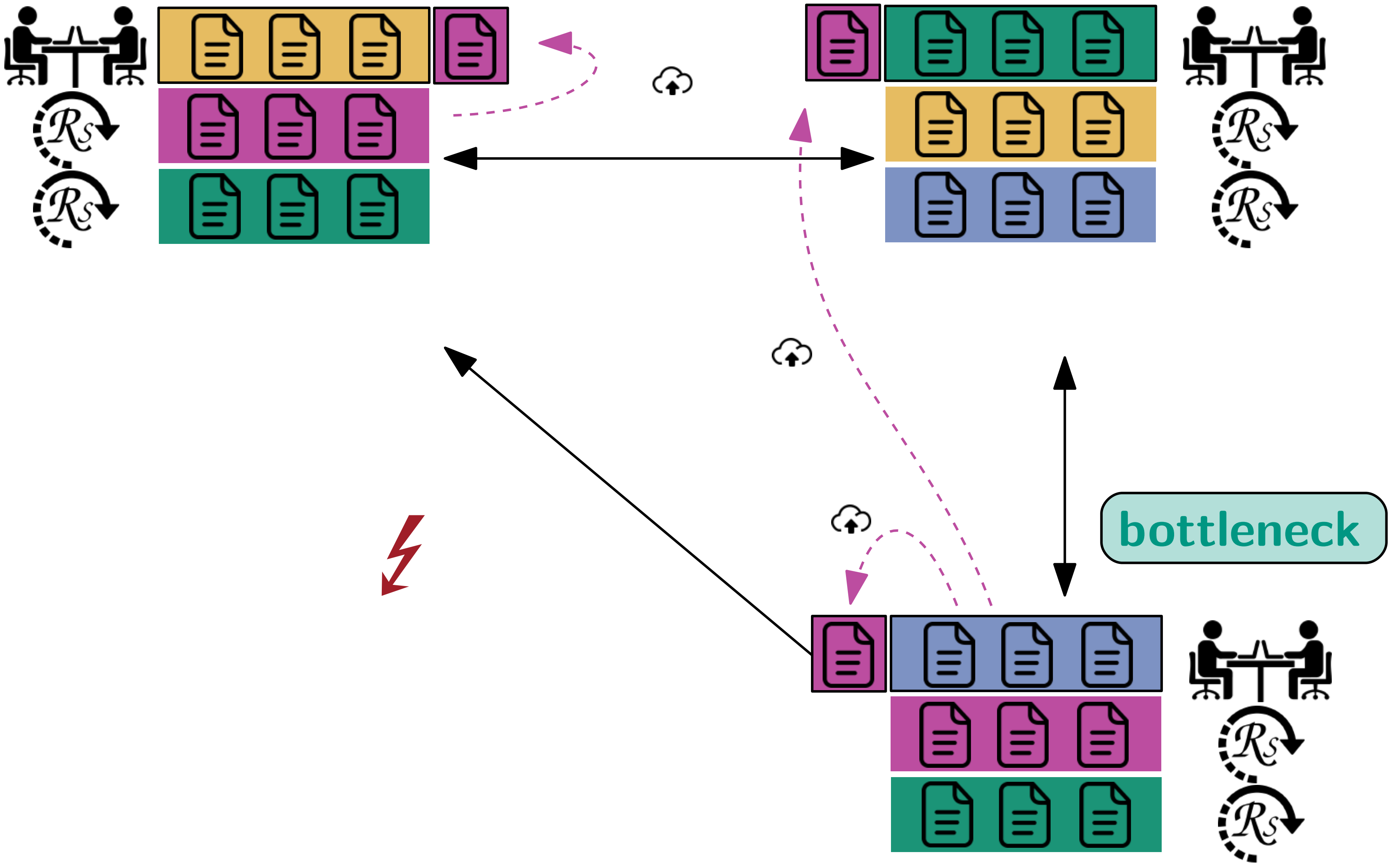
Naïve Data Distribution

- Data distributed across CPUs
- Data divided into blocks 🗎
- Additionally store replicas



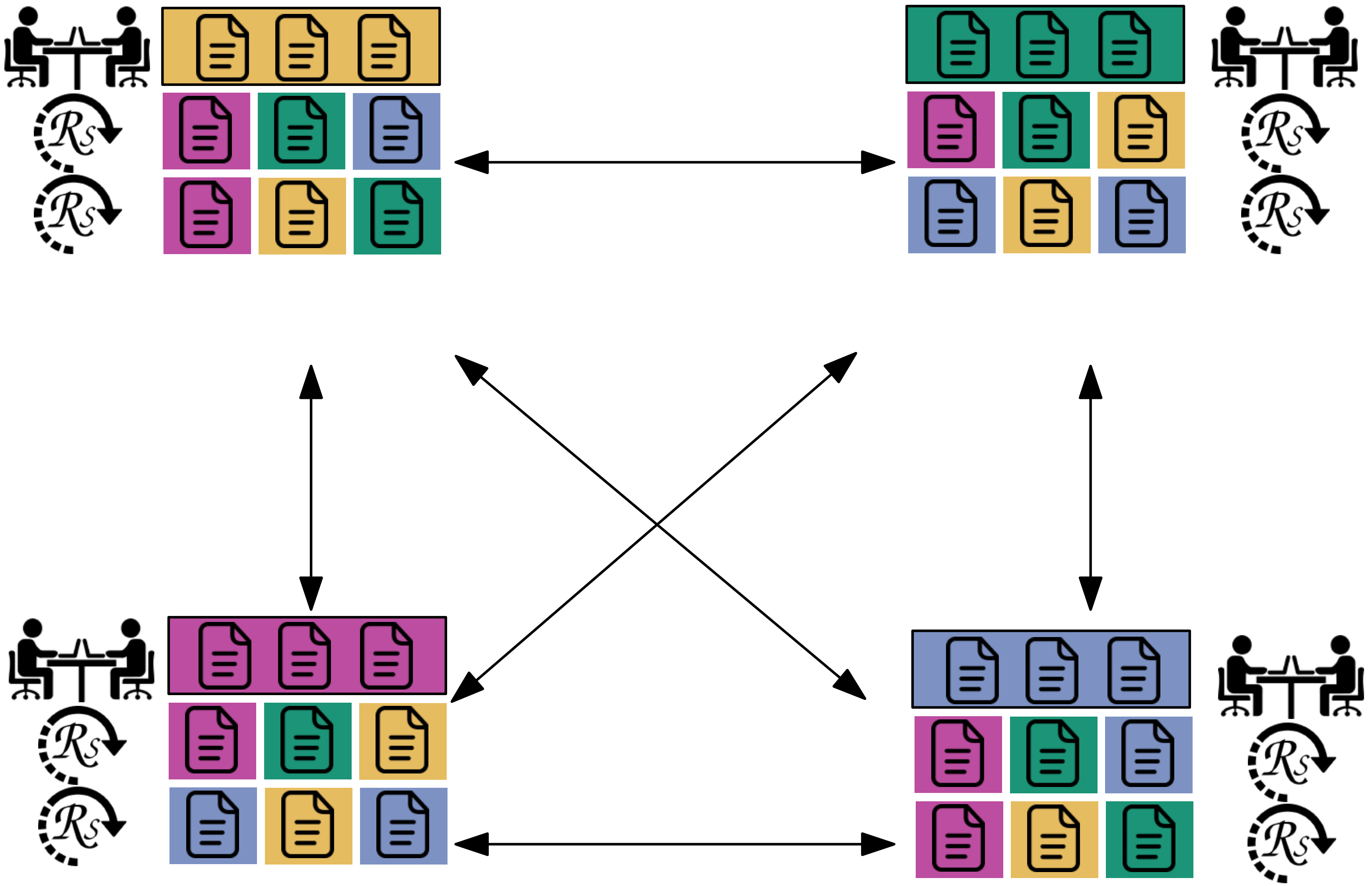
Naïve Data Distribution

- Data distributed across CPUs
- Data divided into blocks 🗎
- Additionally store replicas



Data Distribution for Faster Recovery

- **break up access pattern by randomly distributing blocks**
  → more PEs serving data
- too many PEs serving data
  → messages too small
- empirical optimum: permute 256 KiB together

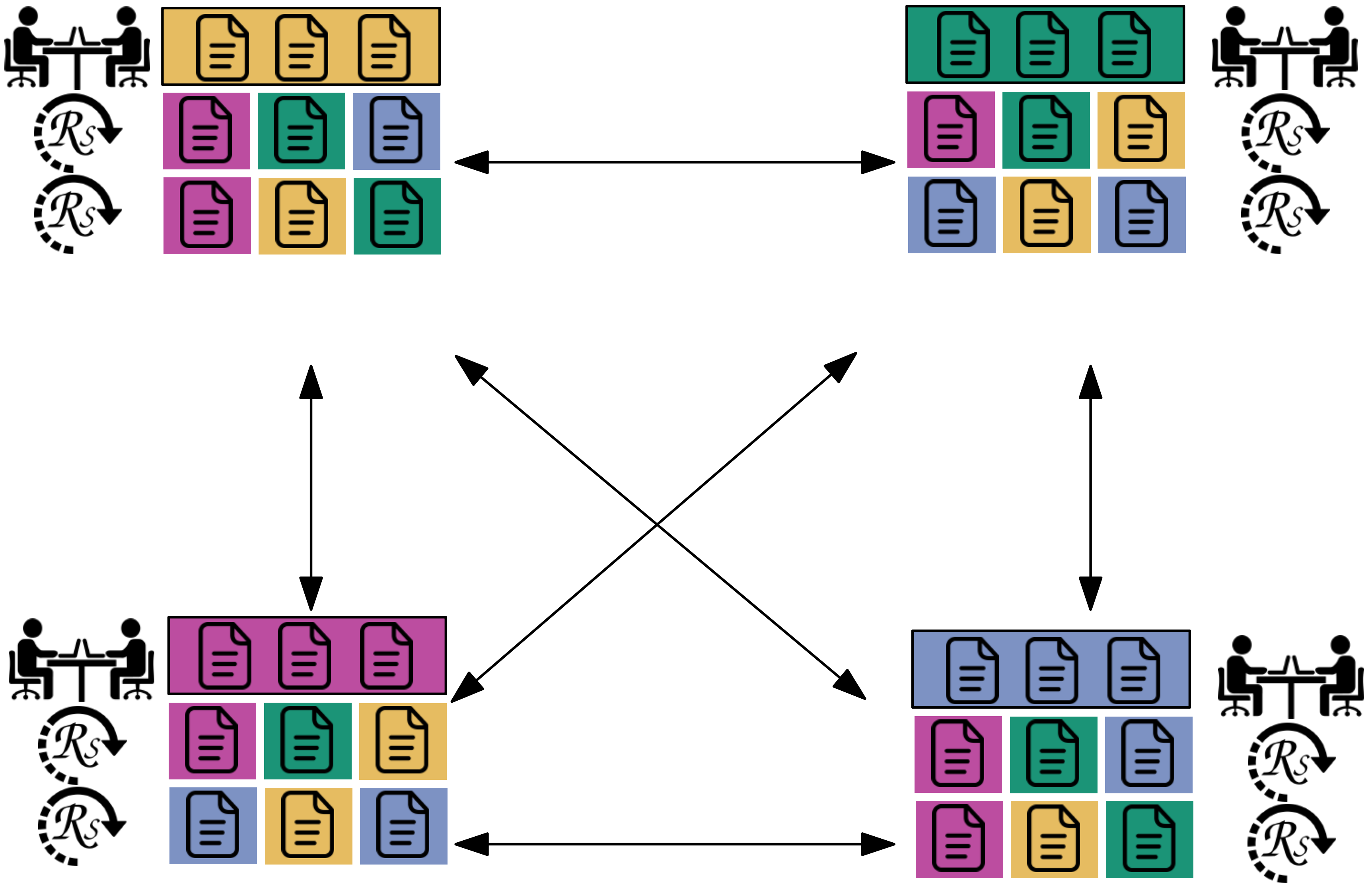


Data Distribution for Faster Recovery

- **break up access pattern by randomly distributing blocks**
  → more PEs serving data
- too many PEs serving data
  → messages too small
- empirical optimum: permute $256\,$KiB together

Constant time and space permutation

- Linear Congruential Generator
- Encrypt block IDs

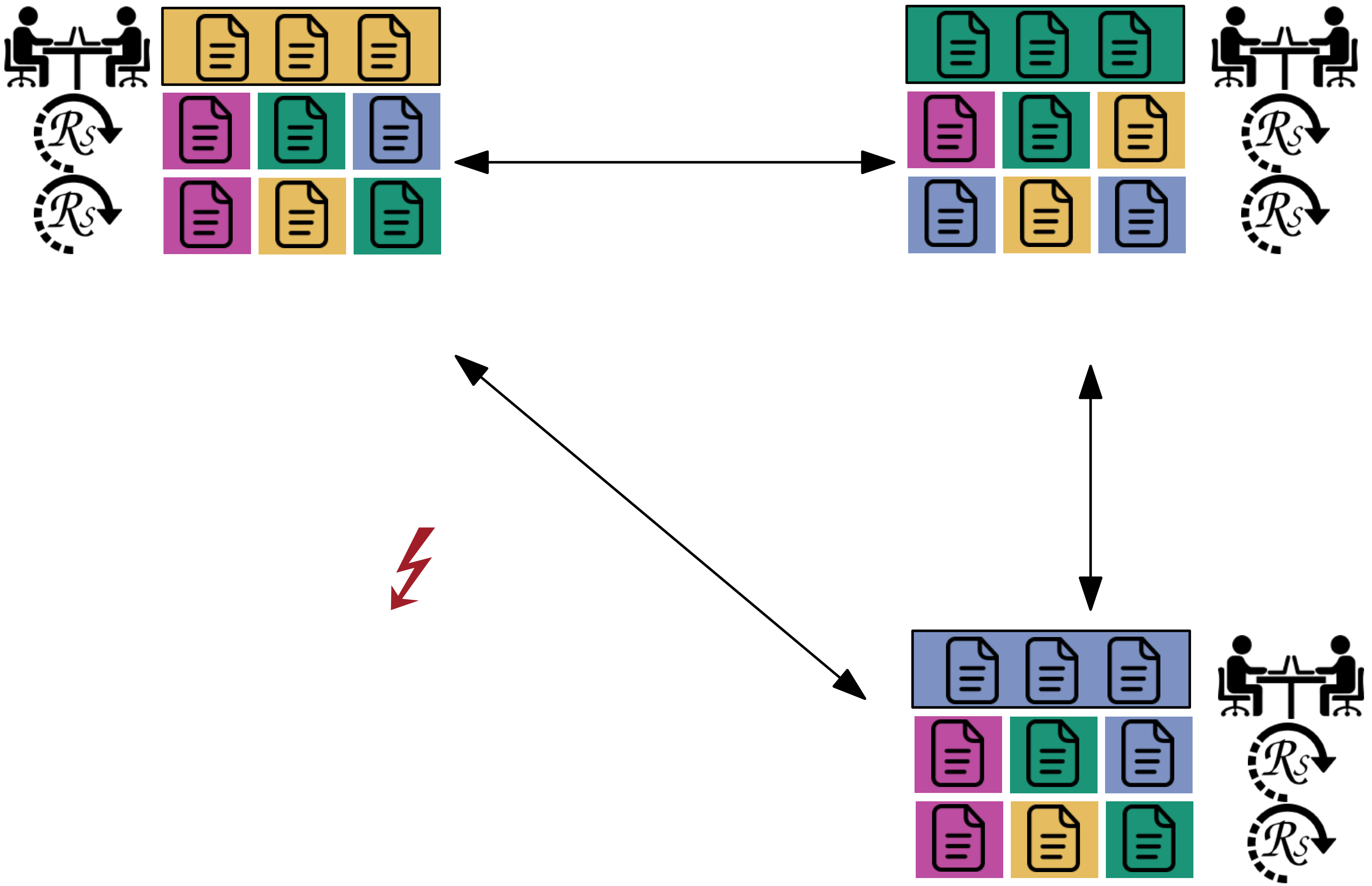


Data Distribution for Faster Recovery

- **break up access pattern by randomly distributing blocks**
  → more PEs serving data
- too many PEs serving data
  → messages too small
- empirical optimum: permute 256 KiB together

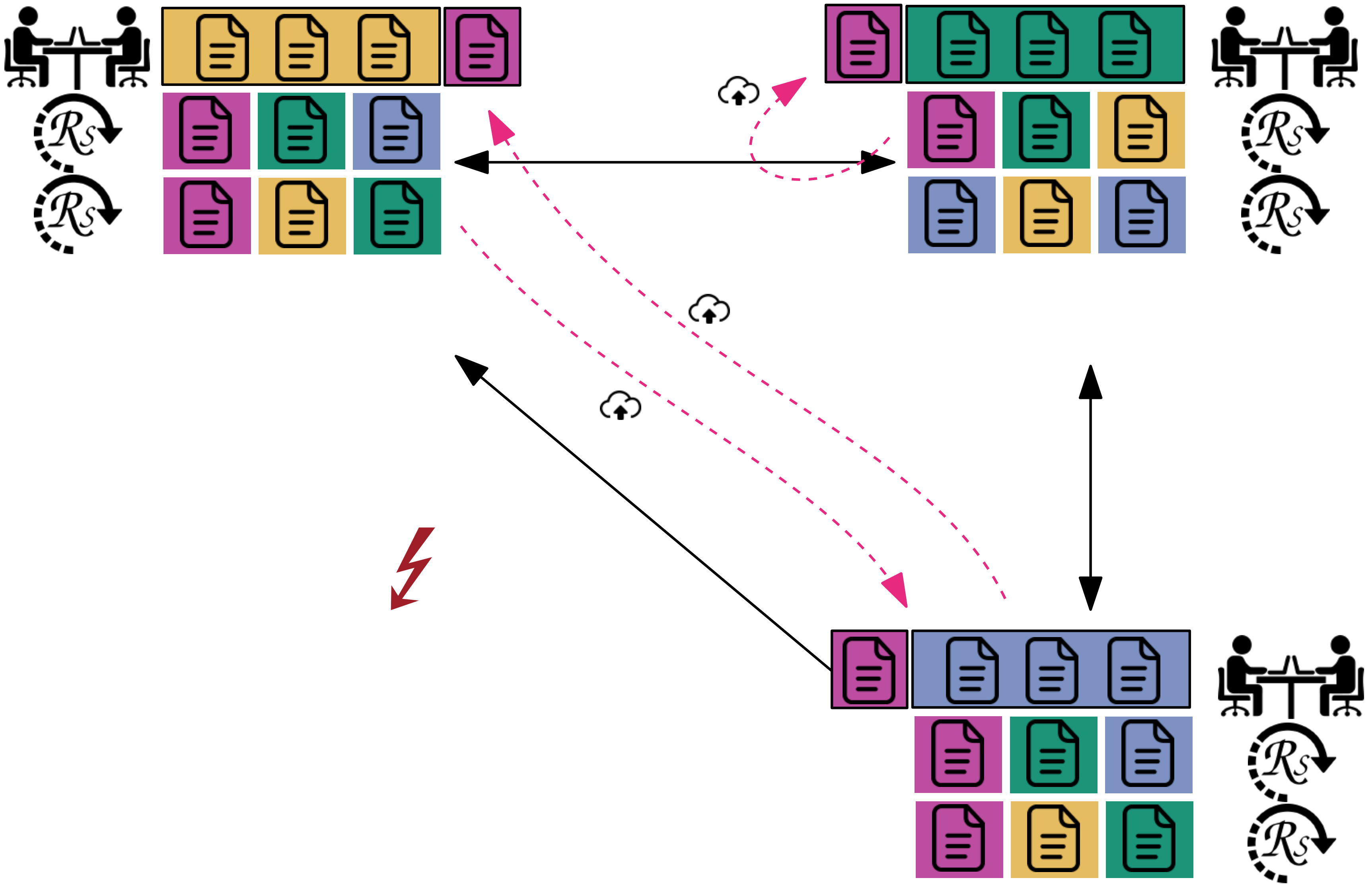
Constant time and space permutation

- Linear Congruential Generator
- Encrypt block IDs



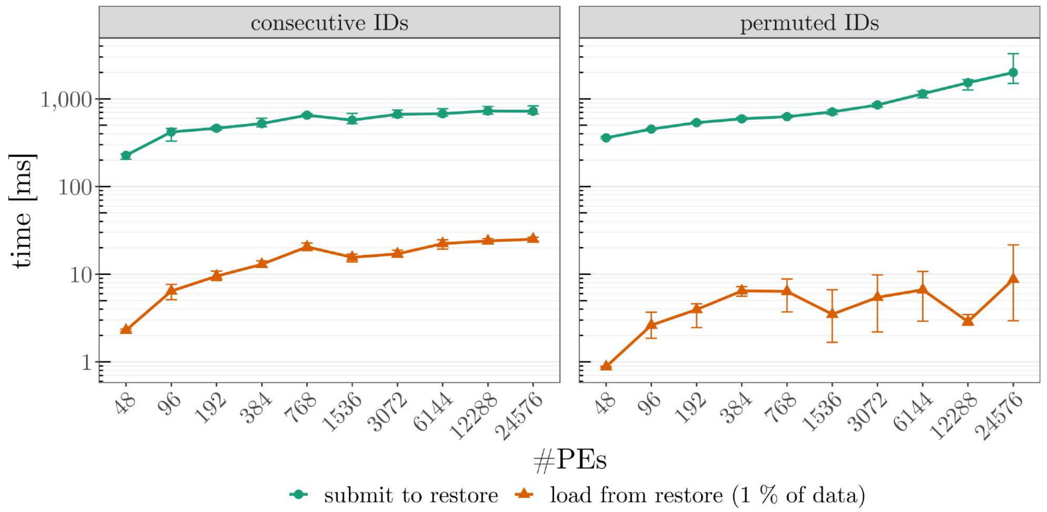
Data Distribution for Faster Recovery

- **break up access pattern by randomly distributing blocks**
  → more PEs serving data
- too many PEs serving data
  → messages too small
- empirical optimum: permute 256 KiB together

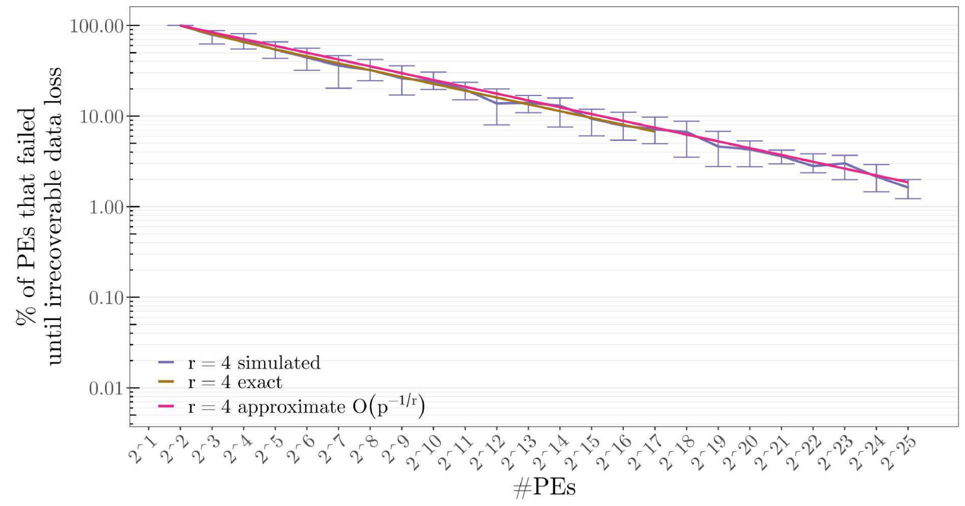


Evaluation

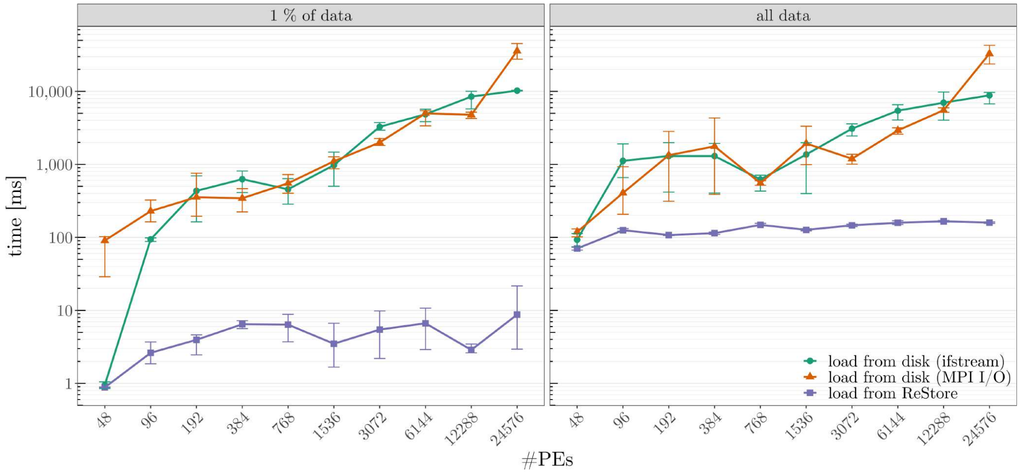
ID-randomization speeds up recovery



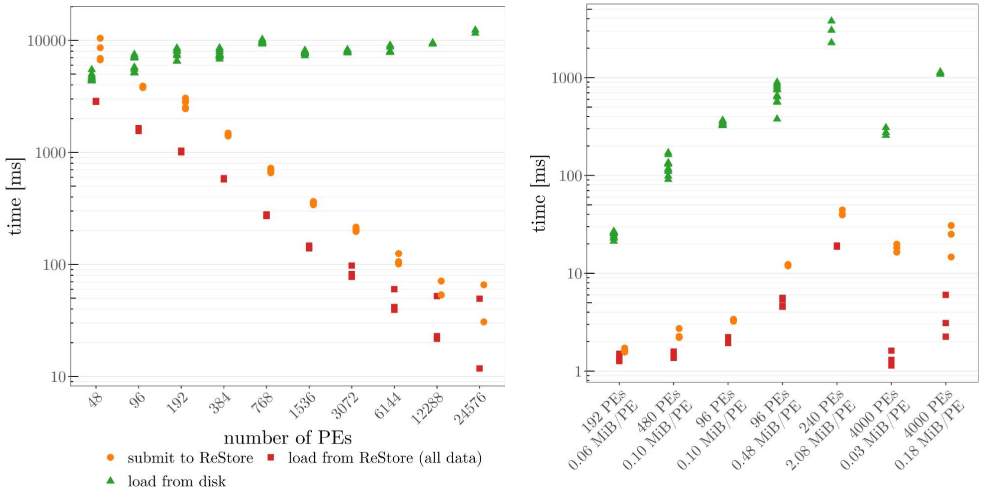
data loss expected after $\mathcal{O}(p^{-1/r})$ failures



substantially faster than disk access

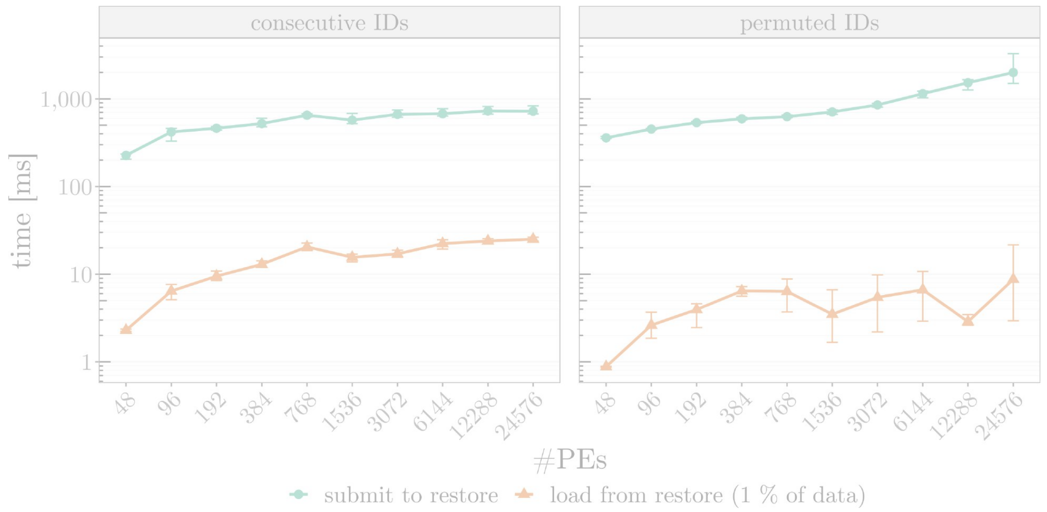


real-world application benchmarks

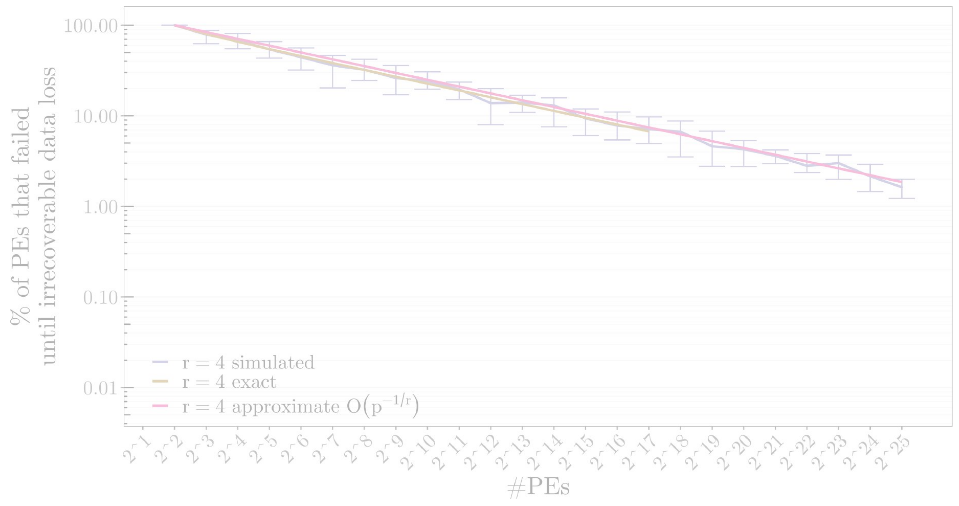


Evaluation

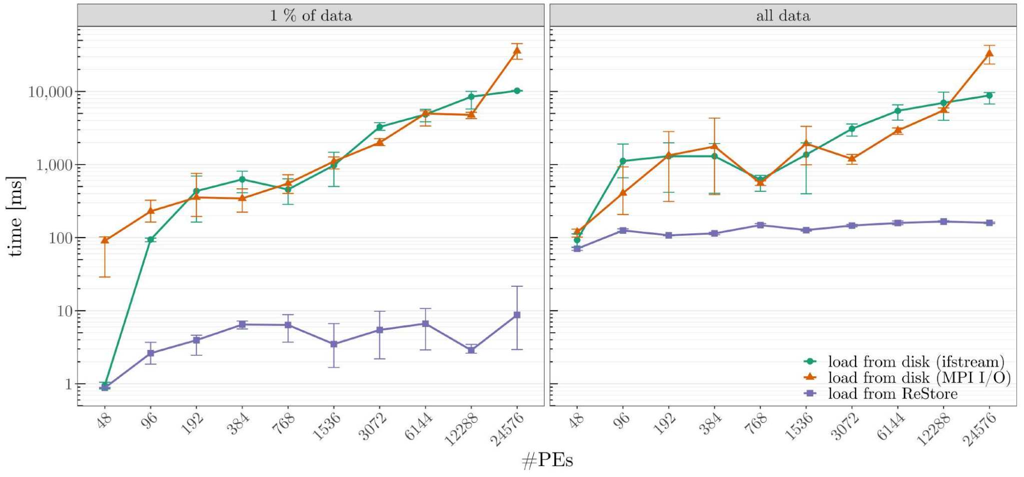
ID-randomization speeds up recovery



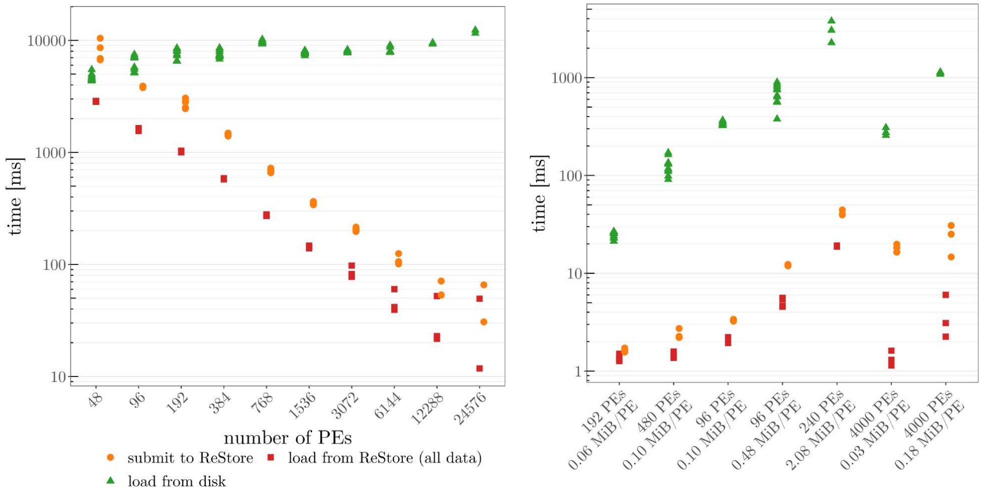
data loss expected after $\mathcal{O}(p^{-1/r})$ failures



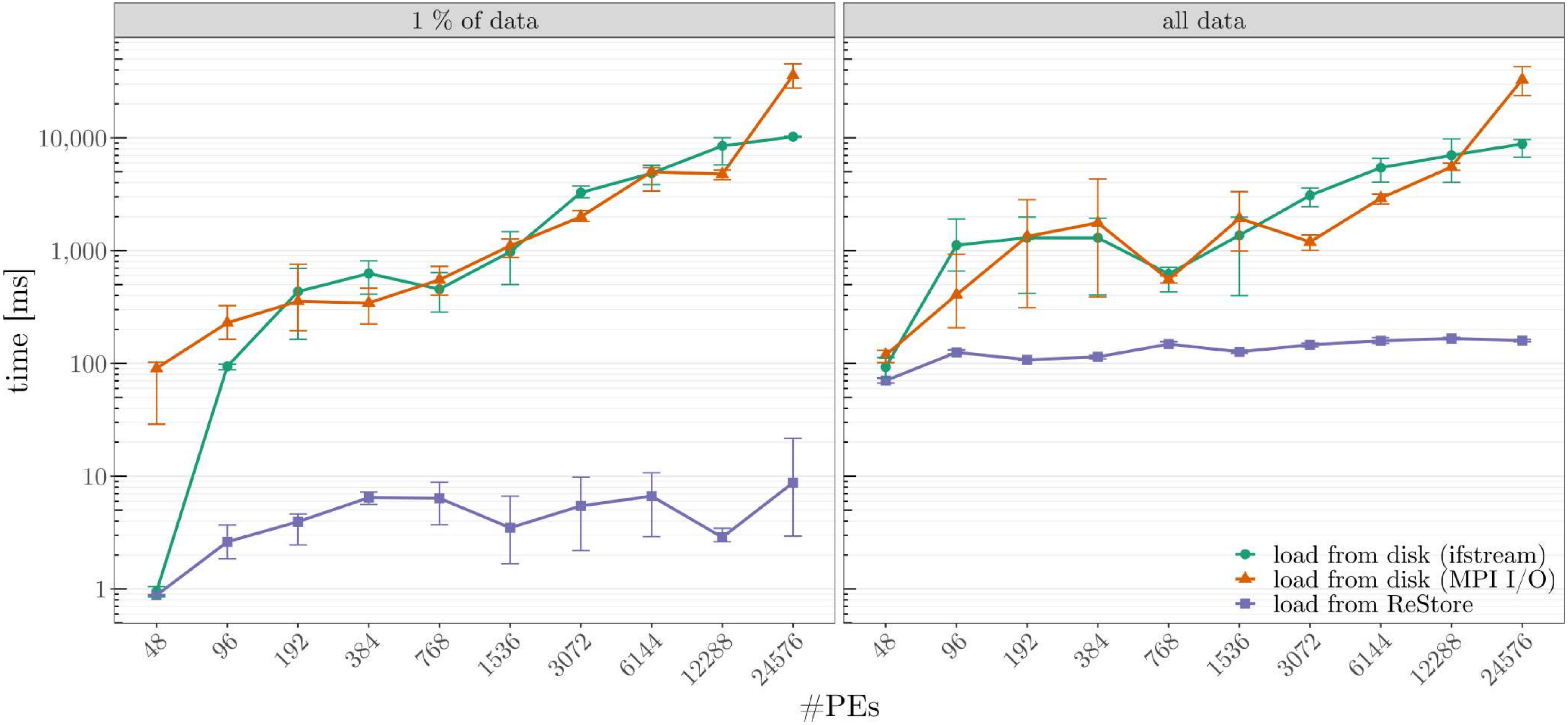
substantially faster than disk access



real-world application benchmarks

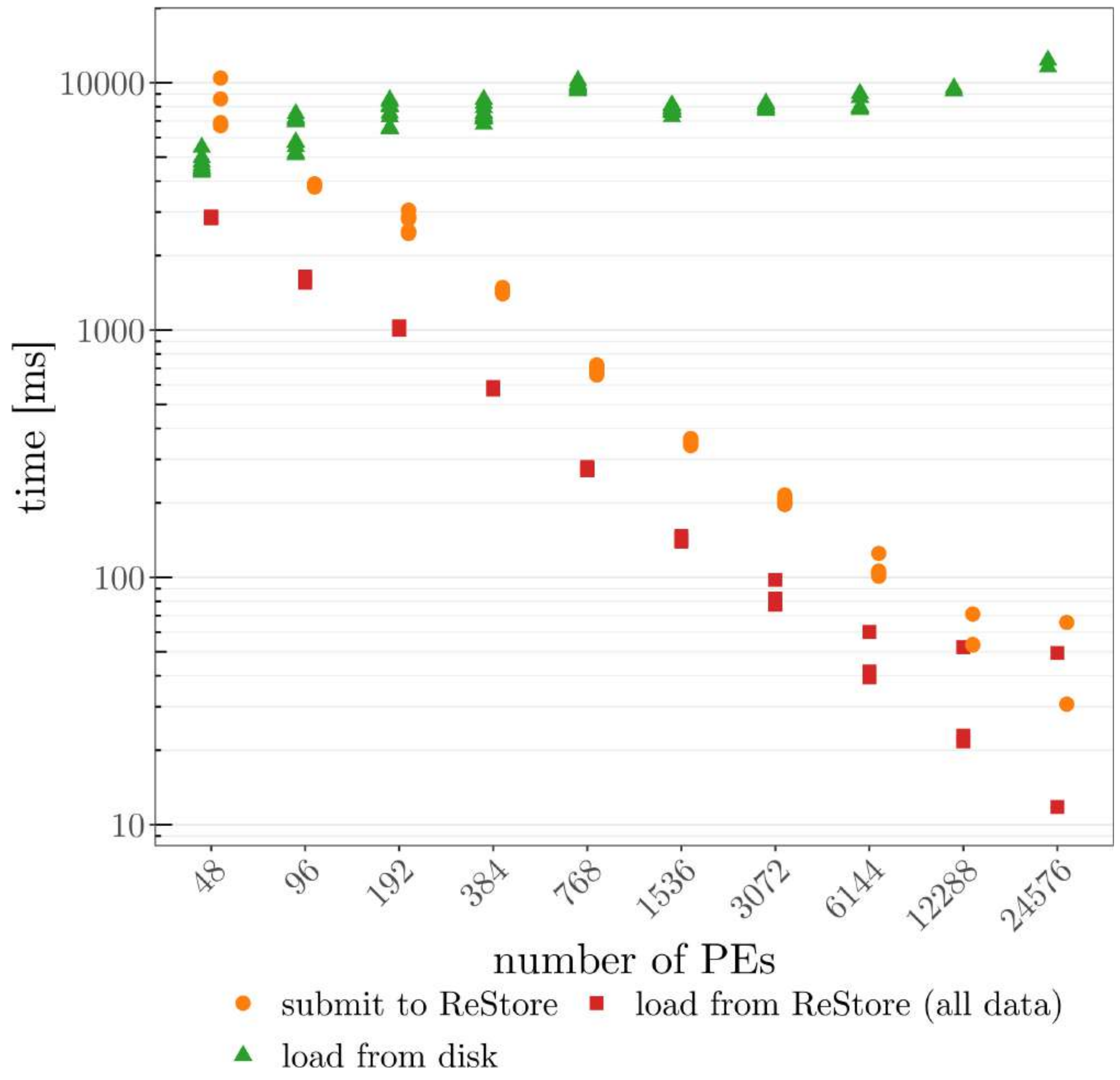


In-Memory vs. Parallel File System



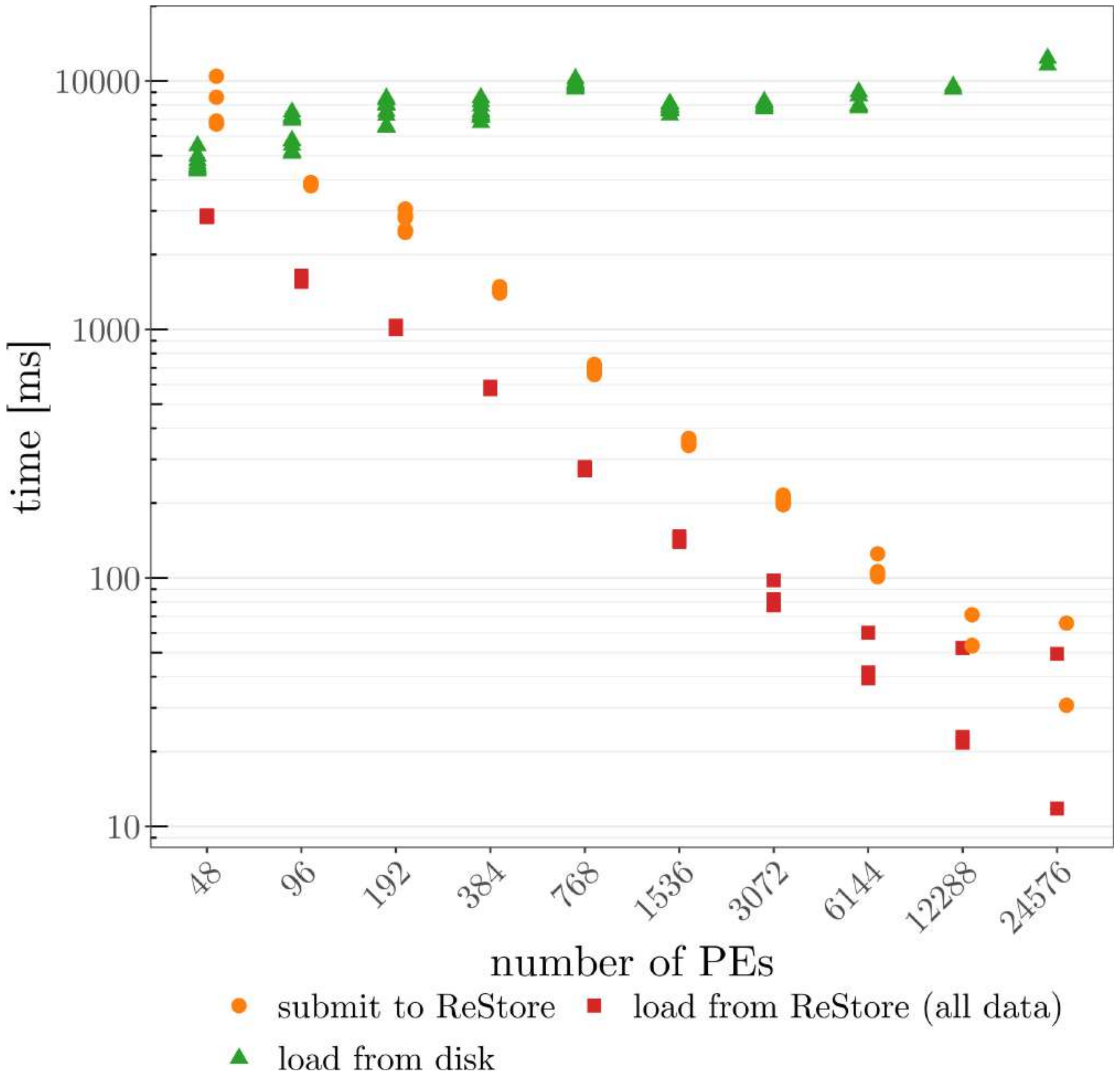
16 MiB data per PE

Overhead of ReStore in RAxML-NG

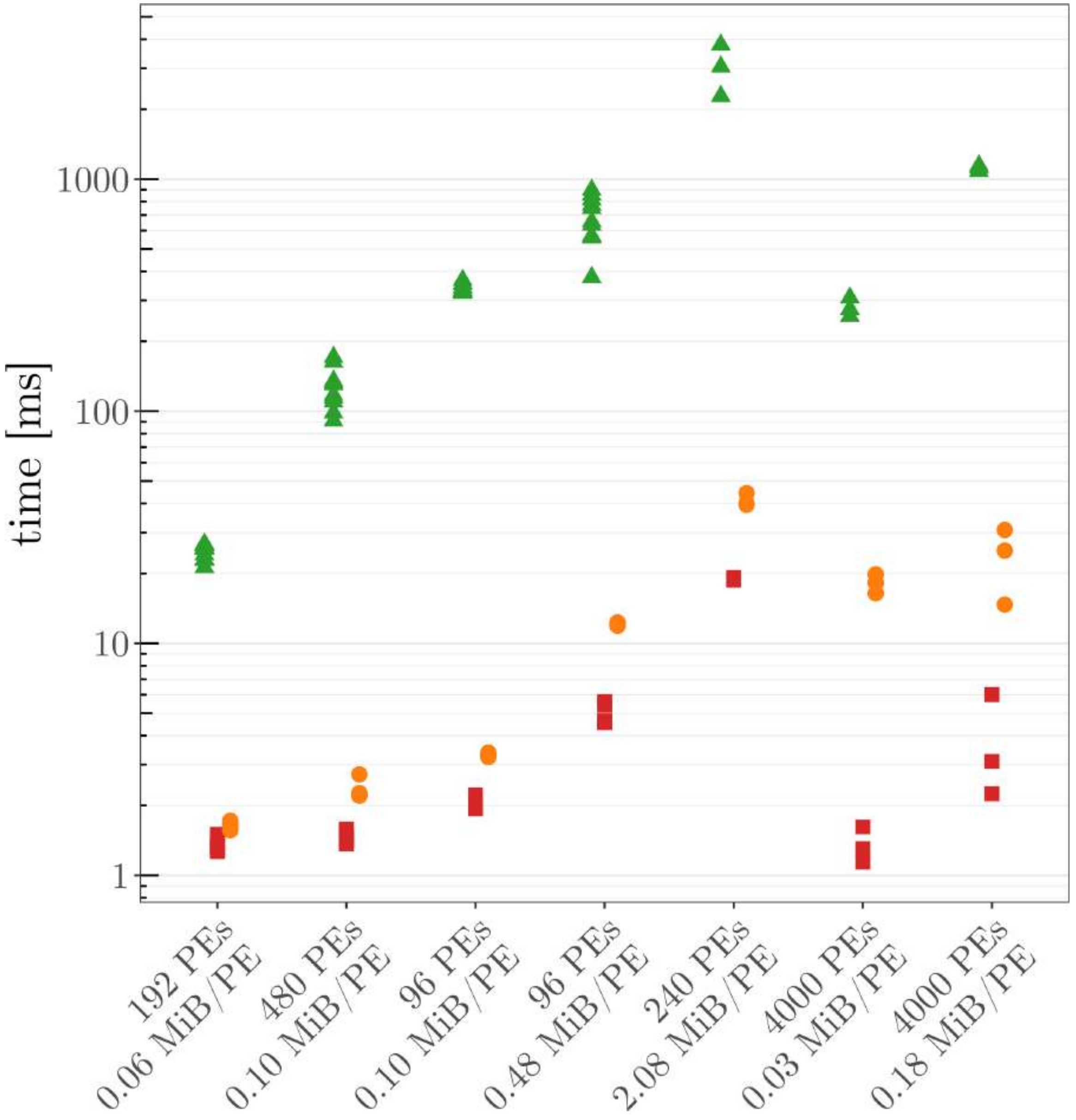


19.1 GiB synthetic dataset

Overhead of ReStore in RAxML-NG



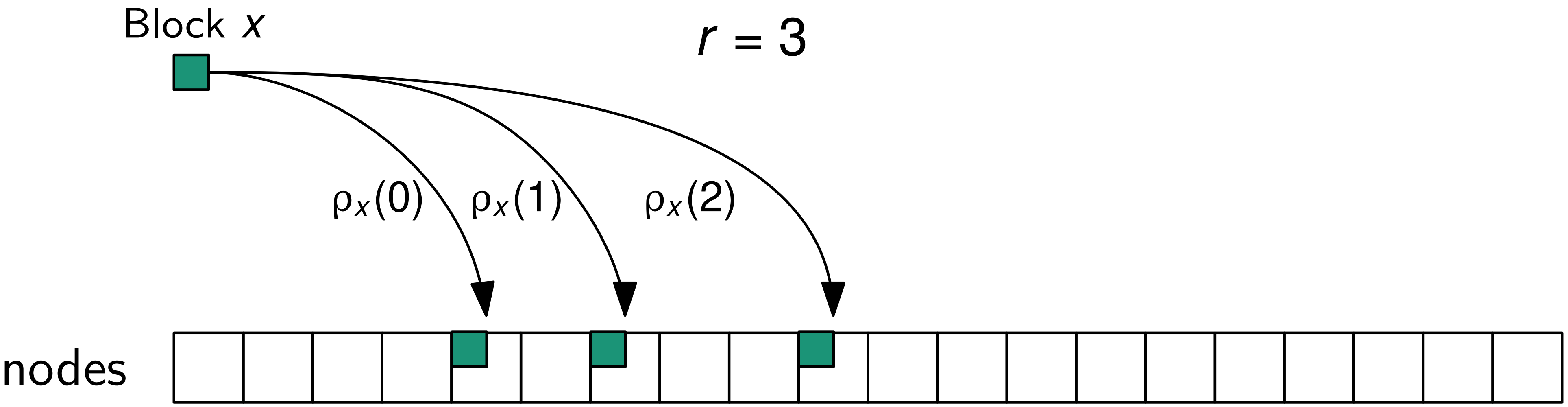
19.1 GiB synthetic dataset



empirical datasets

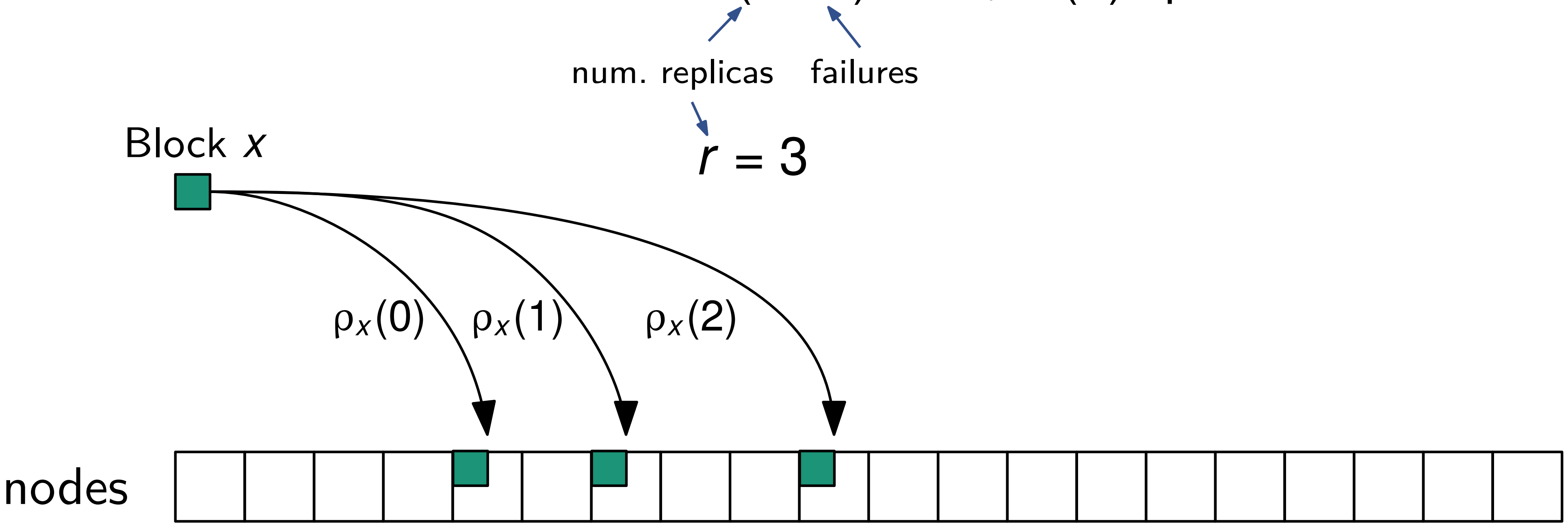
Recovering Replicas After a Node Failure

- **Goal:** Restore lost replicas after a failure; copying only the lost data
- **Idea:** For each block x, draw pseudorandom permutation ρ_x on $[0, p-1]$
- Place copies on $\rho_x(0), \rho_x(1), \ldots$
- Nodes on which this block is stored? $\mathcal{O}(r+f)$ time, $\mathcal{O}(1)$ space



Recovering Replicas After a Node Failure

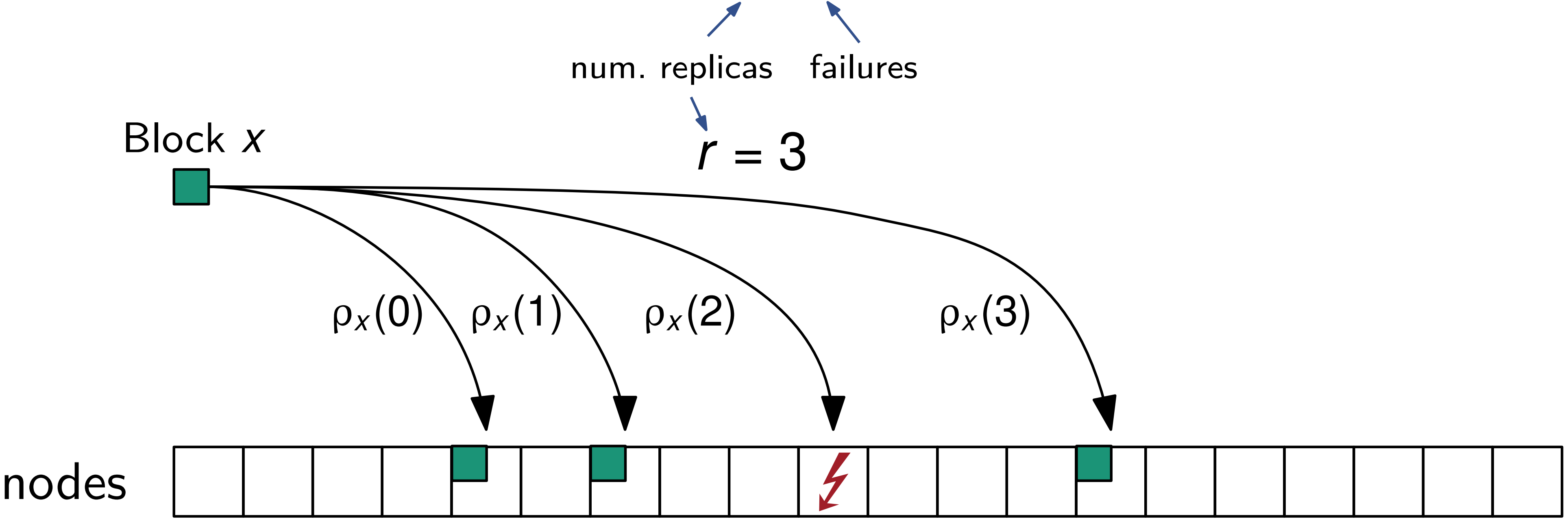
- **Goal:** Restore lost replicas after a failure; copying only the lost data
- **Idea:** For each block x, draw pseudorandom permutation ρ_x on $[0, p-1]$
- Place copies on $\rho_x(0), \rho_x(1), \ldots$
- Nodes on which this block is stored? $\mathcal{O}(r + f)$ time, $\mathcal{O}(1)$ space



Recovering Replicas After a Node Failure

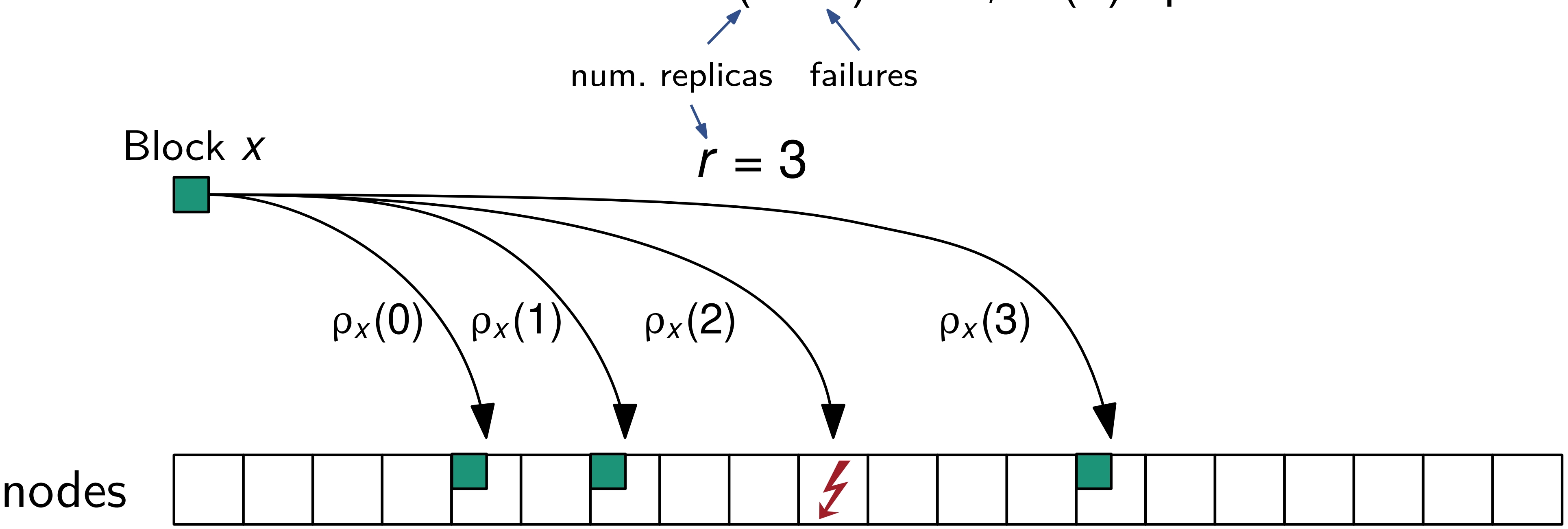
- **Goal:** Restore lost replicas after a failure; copying only the lost data
- **Idea:** For each block x, draw pseudorandom permutation ρ_x on $[0, p-1]$
- Place copies on $\rho_x(0), \rho_x(1), \ldots$
- Nodes on which this block is stored? $\mathcal{O}(r + f)$ time, $\mathcal{O}(1)$ space

num. replicas failures



Recovering Replicas After a Node Failure

- **Goal:** Restore lost replicas after a failure; copying only the lost data
- **Idea:** For each block x, draw pseudorandom permutation ρ_x on $[0, p-1]$
- Place copies on $\rho_x(0), \rho_x(1), \ldots$
- Nodes on which this block is stored? $\mathcal{O}(r + f)$ time, $\mathcal{O}(1)$ space



num. replicas

failures

Block x

$r = 3$

$\rho_x(0)$ $\rho_x(1)$ $\rho_x(2)$ $\rho_x(3)$

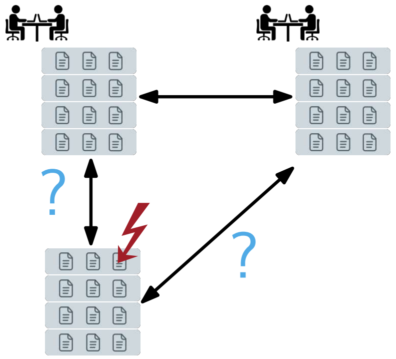
nodes

No need to redistribute any block that did not lose a replica!

Summary Checkpointing & ReStore

Detecting Failures

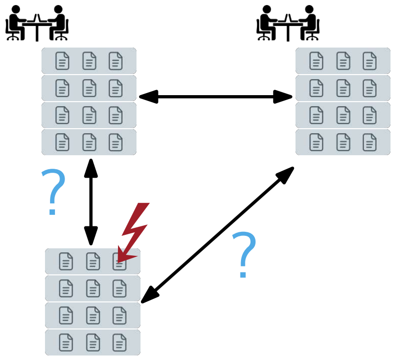
- ULFM in MPI detects **fail-stop** failures via missed heartbeat messages
- MPI will **notify** you of the failure and **repair the communicator**
- You have to repair your application yourself



Summary Checkpointing & ReStore

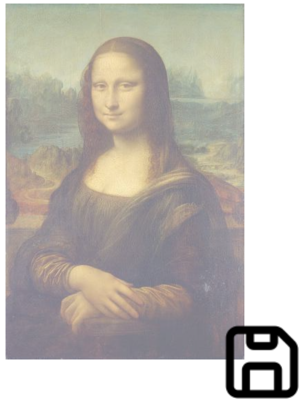
Detecting Failures

- ULFM in MPI detects **fail-stop** failures via missed heartbeat messages
- MPI will **notify** you of the failure and **repair the communicator**
- You have to repair your application yourself



Checkpointing

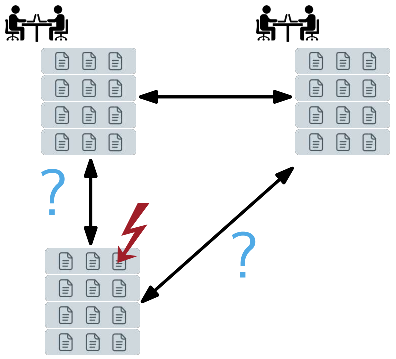
- Repeatedly create backup of the **dynamic** program's state
- Upon failure, roll back to last backup, redistribute work & **static** data
- there is an optimal checkpointing frequency
- recoveries should be **faster** with **more** CPUs



Summary Checkpointing & ReStore

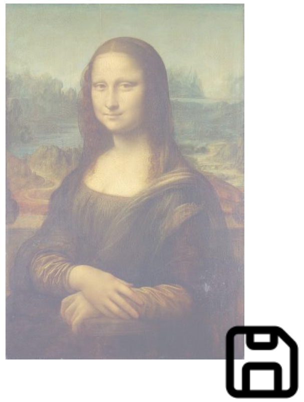
Detecting Failures

- ULFM in MPI detects **fail-stop** failures via missed heartbeat messages
- MPI will **notify** you of the failure and **repair the communicator**
- You have to repair your application yourself



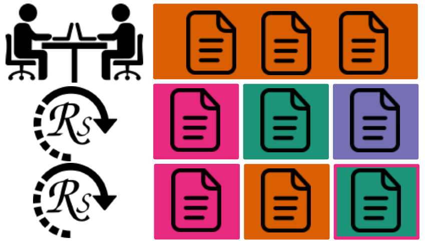
Checkpointing

- Repeatedly create backup of the **dynamic** program's state
- Upon failure, roll back to last backup, redistribute work & **static** data
- there is an optimal checkpointing frequency
- recoveries should be **faster** with **more** CPUs



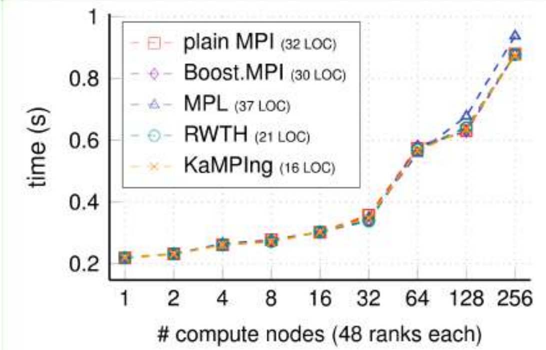
ReStore provides

- **scalable** recovery
- from an **in-memory** storage
- requiring **no extra nodes**
- with adjustable **replication level**



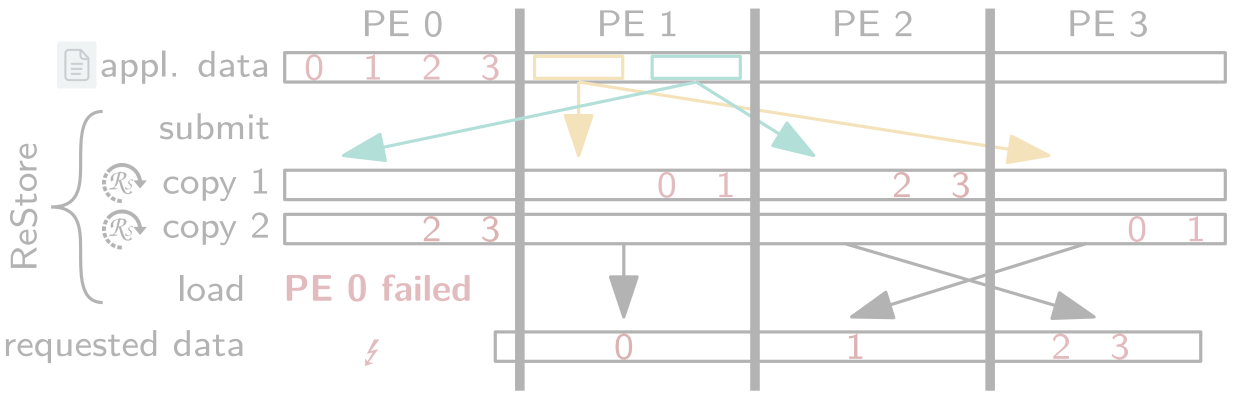
Overview

zero-overhead C++ MPI wrapper and distributed toolbox [SC24]

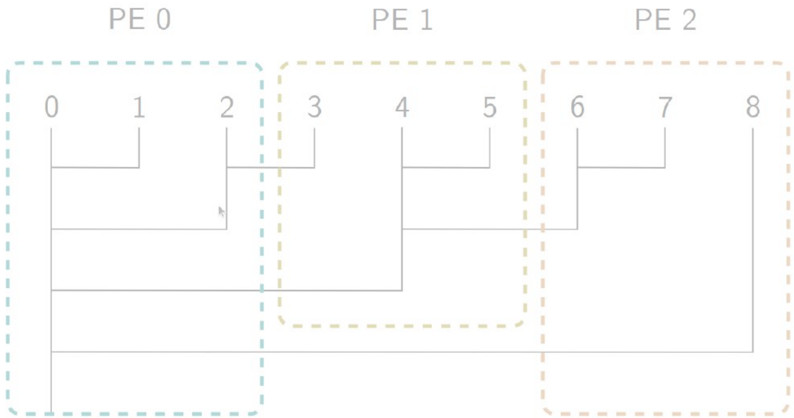


```
recv_buf = comm.allgatherv(send_buf(v_local));
```

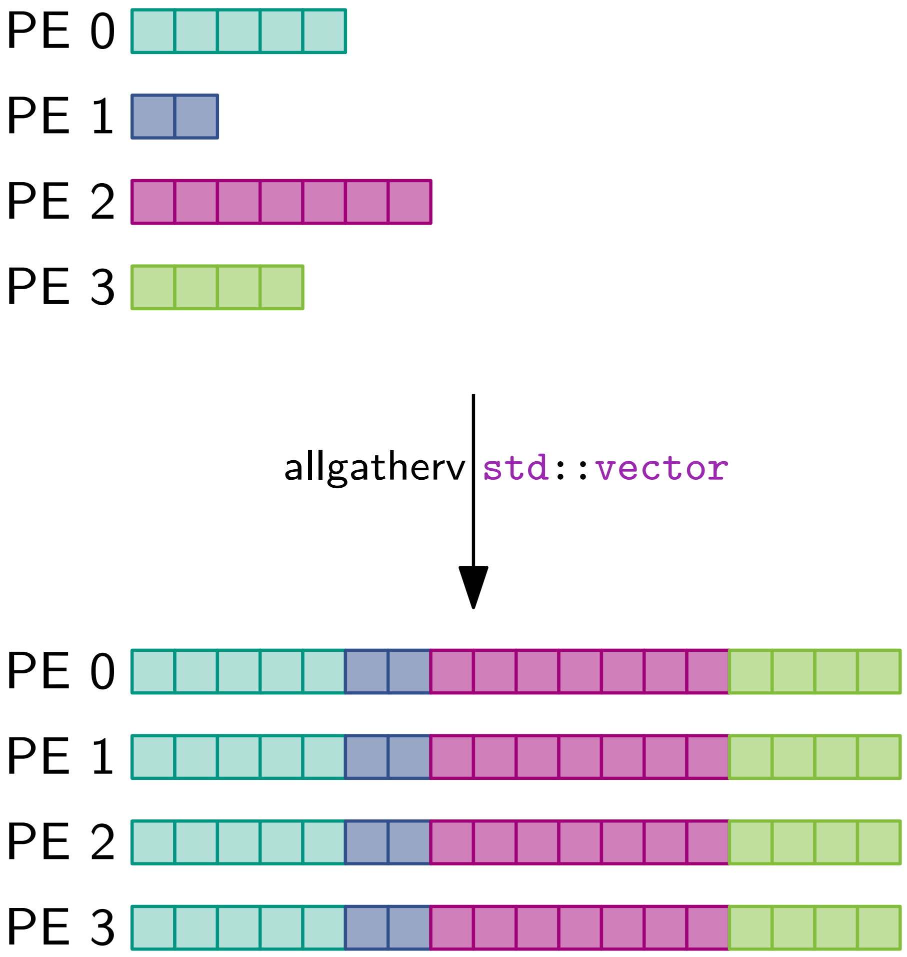
replicated storage for rapid recovery after CPU failure [FTXS22]



reproducible distributed memory reduction

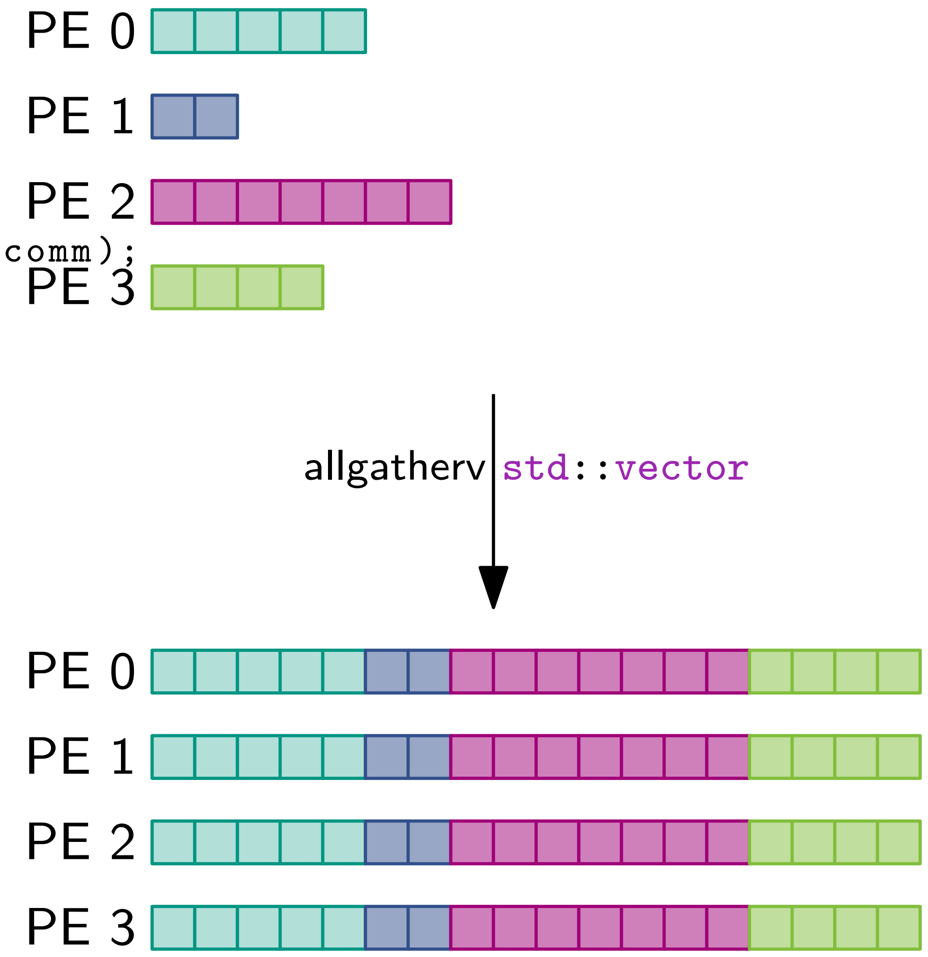


Using MPI from C++



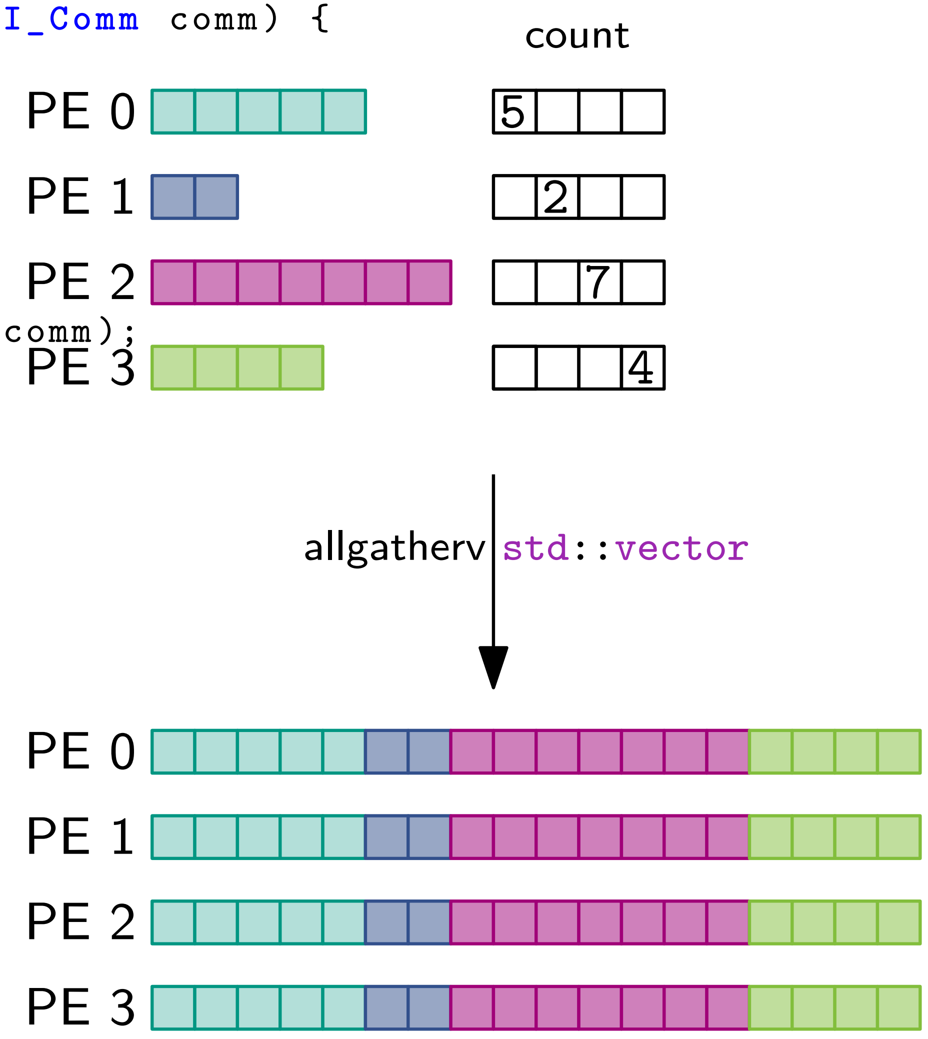
Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[0] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```



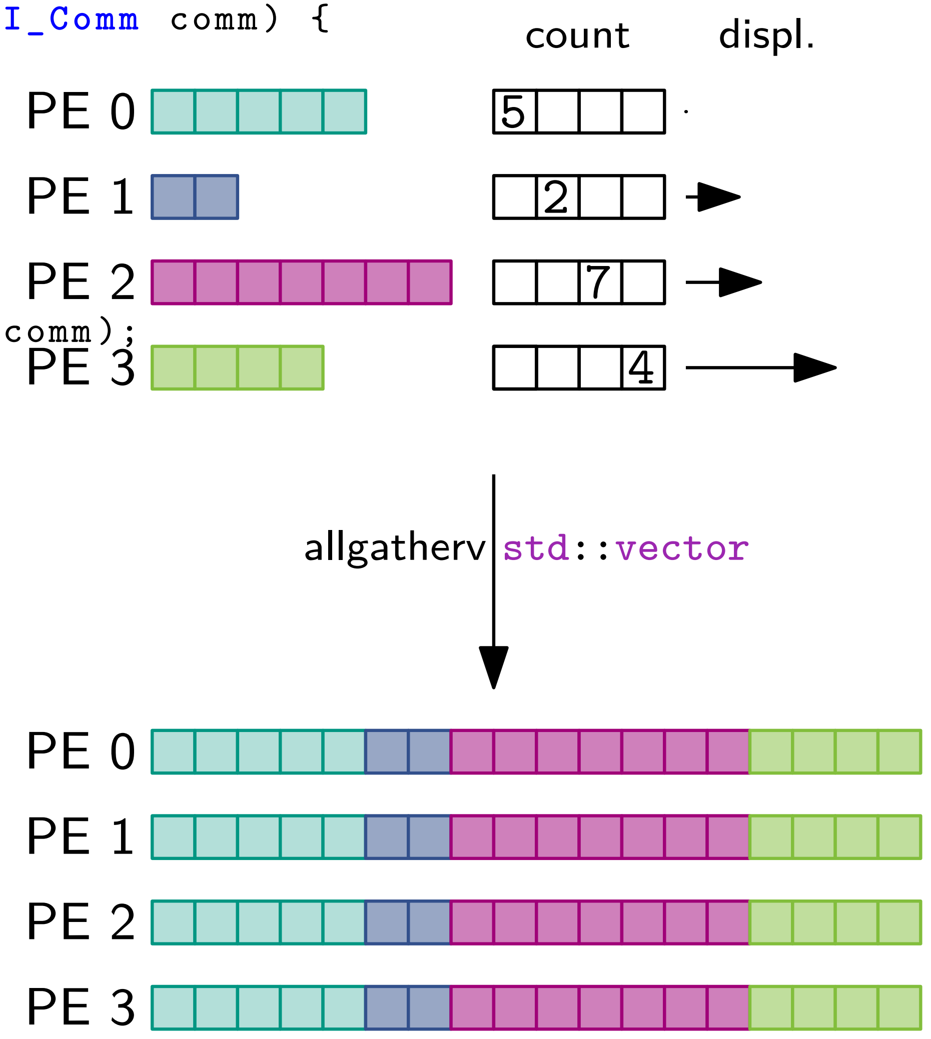
Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[0] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```



Using MPI from C++

```cpp
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[0] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```



Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[0] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  int size;
  int rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  std::vector<int> rc(size), rd(size);
  rc[0] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

C-style API

all other parameters can be inferred

parameter order?

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {
  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1, MPI_INT, comm);
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

all other parameters can be inferred

parameter order?

Goals of KaMPIng:

Karlsruhe **MPI** **n**ext **g**eneration

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {


  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

all other parameters can be inferred

parameter order?

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
std::vector<double> get_whole_vector(std::vector<double> const& v_local, MPI_Comm comm) {

  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<double> v_global(rd.back() + rc.back());
  MPI_Allgatherv(v_local.data(), v_local.size(), MPI_DOUBLE,
                 v_global.data(), rc.data(), rd.data(),
                 MPI_DOUBLE, comm);
  return v_global;
}
```

all other parameters can be inferred

parameter order?

generalization?

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global(rd.back() + rc.back());
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

all other parameters can be inferred

arbitrary parameter order!

Goals of KaMPIng:

Karlsruhe MPI next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global(rd.back() + rc.back());
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

manual allocation

Goals of KaMPIng:

Karlsruhe **MPI** next generation

- ☐ zero-overhead **abstraction** over MPI
- ☐ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- ☐ flexible **parameter handling**, sensible defaults
- ☐ configurable **memory management**
- ☐ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {


  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

common idiom!

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {




  std::vector<int> rc(comm.size()), rd(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));
  std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc), recv_displs(rd));

  return v_global;
}
```

common idiom!

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {


  std::vector<int> rc(comm.size());
  rc[0] = v_local.size();
  comm.allgather(send_recv_buf(rc), send_count(1));

  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                  recv_counts(rc));

  return v_global;
}
```

common idiom!

automatic or manual allocation

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {



  std::vector<T> v_global;
  comm.allgatherv(send_buf(v_local), recv_buf(v_global));


  return v_global;
}
```

return by reference

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {

  return comm.allgatherv(send_buf(v_local));

}
```

return by reference

or by value

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {
  return comm.allgatherv(send_buf(v_local));
}
```

Goals of KaMPIng:

Karlsruhe MPI next generation

- □ zero-overhead **abstraction** over MPI
- □ covering whole abstraction **range**:
  rapid prototyping ↔ highly engineered algorithms
- □ flexible **parameter handling**, sensible defaults
- □ configurable **memory management**
- □ compatible with **move semantics**

Using MPI from C++

```
template<typename T>
std::vector<T> get_whole_vector(std::vector<T> const& v_local, Communicator const& comm) {
  return comm.allgatherv(send_buf(v_local));
}
```

```
// avoid implicit allocation
comm.allgatherv(send_buf(v_local),
                recv_counts_out<no_resize>(some_buf));

// pass buffer ownership to calls
rc = comm.allgatherv(send_buf(v_local), recv_buf(v_global),
                     recv_counts_out<resize_to_fit>(std::move(rc)));

// retrieve auxiliary data
auto [recvbuf, counts] = comm.allgatherv(send_buf(v_local),
                                         recv_counts_out());
```

Goals of KaMPIng:

Karlsruhe MPI next generation

□ zero-overhead **abstraction** over MPI

□ covering whole abstraction **range**: rapid prototyping ↔ highly engineered algorithms

□ flexible **parameter handling**, sensible defaults

□ configurable **memory management**

□ compatible with **move semantics**

Memory Management

Who manages memory?

- Avoid memory leaks
- Re-use allocated memory
- Usability
- Performance overhead

Memory Management

Who manages memory?

- Avoid memory leaks
- Re-use allocated memory
- Usability
- Performance overhead

```
// Library allocates receive buffer
auto recv_buf = comm.allgatherv(send_buf(v_local));

// Re-use existing buffer
// std::vector<T> recv_buf // allocated somewhere
comm.allgatherv(send_buf(v_local),
                recv_counts<no_resize>(recv_buf));

// Pass buffer ownership to calls
recv_buf = comm.allgatherv(send_buf(v_local),
                           recv_buf<resize_to_fit>(std::move(recv_buf)));

// Reference counting

// Let the user manage memory
```

Memory Management

Who manages memory?

- Avoid memory leaks
- Re-use allocated memory
- Usability
- Performance overhead

```
// Library allocates receive buffer
auto recv_buf = comm.allgatherv(send_buf(v_local));

// Re-use existing buffer
// std::vector<T> recv_buf // allocated somewhere
comm.allgatherv(send_buf(v_local),
                recv_counts<no_resize>(recv_buf));

// Pass buffer ownership to calls
recv_buf = comm.allgatherv(send_buf(v_local),
                     recv_buf<resize_to_fit>(std::move(recv_buf)));

// Reference counting

// Let the user manage memory
```

my post-implementation opinion

+ no memory leaks
+ memory re-usable

🟨 comfortable to use but uncommon

– tricky and complex implemenation

Named Parameters

Fewer parameters

- Auto-infer where possible
- Sane defaults
- Arbitrary order
- No runtime overhead
- Type safe and generalizable

```
auto [recvbuf, displ] = comm.gatherv(
        send_buf(v_local), // send buffer given
        // send count automatically computed
        // send type inferred
        // receive counts automatically computed
        recv_displ_out(), // receive displacements computed and returned
        // receive type inferred
        // default root: 0
);
```

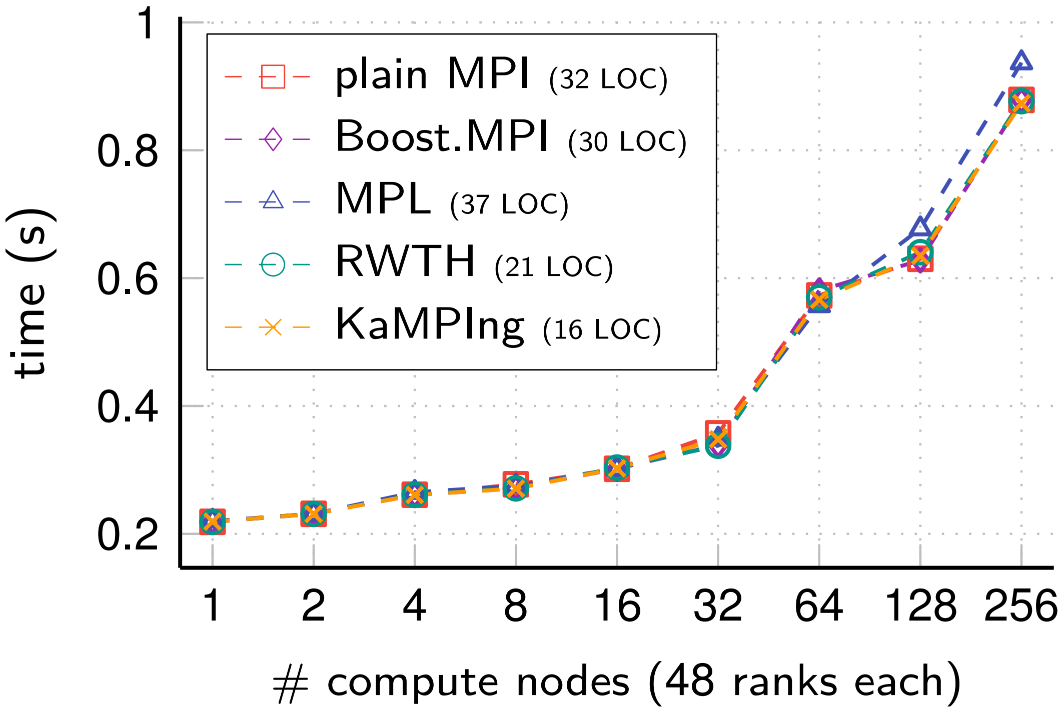
my post-implementation opinion

+ so much easier than plain MPI
+ no runtime overhead
+ type safe and general

▪ implementation mangeable

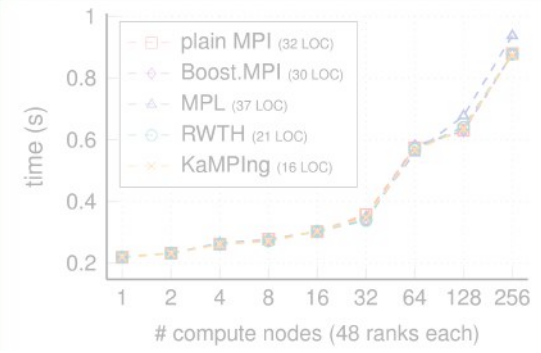
High-Level Features

- Fork of RAxML-NG (widely used phylogenetic inference tool) using KaMPIng
- A plugin system with hooks enables:
  - Abstractions of MPI's upcoming **fault-tolerance** features
  - Integration of **reproducible reduce** with custom reduction operations
  - Automatic serialization
  - . . .



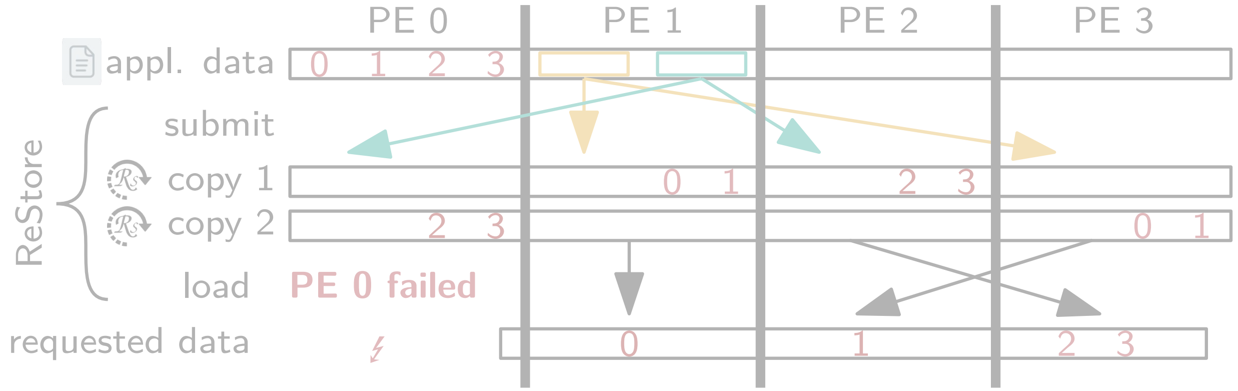
Overview

zero-overhead C++ MPI wrapper and distributed toolbox [SC24]

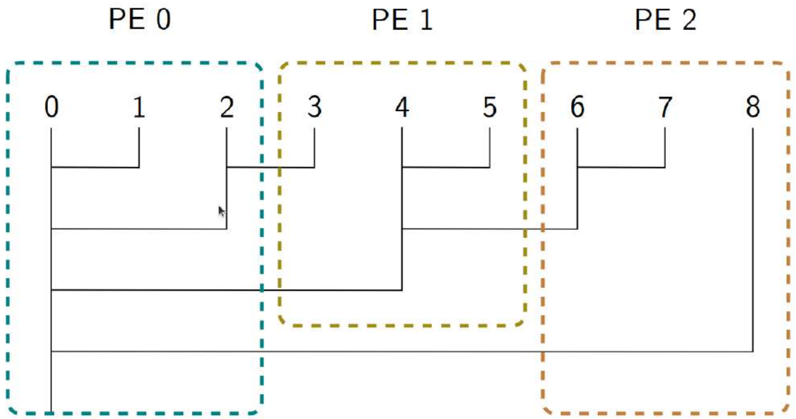


```
recv_buf = comm.allgatherv(send_buf(v_local));
```

replicated storage for rapid recovery after CPU failure [FTXS22]



reproducible distributed memory reduction



Computational Reproducibility

Science is not just **reporting** results, but also **convincing** other that they are correct. → **Reproducible experiments** help with that

Computational Reproducibility

Science is not just **reporting** results, but also **convincing** other that they are correct. → **Reproducible experiments** help with that

bit-wise reproducibility
Running the experiment twice should result in **bit-identical** results

Computational Reproducibility

Science is not just **reporting** results, but
also **convincing** other that they are correct.

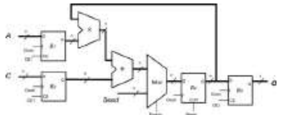
→

Reproducible experiments
help with that

bit-wise reproducibility
Running the experiment twice
should result in **bit-identical** results

Common approaches

- Document compiler, linker, OS, library . . . versions
- Document hardware
- Fix random seed
- Archive data and source code (with DOIs)
- Document procedure, automate as far as possible



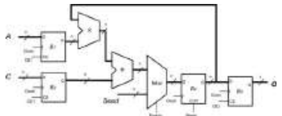
Computational Reproducibility

Science is not just **reporting** results, but also **convincing** other that they are correct. → **Reproducible experiments** help with that

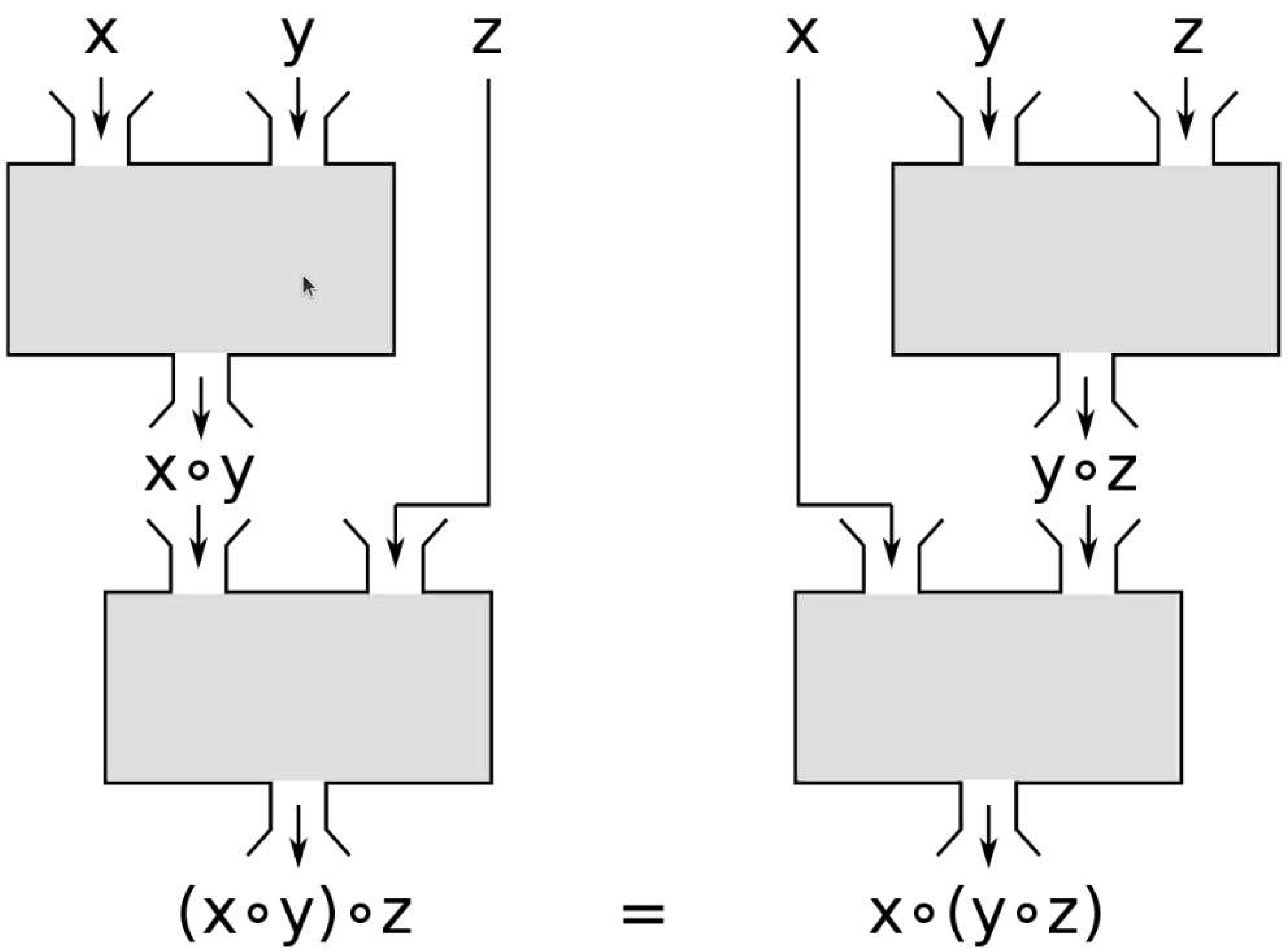
bit-wise reproducibility
Running the experiment twice should result in **bit-identical** results

Common approaches

- Document compiler, linker, OS, library . . . versions
- Document hardware archiving not trivial
- Fix random seed
- Archive data and source code (with DOIs)
- Document procedure, automate as far as possible



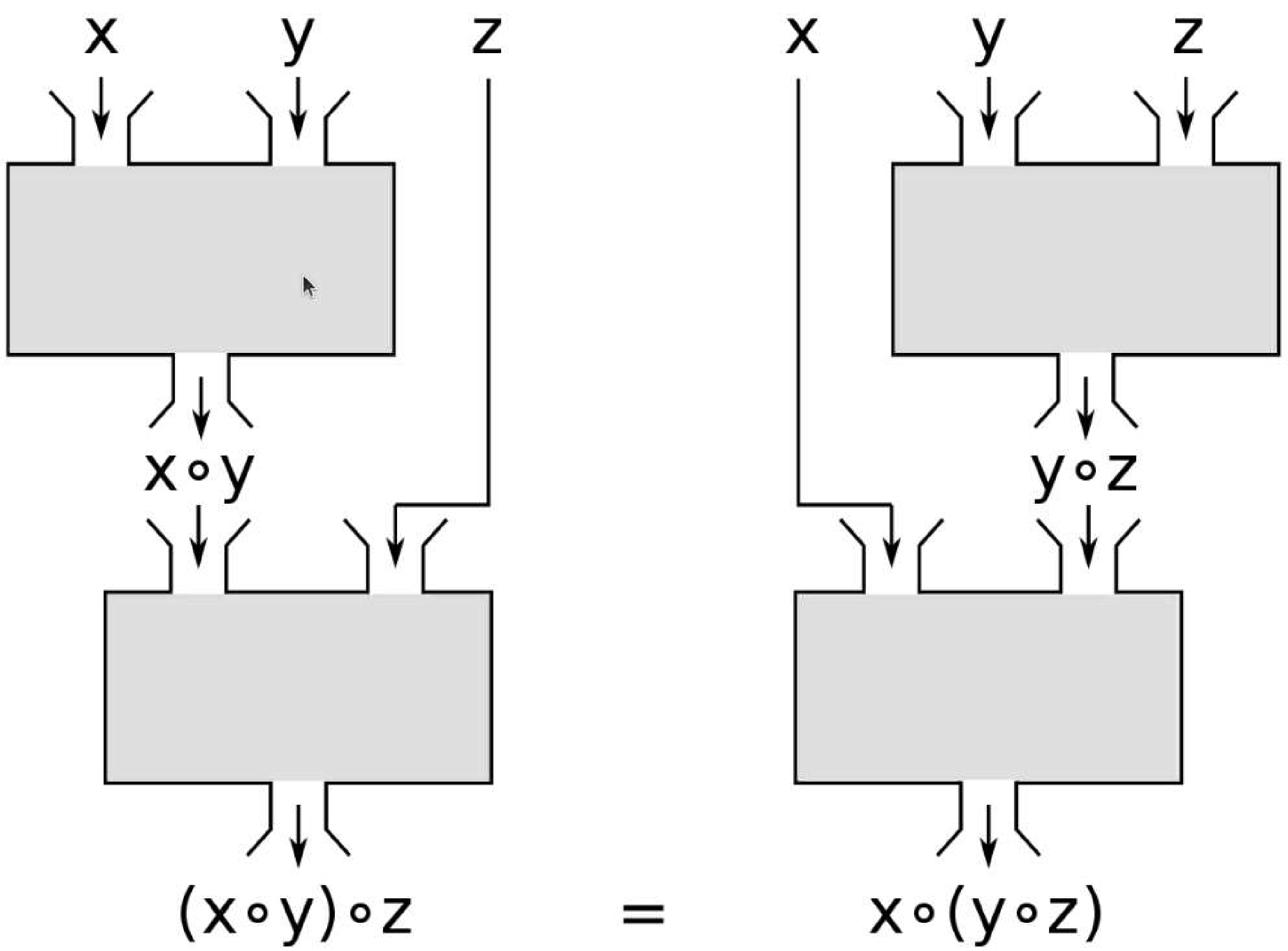
Floating-Point Math is Non-Associative



non-associativity

- $(a + b) + c \neq a + (b + c)$
- different round-off errors

Floating-Point Math is Non-Associative



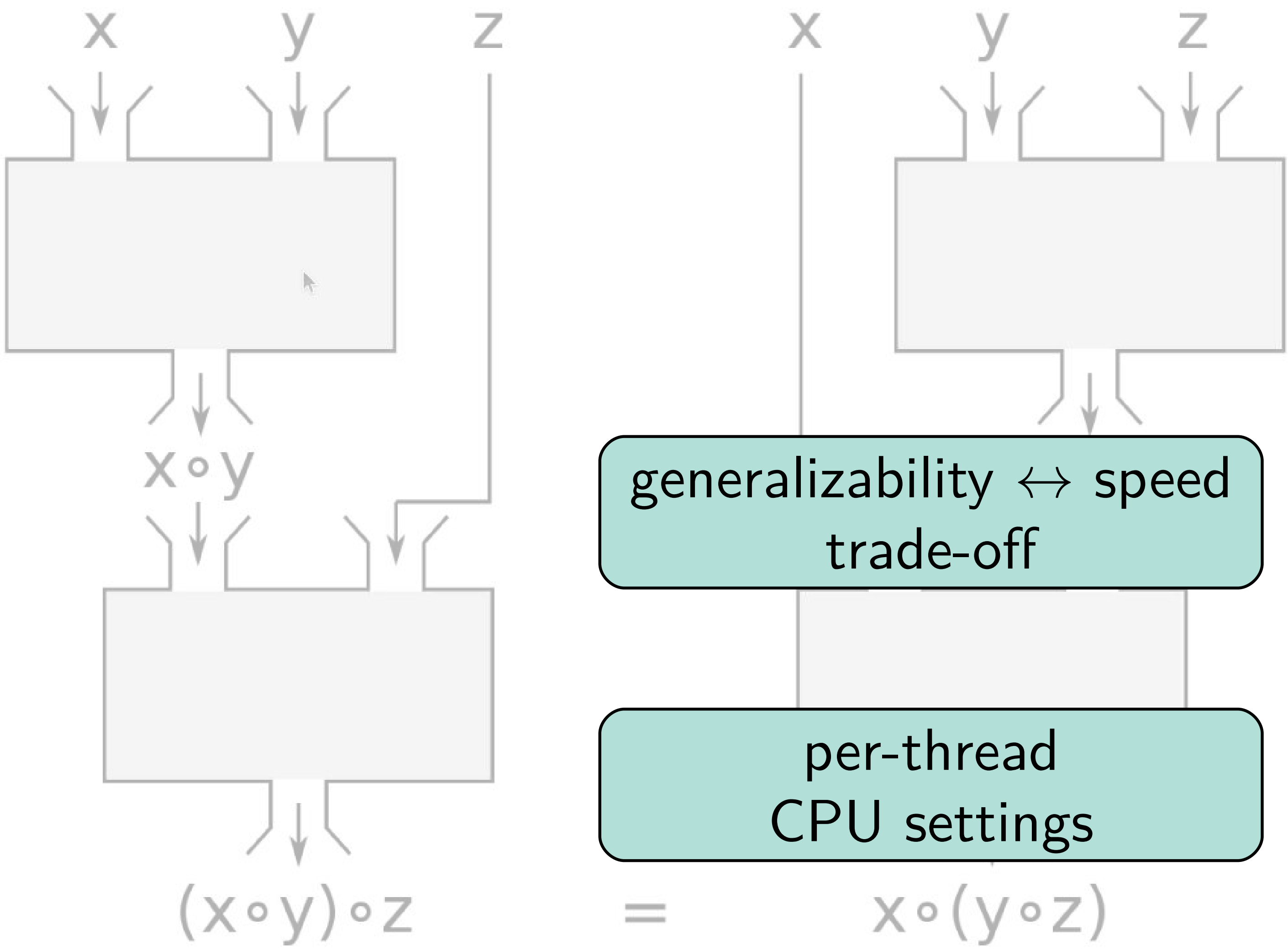
non-associativity

- $(a + b) + c \neq a + (b + c)$
- different round-off errors

when does this happen?

- different SIMD register widths (horizontal add)
- fused multiply-and-add available?
- different rounding mode
- different x87 register precision
- denormalization
- number of CPUs

Floating-Point Math is Non-Associative



generalizability ↔ speed trade-off

per-thread CPU settings

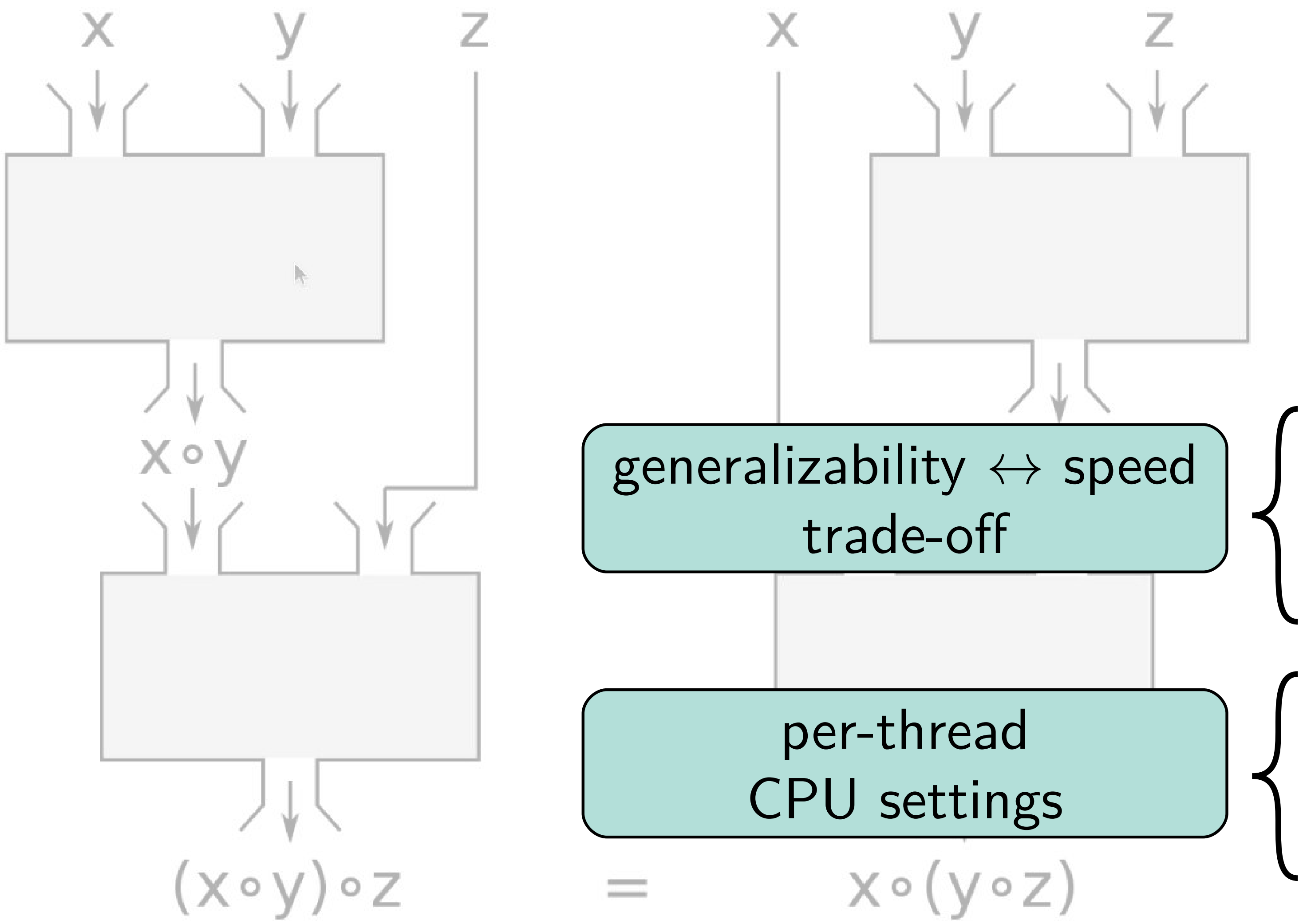
non-associativity

- $(a+b)+c \neq a+(b+c)$
- different round-off errors

when does this happen?

- different SIMD register widths (horizontal add)
- fused multiply-and-add available?
- different rounding mode
- different x87 register precision
- denormalization
- number of CPUs

Floating-Point Math is Non-Associative



generalizability ↔ speed trade-off

per-thread CPU settings

non-associativity

- $(a + b) + c \neq a + (b + c)$
- different round-off errors

when does this happen?

- different SIMD register widths (horizontal add)
- fused multiply-and-add available?
- different rounding mode
- different x87 register precision
- denormalization
- → number of CPUs

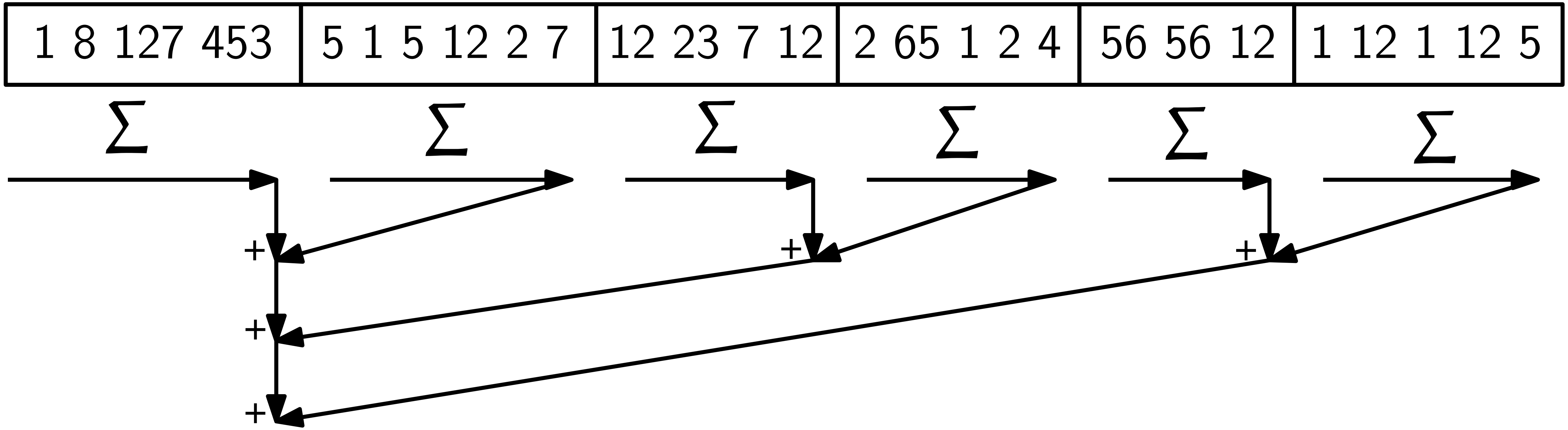
Multi-Thread and Multi-Processor Reduce

- we have to sum numbers stored across multiple CPUs
- same binary, same CPU settings on same hardware with different number of CPUs
- different results on empirical data

1 8 127 453	5 1 5 12 2 7	12 23 7 12	2 65 1 2 4	56 56 12	1 12 1 12 5

Multi-Thread and Multi-Processor Reduce

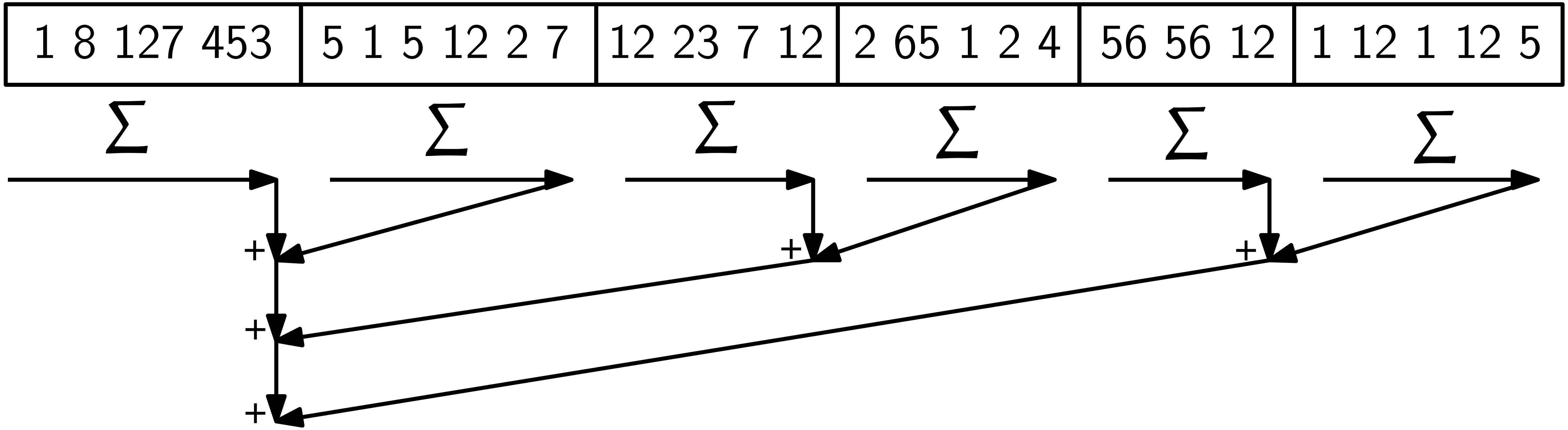
- we have to sum numbers stored across multiple CPUs
- same binary, same CPU settings on same hardware with different number of CPUs
- different results on empirical data



1 8 127 453	5 1 5 12 2 7	12 23 7 12	2 65 1 2 4	56 56 12	1 12 1 12 5

Multi-Thread and Multi-Processor Reduce

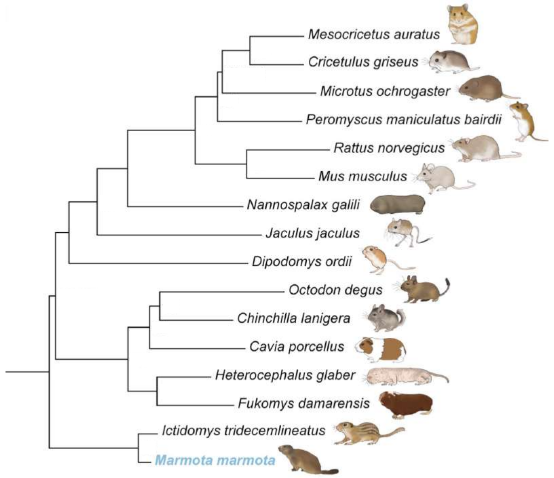
- we have to sum numbers stored across multiple CPUs
- same binary, same CPU settings on same hardware with different number of CPUs
- different results on empirical data



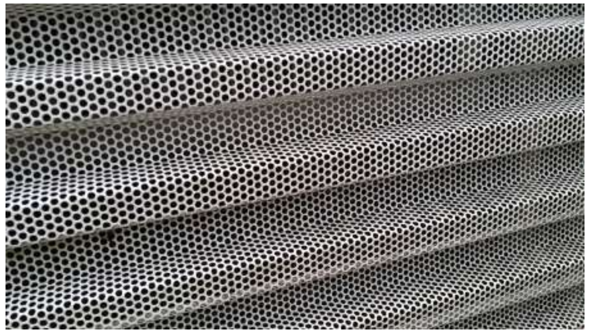
1 8 127 453	5 1 5 12 2 7	12 23 7 12	2 65 1 2 4	56 56 12	1 12 1 12 5

number of CPUs influences round-off errors

Issues in Real-World Software



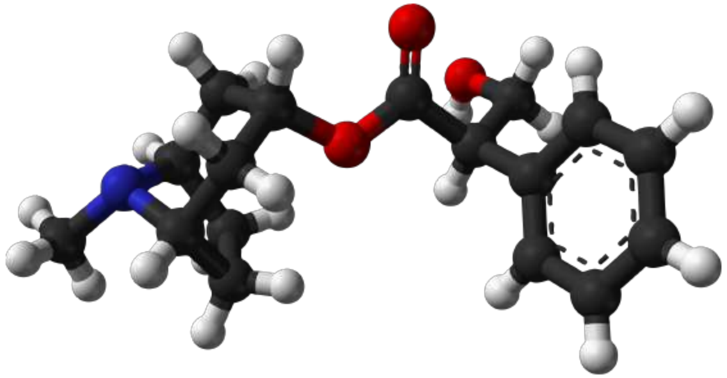
phylogenetics



sheet metal forming

fluid dynamics

climate modelling

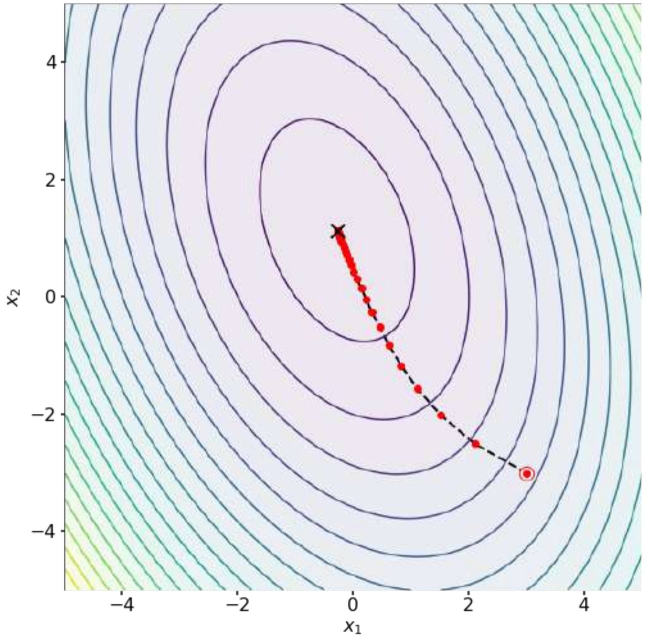


molecular dynamics

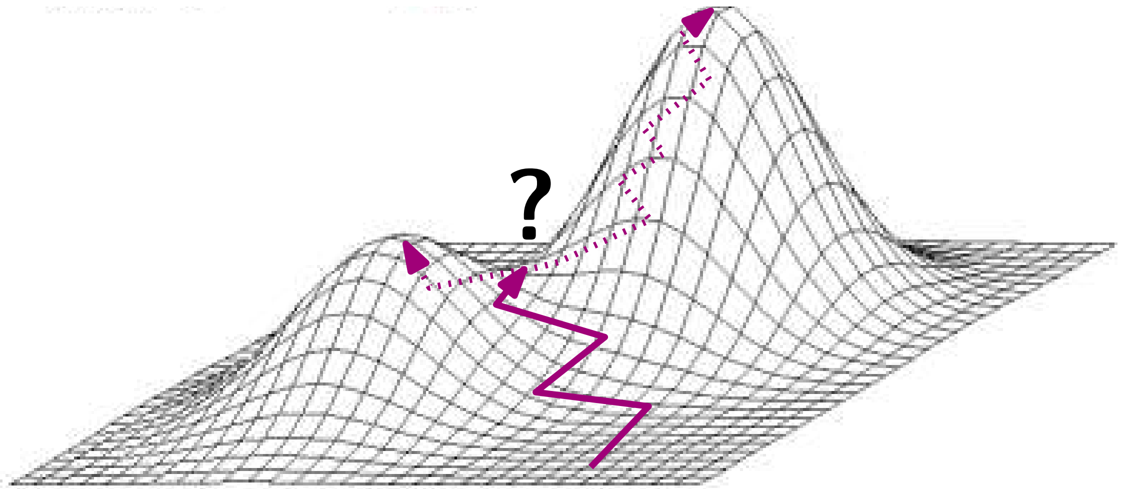
power grid analysis

Small Differences Affect High-Level Results

iterative algorithm



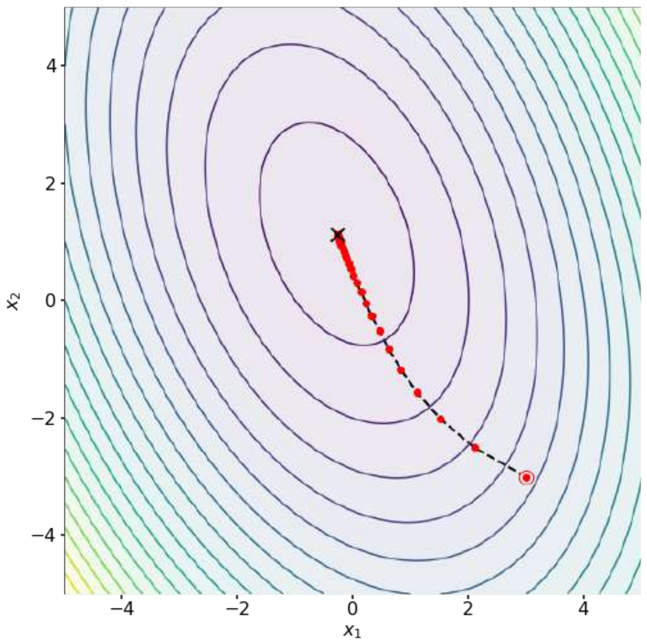
hillclimber



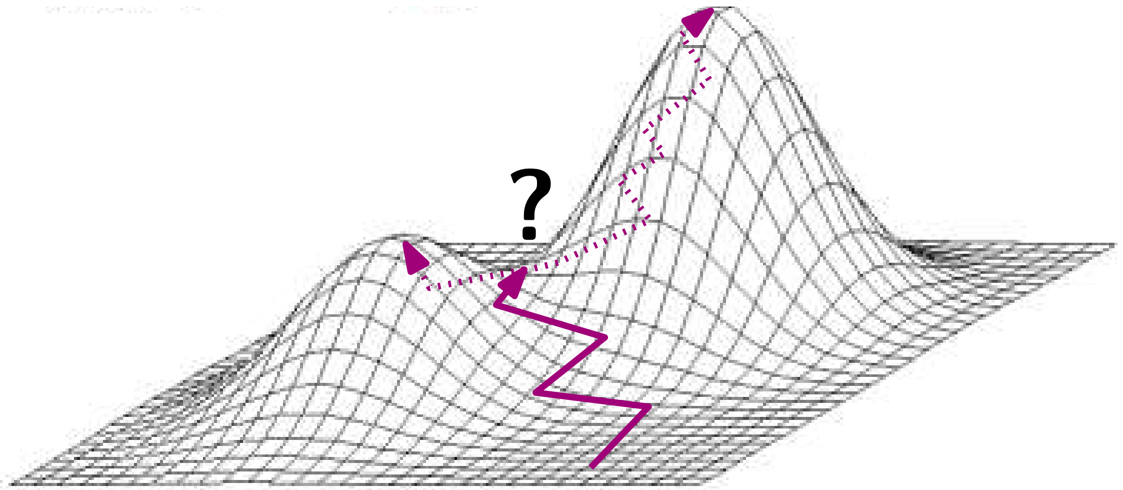
→
- different results
- different running times

Small Differences Affect High-Level Results

iterative algorithm

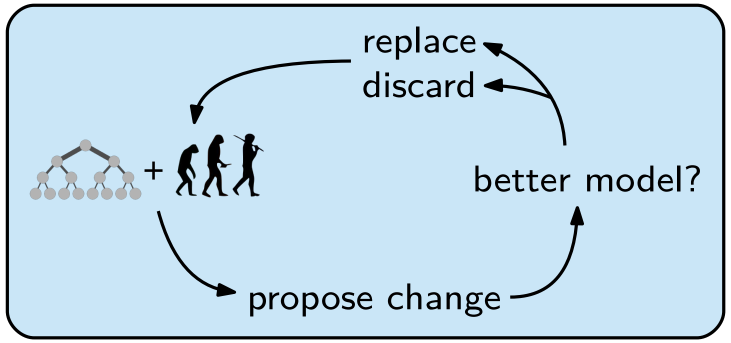


hillclimber

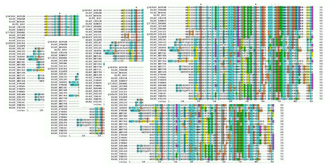


→

- different results
- different running times

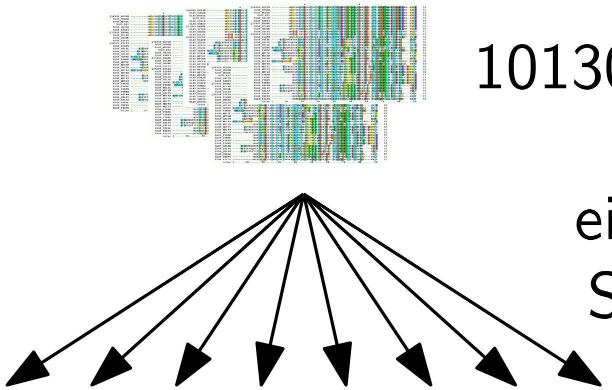


Effects in Phylogenetic Tree Search



10130 datasets

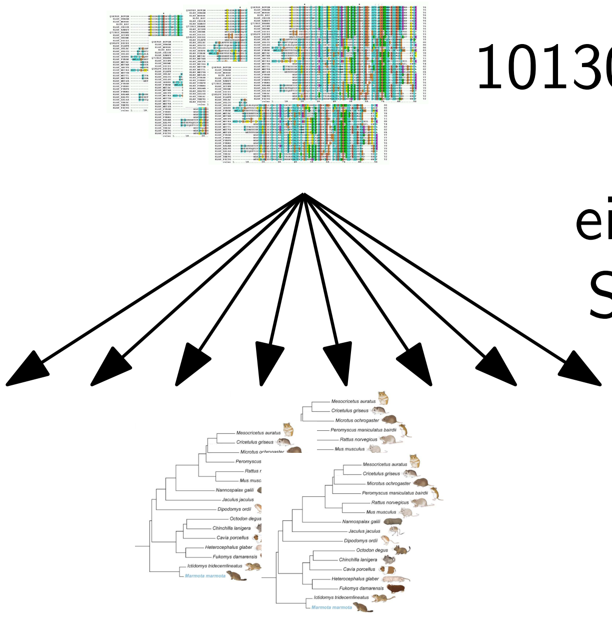
Effects in Phylogenetic Tree Search



10130 datasets

eight searches using different SIMD-variant & CPU-count

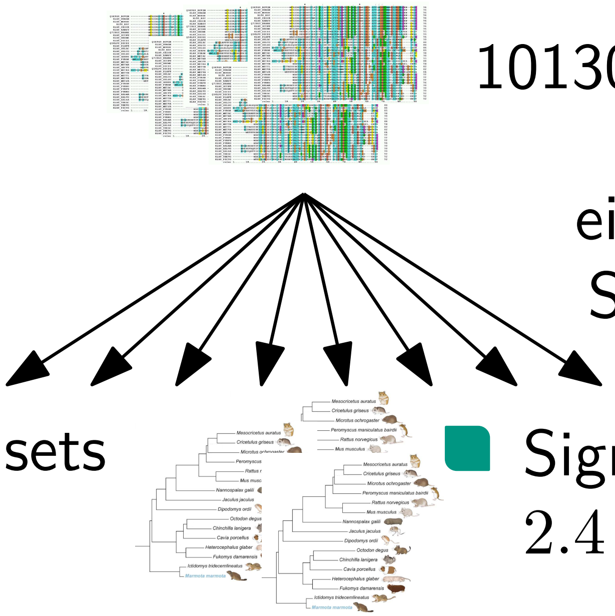
Effects in Phylogenetic Tree Search



10130 datasets

eight searches using different SIMD-variant & CPU-count

Effects in Phylogenetic Tree Search

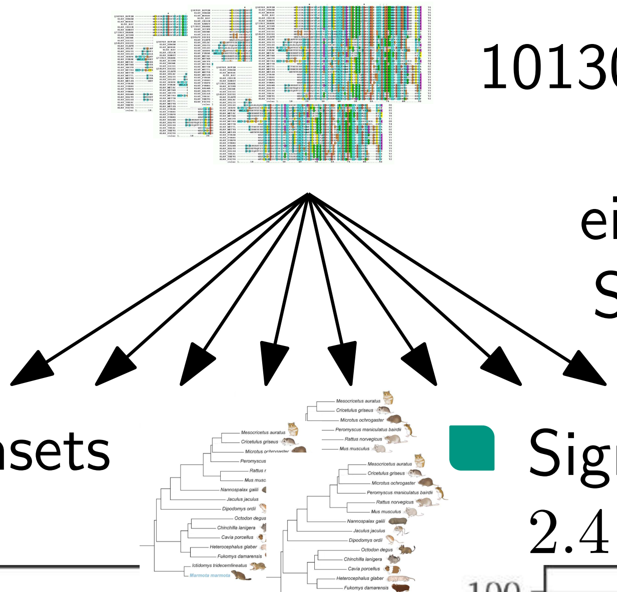


10130 datasets

eight searches using different SIMD-variant & CPU-count

- Different trees for $14.8\,\%$ of datasets
- Significantly ($p < 0.05$) different trees for $2.4\,\%$ of datasets

Effects in Phylogenetic Tree Search

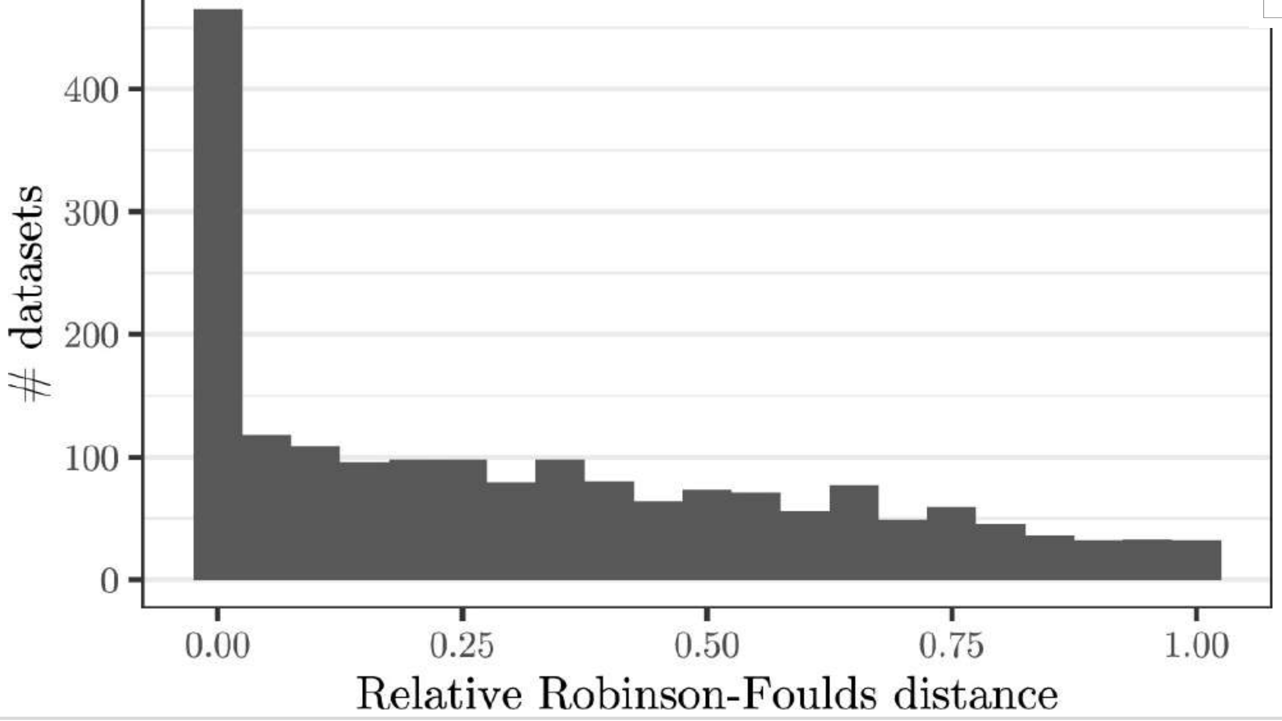


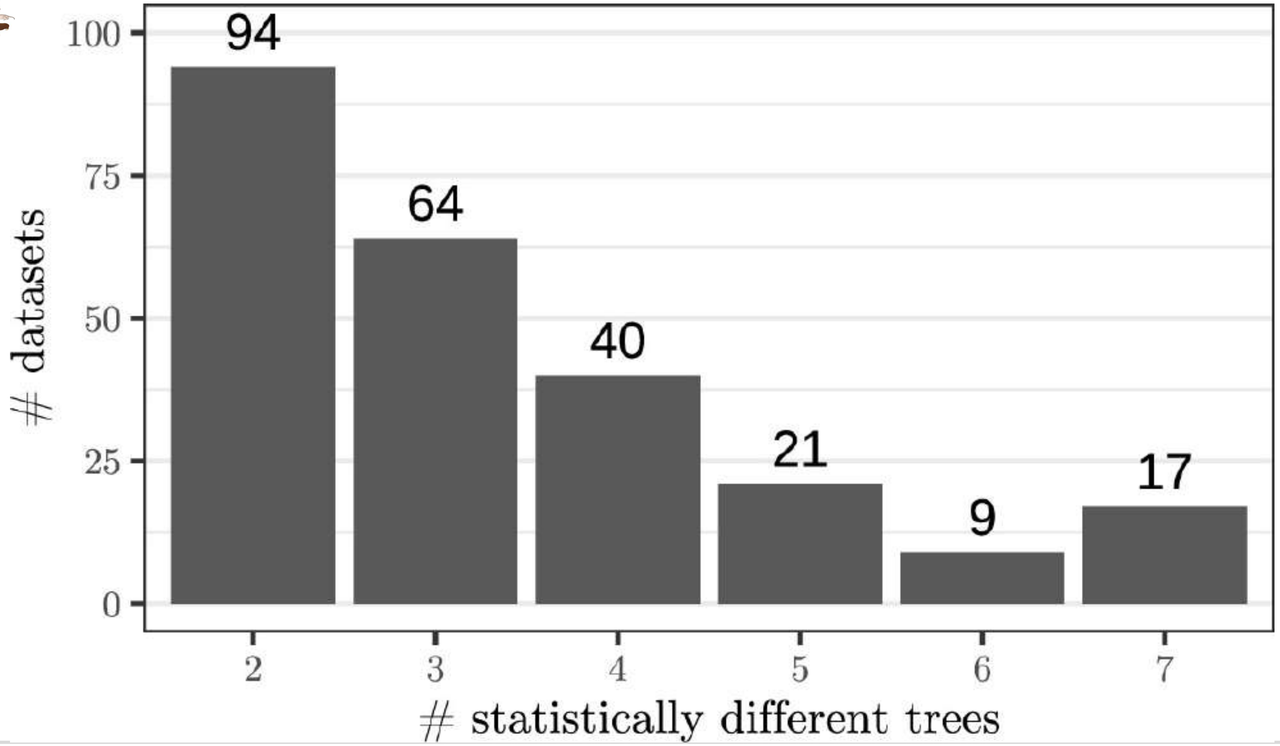
10130 datasets

eight searches using different SIMD-variant & CPU-count

- Different trees for $14.8\,\%$ of datasets

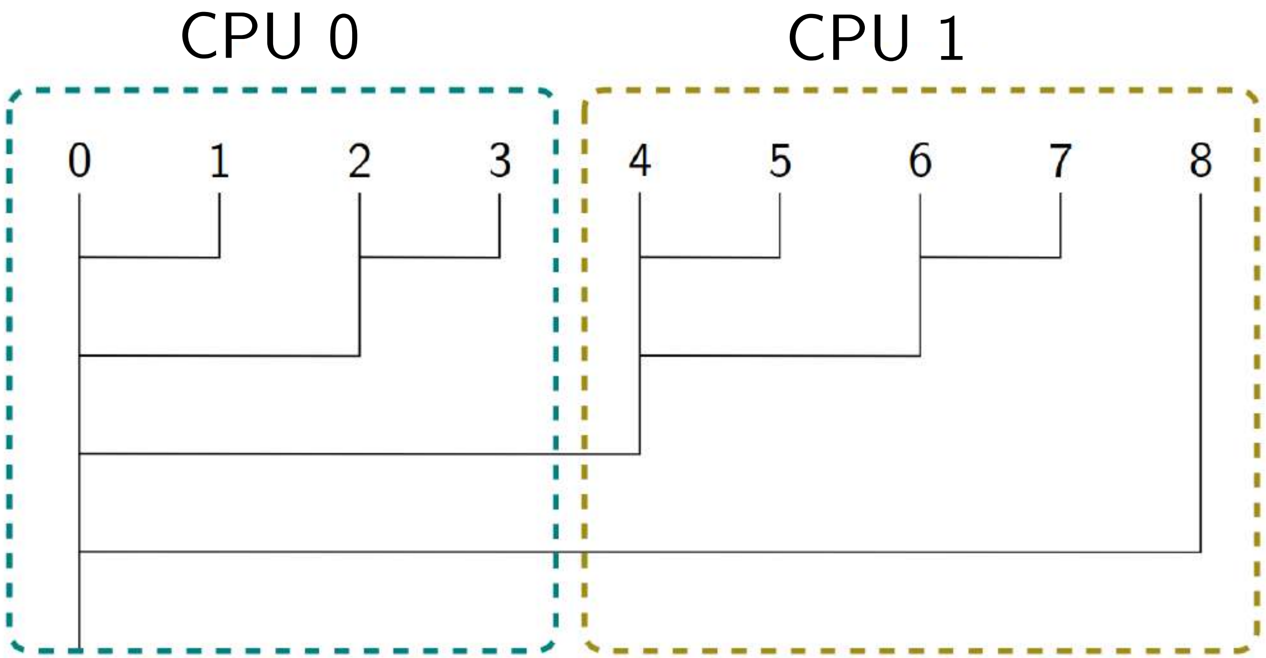
- Significantly ($p < 0.05$) different trees for $2.4\,\%$ of datasets

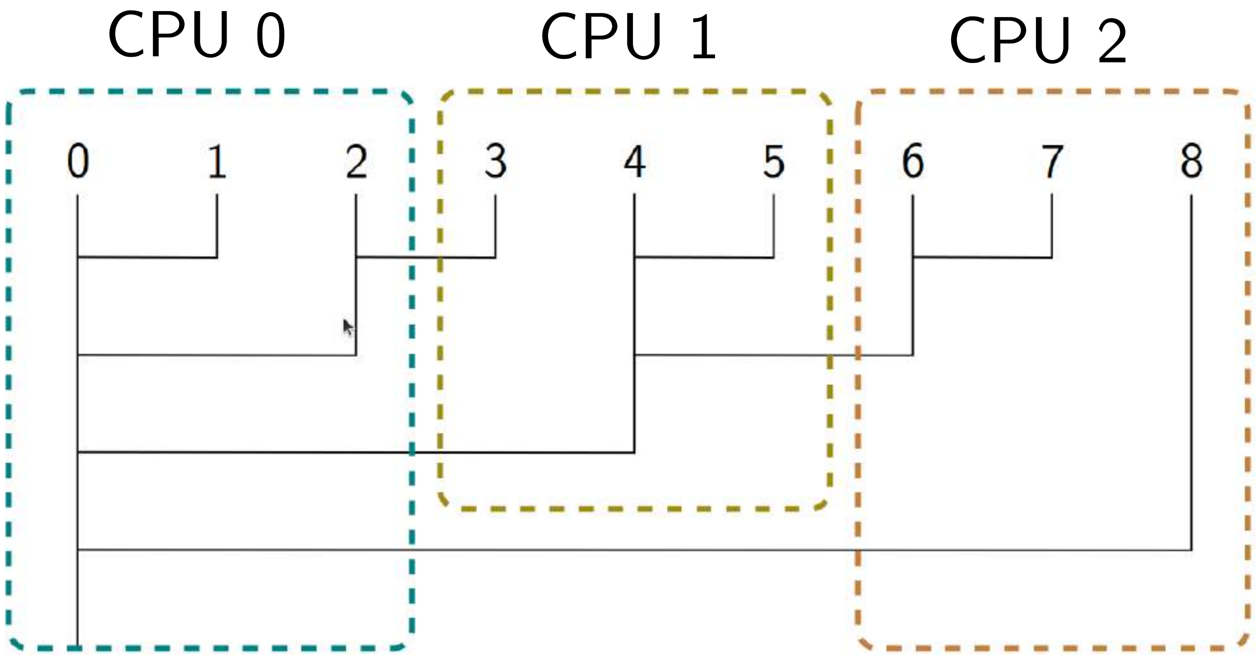




Multi-Thread and Multi-Processor Reduce

- **idea:** Do local summation as a tree, too. Send intermediate results over network
- same order of summation → same round-off error
- cache messages, use base case, and *k*-ary trees to improve performance



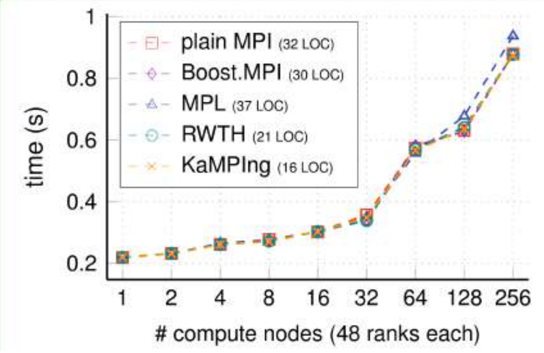


Summary Reproducible Reduction

- IEEE754 floating-point math is **non-associative**
- different CPU-counts affect result of reduction
- these low-level differences propagate up to high-level results
- fixing the order of operations is the only method agnostic of the reduction operation
- we employ message buffering, a k-ary reduction tree, and a base-case to make the algorithm faster in practice

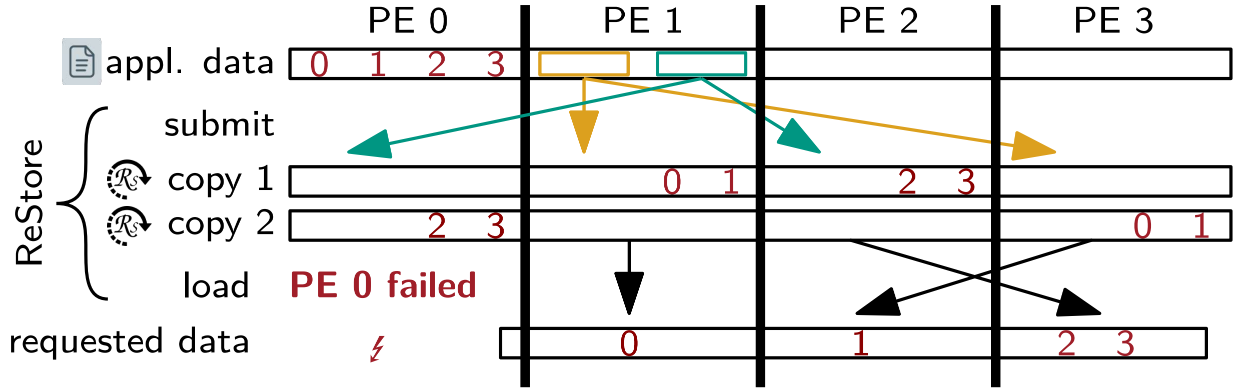
Overview

zero-overhead C++ MPI wrapper and distributed toolbox [SC24]



```
recv_buf = comm.allgatherv(send_buf(v_local));
```

replicated storage for rapid recovery after CPU failure [FTXS22]



reproducible distributed memory reduction

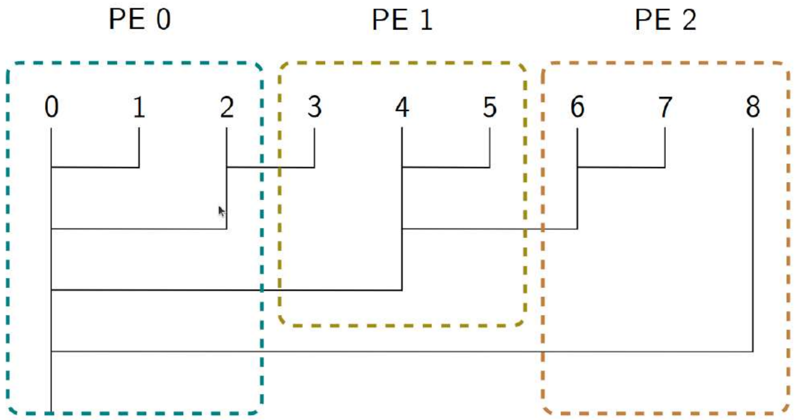
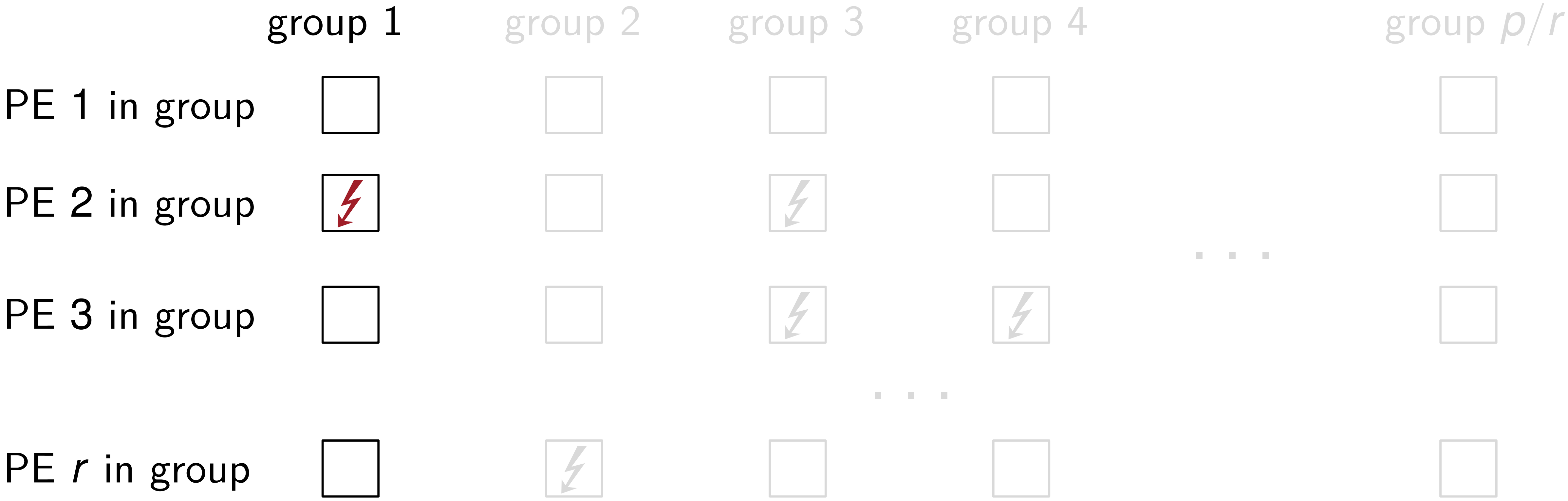


Image Sources

- *Phylogenetics of Rodents*: Toni I. Gossmann, Achchuthan Shanmugasundram, Stefan Börno, John J. Welch, Bernd Timmermann, Markus Ralser: Ice-Age Climate Adaptations Trap the Alpine Marmot in a State of Low Genetic Diversity. Current Biology, VOLUME 29, ISSUE 10, P1712-1720.E7, Mai, 2019, DOI: 10.1016/j.cub.2019.04.020

Probability of Irrecoverable Data Loss

Given f failures, what is the probability, that all copies of group 1 failed?

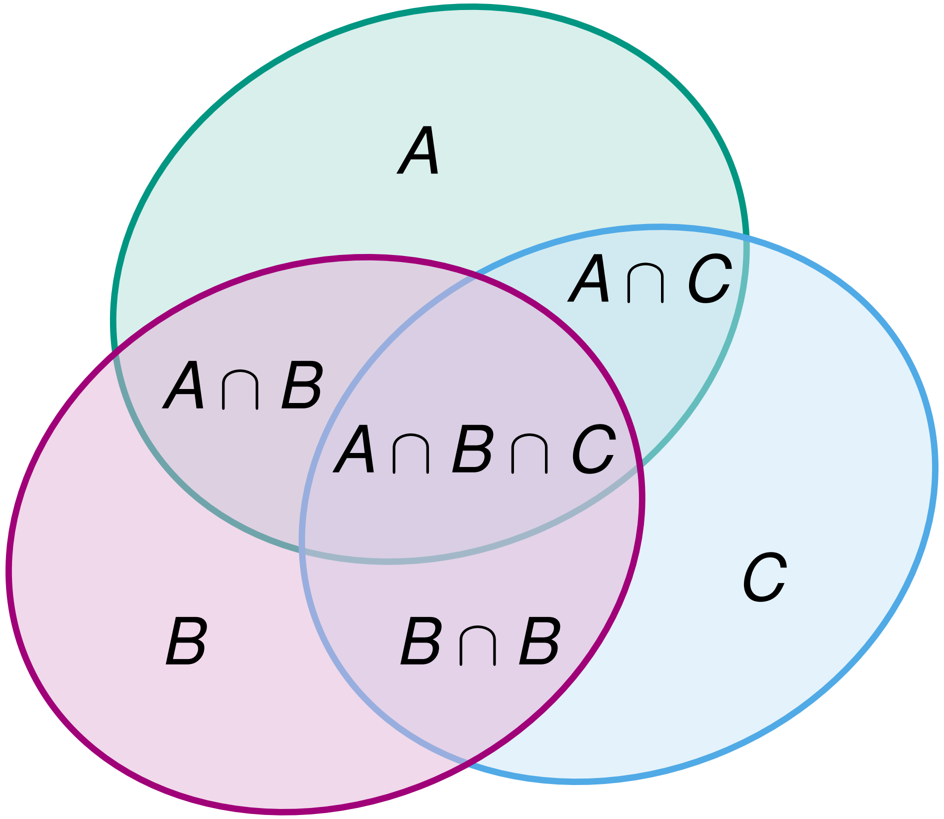


- Number of possibilities to draw f nodes from p nodes: $\binom{p}{f}$
- Number of possibilities to draw all r copies of group 1 plus $f - r$ other nodes: $\binom{p-r}{f-r}$
- $P(\text{All nodes of group 1 failed}) = \binom{p-r}{f-r} / \binom{p}{f}$

Probability of Irrecoverable Data Loss

Inclusion-exclusion principle

$$\begin{aligned}|A \cup B \cup C| =&|A| + |B| + |C|\\ &-|A \cap B| - |A \cap C| - |B \cap C|\\ &+|A \cap B \cap C|\end{aligned}$$



Probability of Irrecoverable Data Loss

- Given f, there are $\binom{p-r}{f-r}$ configurations of failed nodes which lead to data loss
- Summing up over all groups would count certain states twice, trice, . . .
- E.g., states in which *all* nodes of group 1 and group 2 failed would be counted twice

$$P^{\leq}_{\mathrm{IDL}}(f) = \sum_{j=1}^{g} (-1)^{j+1} \binom{g}{j} \frac{\binom{p-jr}{f-jr}}{\binom{p}{f}}$$

Probability of Irrecoverable Data Loss

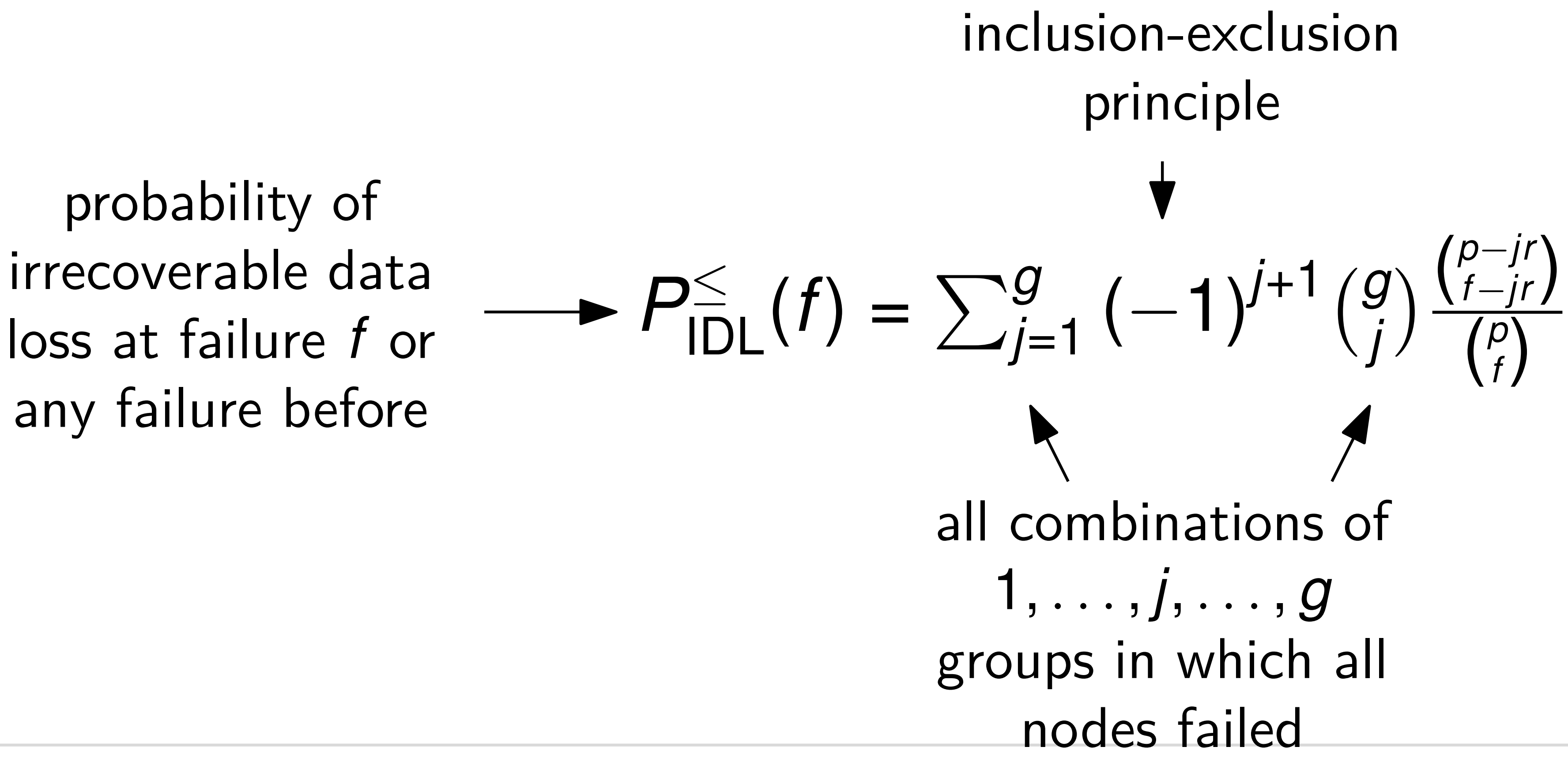
- Given f, there are $\binom{p-r}{f-r}$ configurations of failed nodes which lead to data loss
- Summing up over all groups would count certain states twice, trice, . . .
- E.g., states in which *all* nodes of group 1 and group 2 failed would be counted twice

probability of
irrecoverable data
loss at failure f or
any failure before

$$\longrightarrow P^{\leq}_{\mathrm{IDL}}(f) = \sum_{j=1}^{g} (-1)^{j+1} \binom{g}{j} \frac{\binom{p-jr}{f-jr}}{\binom{p}{f}}$$

Probability of Irrecoverable Data Loss

- Given f, there are $\binom{p-r}{f-r}$ configurations of failed nodes which lead to data loss
- Summing up over all groups would count certain states twice, trice, . . .
- E.g., states in which *all* nodes of group 1 and group 2 failed would be counted twice



inclusion-exclusion principle

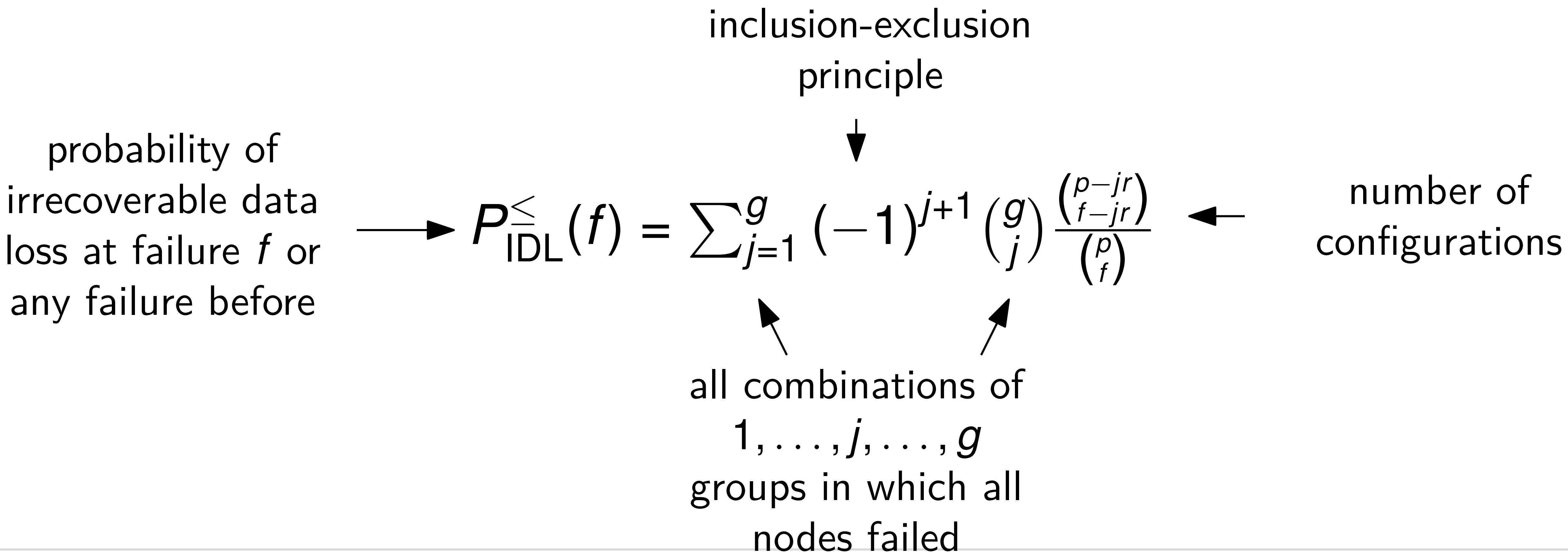
probability of irrecoverable data loss at failure f or any failure before

$$P_{\text{IDL}}^{\leq}(f) = \sum_{j=1}^{g} (-1)^{j+1} \binom{g}{j} \frac{\binom{p-jr}{f-jr}}{\binom{p}{f}}$$

all combinations of $1, \ldots, j, \ldots, g$ groups in which all nodes failed

Probability of Irrecoverable Data Loss

- Given f, there are $\binom{p-r}{f-r}$ configurations of failed nodes which lead to data loss
- Summing up over all groups would count certain states twice, trice, . . .
- E.g., states in which *all* nodes of group 1 and group 2 failed would be counted twice



$$P^{\le}_{\mathrm{IDL}}(f) = \sum_{j=1}^{g} (-1)^{j+1} \binom{g}{j} \frac{\binom{p-jr}{f-jr}}{\binom{p}{f}}$$

inclusion-exclusion principle

probability of irrecoverable data loss at failure f or any failure before

number of configurations

all combinations of $1, \ldots, j, \ldots, g$ groups in which all nodes failed

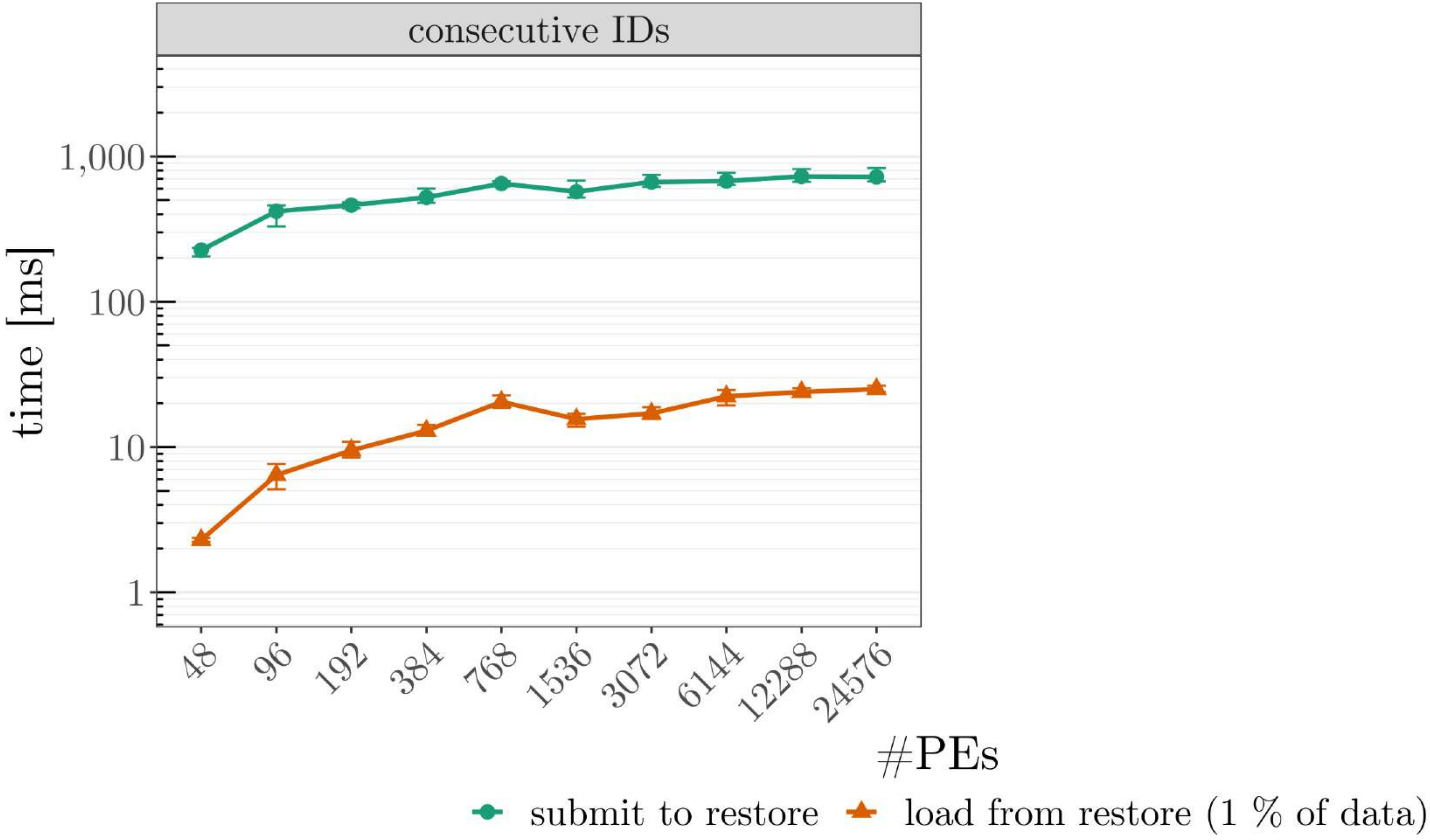
Related Work

	ftRMA	Fenix	SCR	Lu	GPI_CP	*ReStore*
Features						
in-memory checkpointing	✓	✓	✗	✓	✓	✓
substituting recovery	✓	✓	✓	✓	✓	✓
shrinking recovery	✗	✗	✗	✗	✗	✓
all nodes participate in computation	✗[2]	(✓)[1]	(✓)[1]	✗[2]	(✓)[1]	✓
scaleable recovery	✗	✗	✗	✗	✗	✓
programming model	MPI RDMA	MPI	MPI	MPI	PGAS/GPI	MPI

[1] Need for nodes idling until they replace a failing node

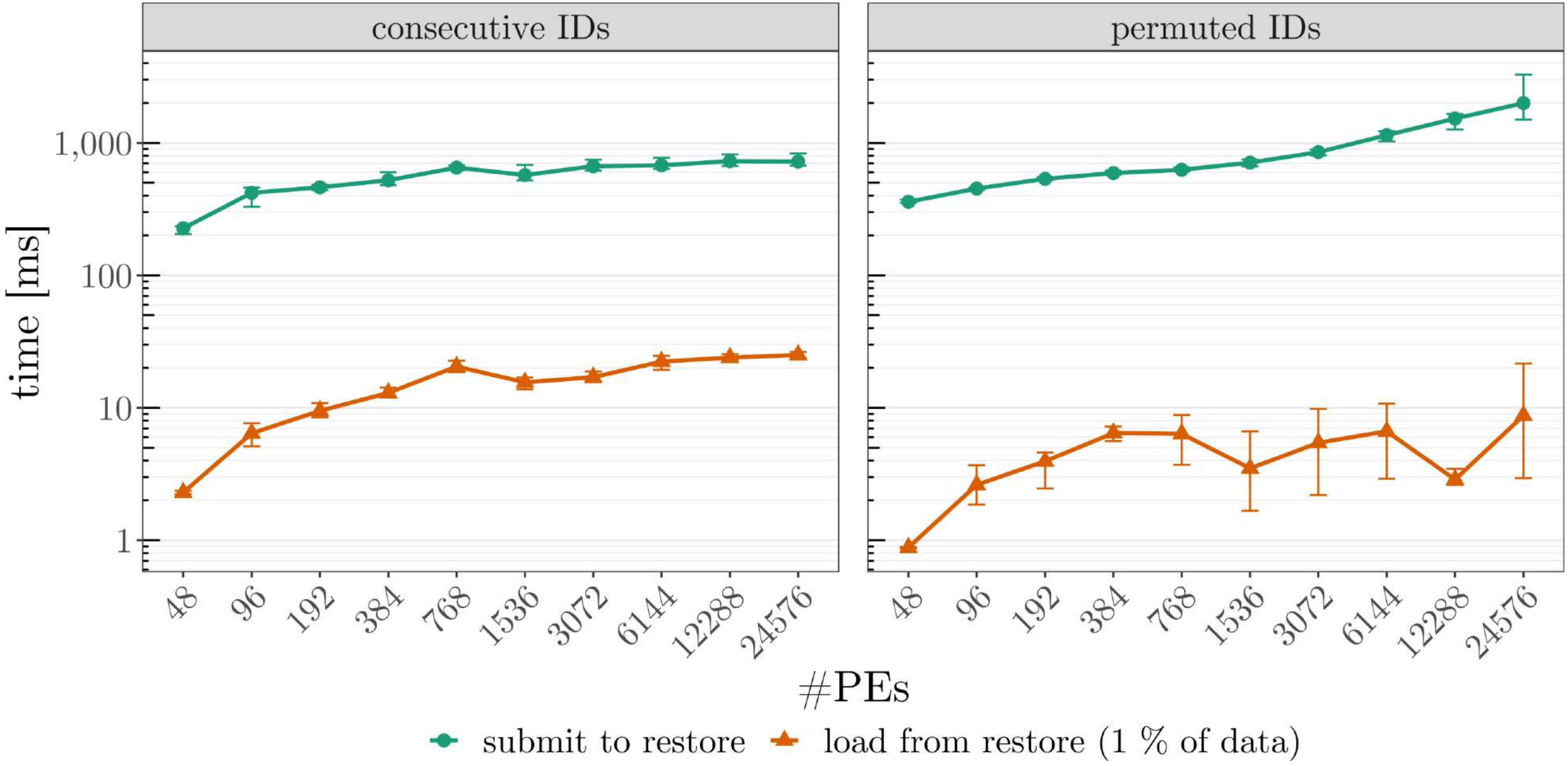
[2] Additionally, some nodes used solely to store checkpoints

Evaluating ID Randomization



16 MiB/PE

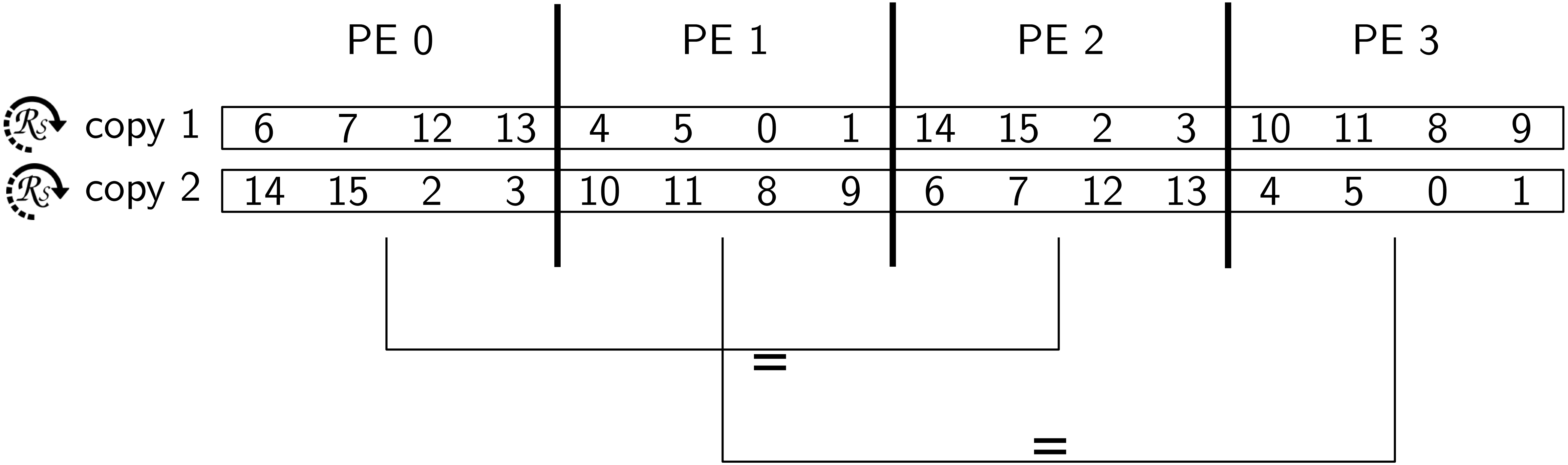
Evaluating ID Randomization



16 MiB/PE

Probability of Irrecoverable Data Loss

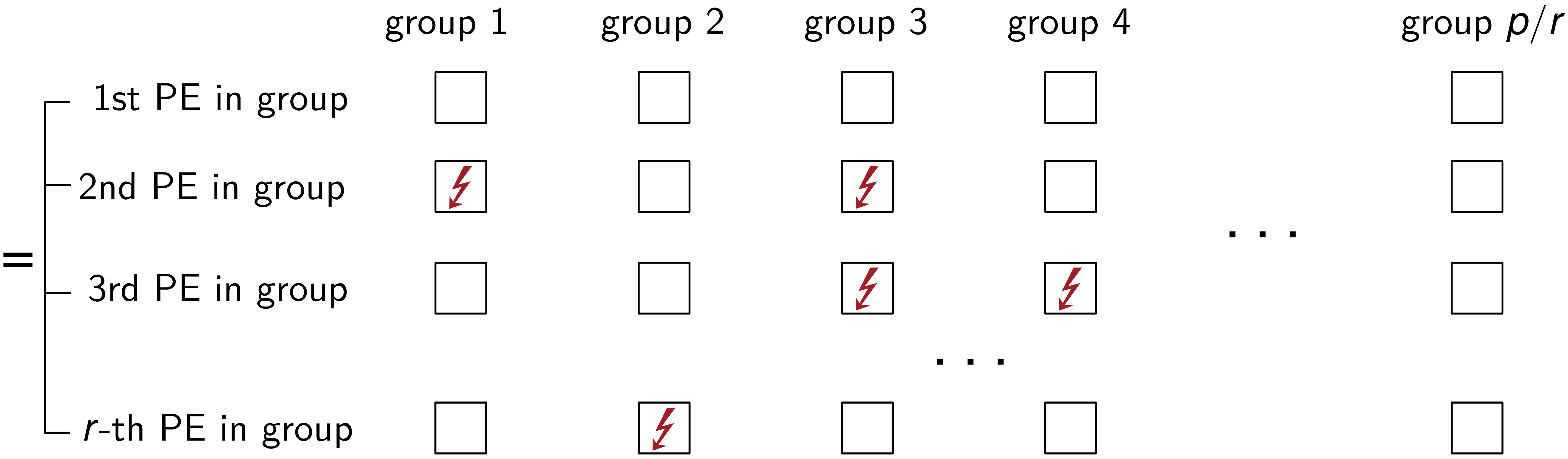
Number of replicas r divides number of PEs p
→ *groups* of PEs storing the same data



	PE 0				PE 1				PE 2				PE 3			
copy 1	6	7	12	13	4	5	0	1	14	15	2	3	10	11	8	9
copy 2	14	15	2	3	10	11	8	9	6	7	12	13	4	5	0	1

Probability of Irrecoverable Data Loss

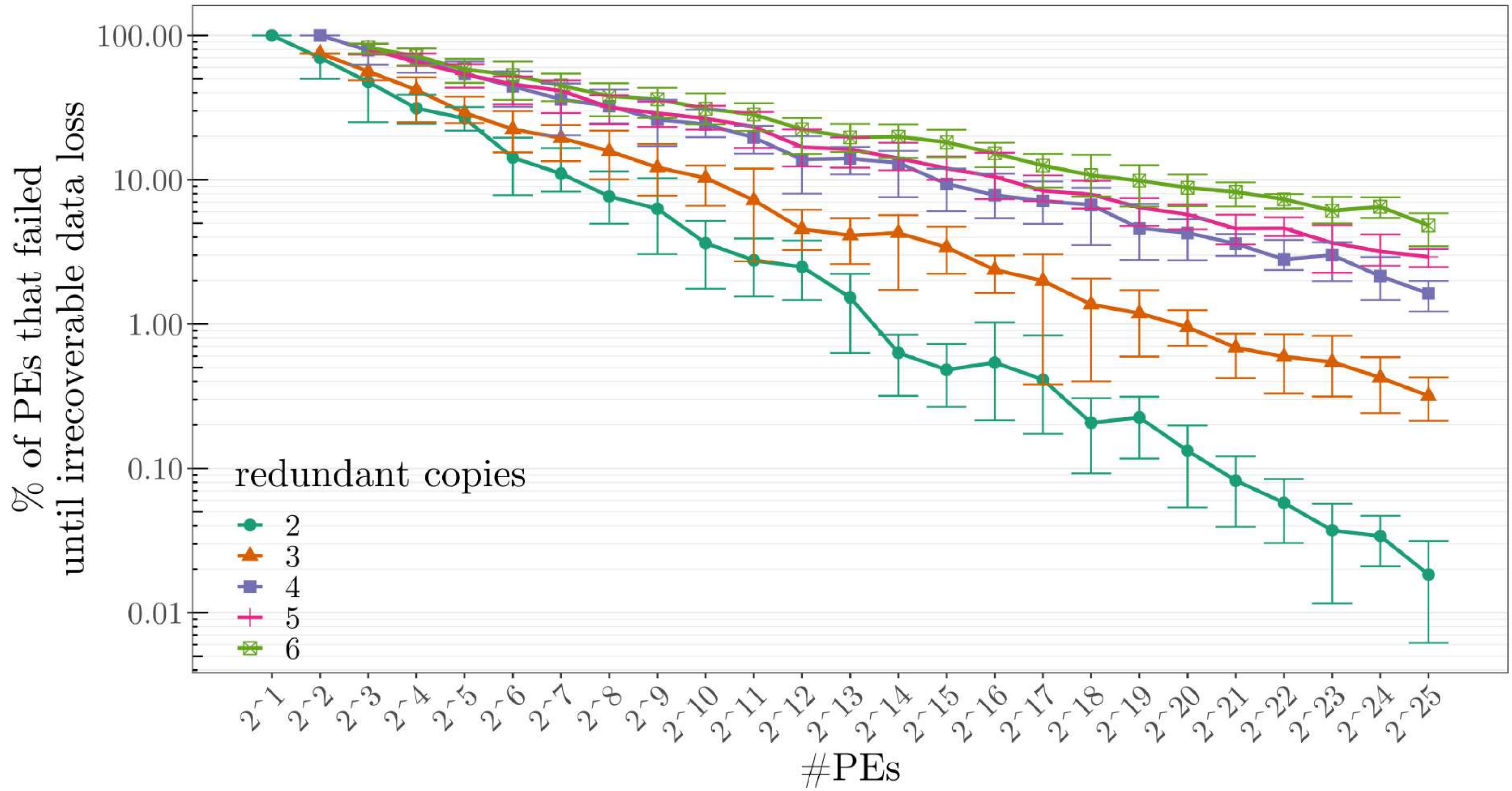
Given f failures, what is the probability, that all PEs of any group failed?



=

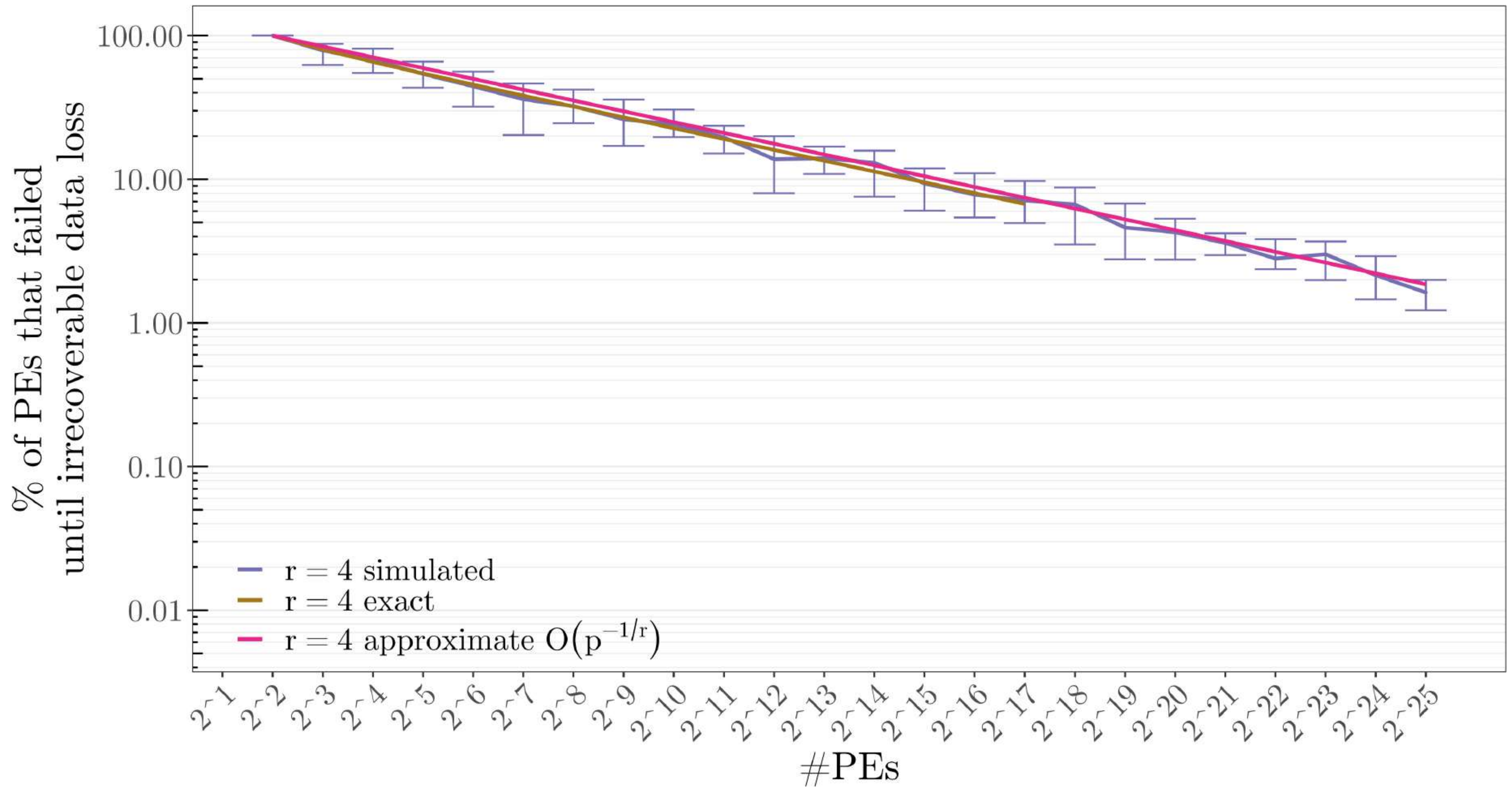
	group 1	group 2	group 3	group 4	…	group p/r
1st PE in group	☐	☐	☐	☐		☐
2nd PE in group	⚡	☐	⚡	☐		☐
3rd PE in group	☐	☐	⚡	⚡		☐
…						
r-th PE in group	☐	⚡	☐	☐		☐

Probability of Irrecoverable Data Loss



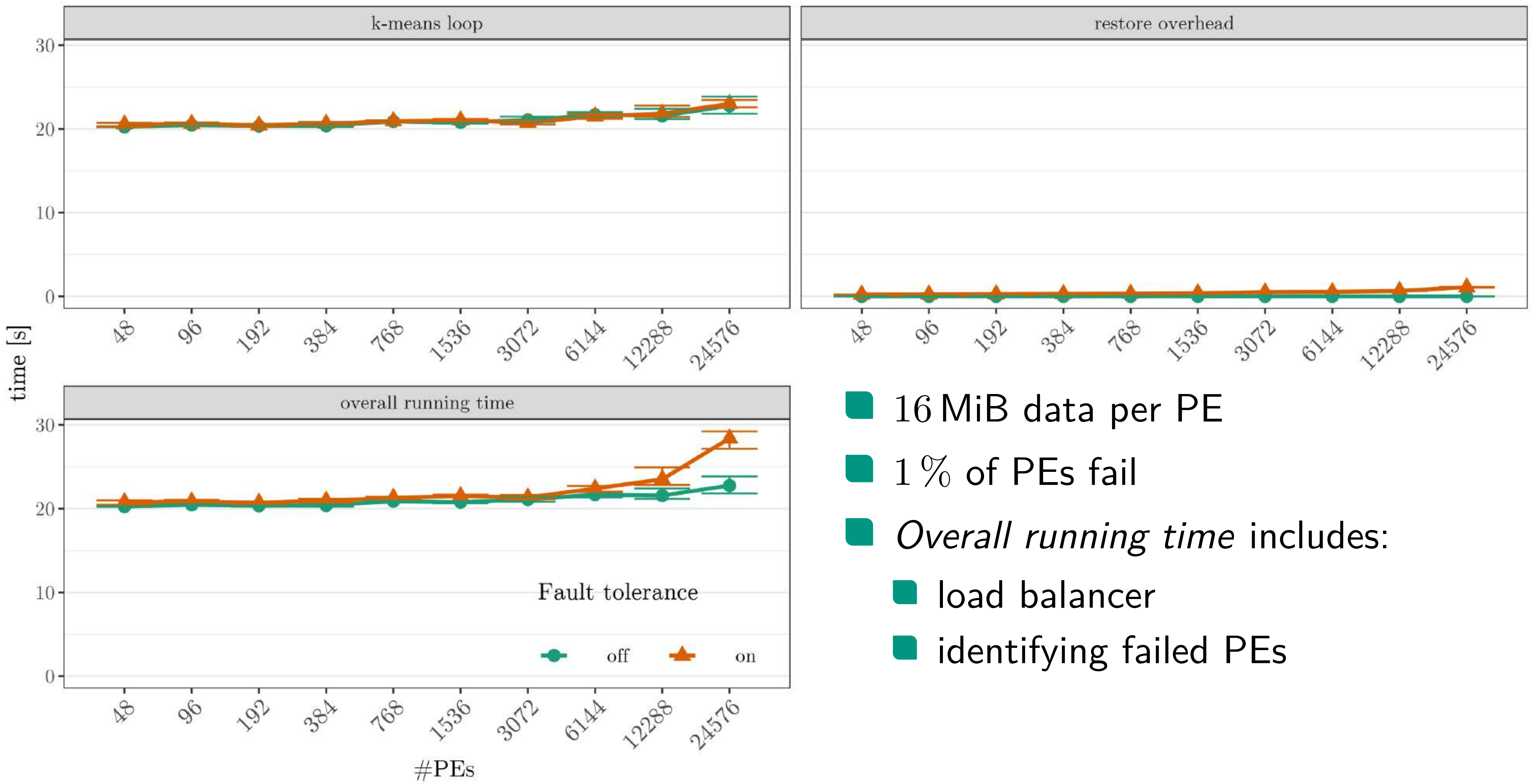
empirical

Probability of Irrecoverable Data Loss



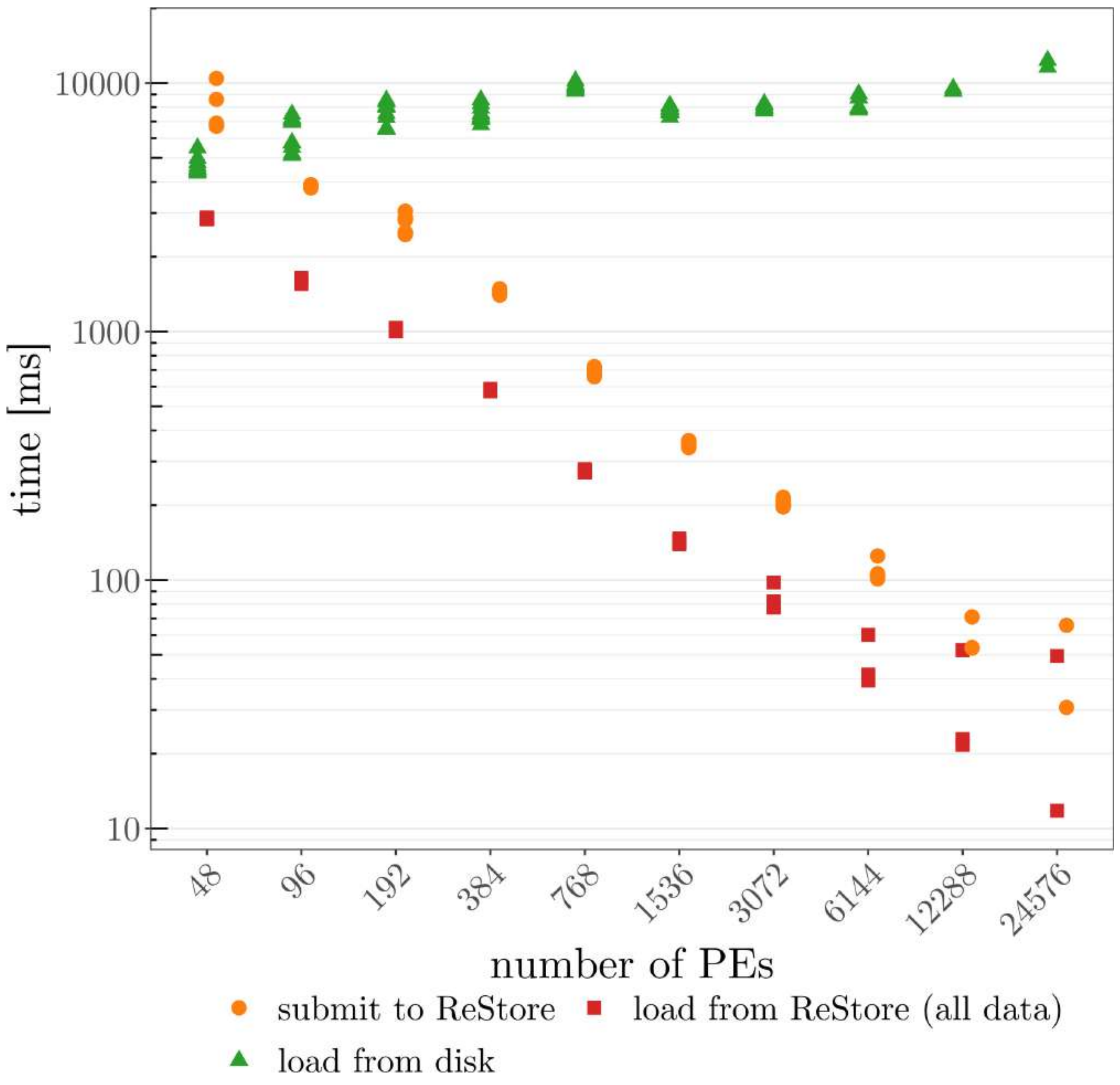
theoretical

Overhead of ReStore in k-means



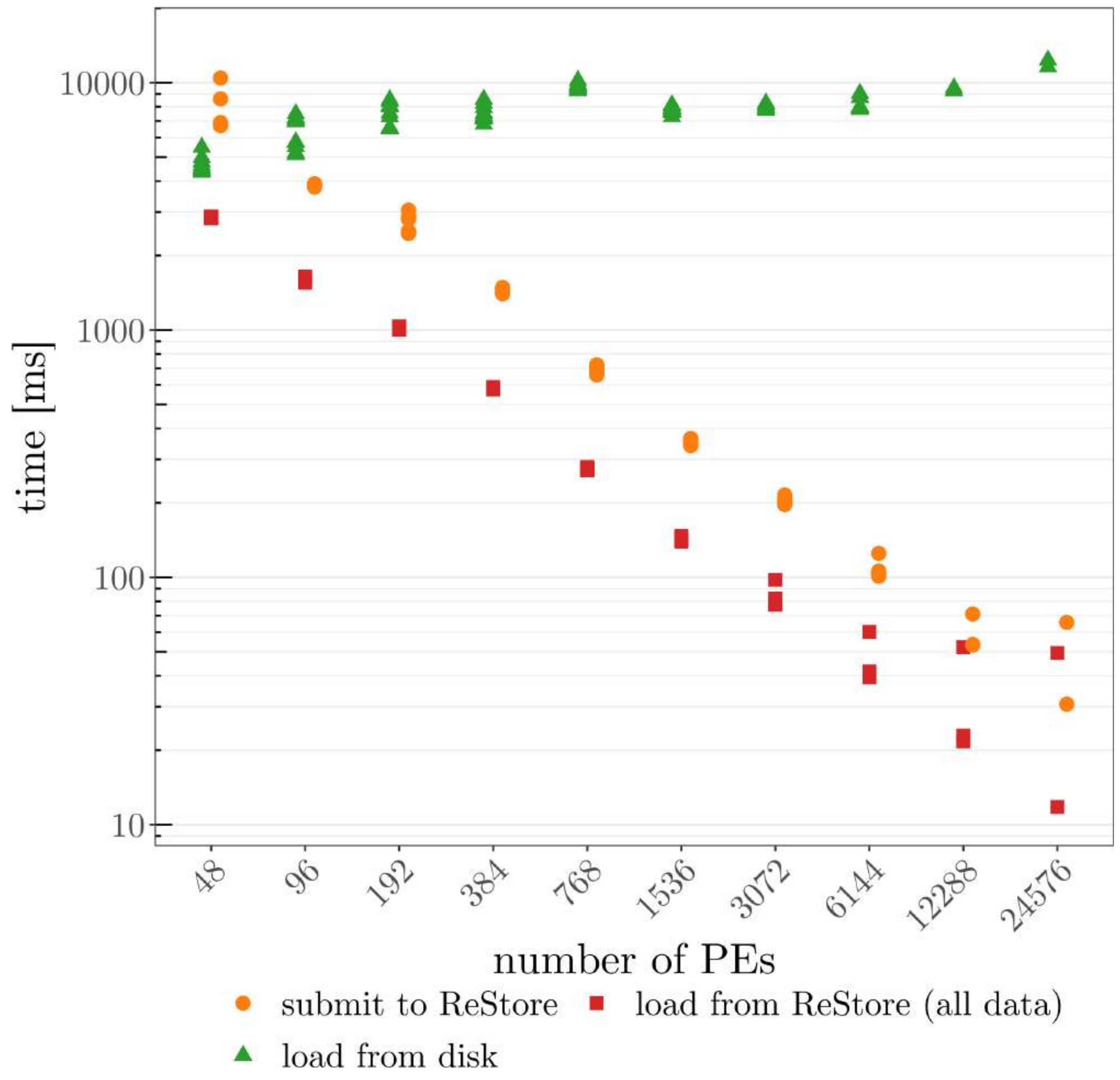
- 16 MiB data per PE
- $1\,\%$ of PEs fail
- *Overall running time* includes:
  - load balancer
  - identifying failed PEs

Overhead of ReStore in RAxML-NG

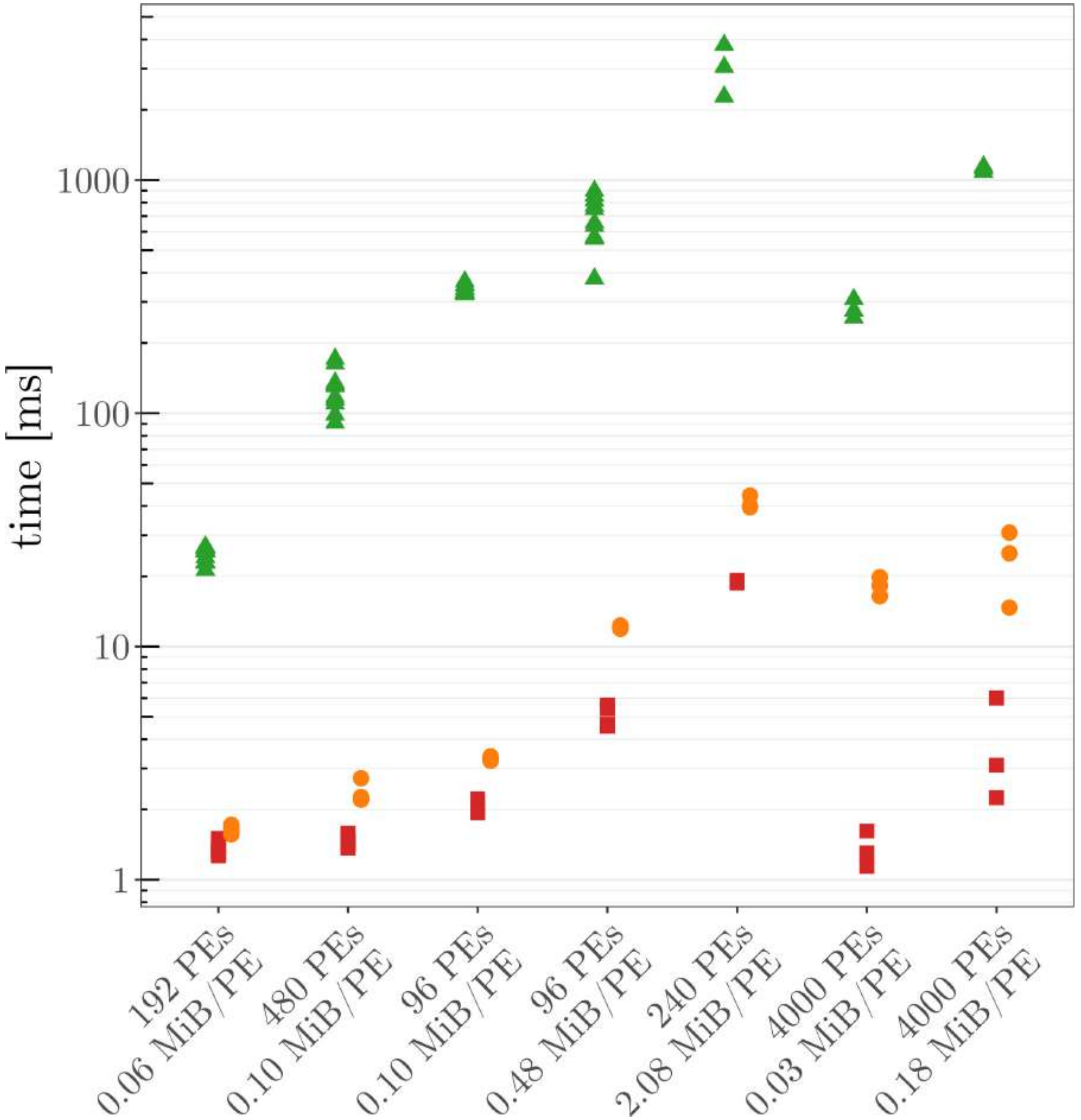


19.1 GiB synthetic dataset

Overhead of ReStore in RAxML-NG



19.1 GiB synthetic dataset



empirical datasets