A Toolset for Debugging Non-deterministic MPI Applications

AICS/RIKEN Talk
February 9th, 2016

Kento Sato
Lawrence Livermore National Laboratory

LLNL-PRES-681938
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Reproducibility (再現性)

Outline

- Reproducibility/Irreproducibility in HPC
- Existing toolset for irreproducibility bugs
  - Spindle
  - ReMPI (MPI record-and-replay)
  - Io-watchdog
  - STAT
- Clock delta compression for ReMPI

Reproducibility (再現性)

- Reproducibility is an ability to reproduce the same/similar experimental results even by other persons
- Important to show that the experiment is valid and correct

Reproducible
by other persons

- In computational science, relatively easy to reproduce behavior of applications, and the final numerical results
- But, several factors hamper reproducibility

What hampers reproducibility in computation ?

Experimental environment

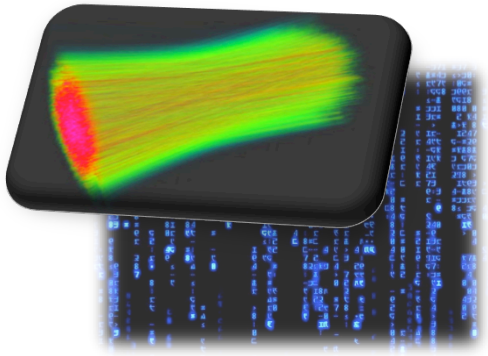
Hardware

- CPU/GPU model
- Memory
- Network
- Storage system
- and others …

Software

- Kernel version
- Compiler version & option
- Library version
- and others …

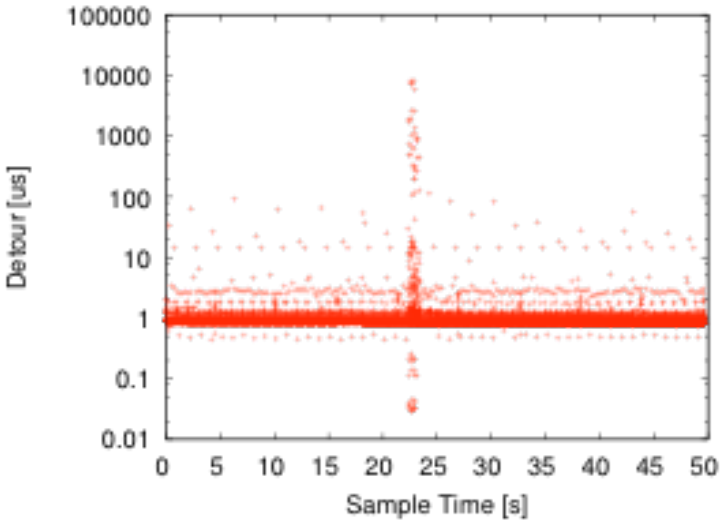
Other factors



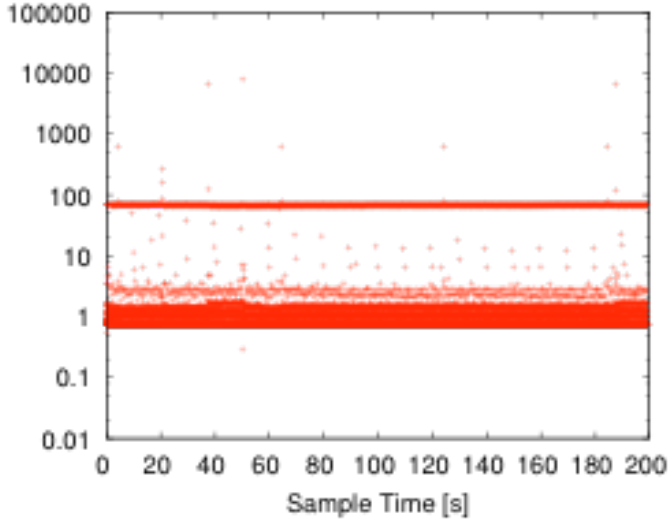
Different behaviors
Different numerical results

Impact of OS noises to applications

- Netgauge: Benchmark for OS noises
- System daemons periodically wake up and run



(a) CHiC (Linux)



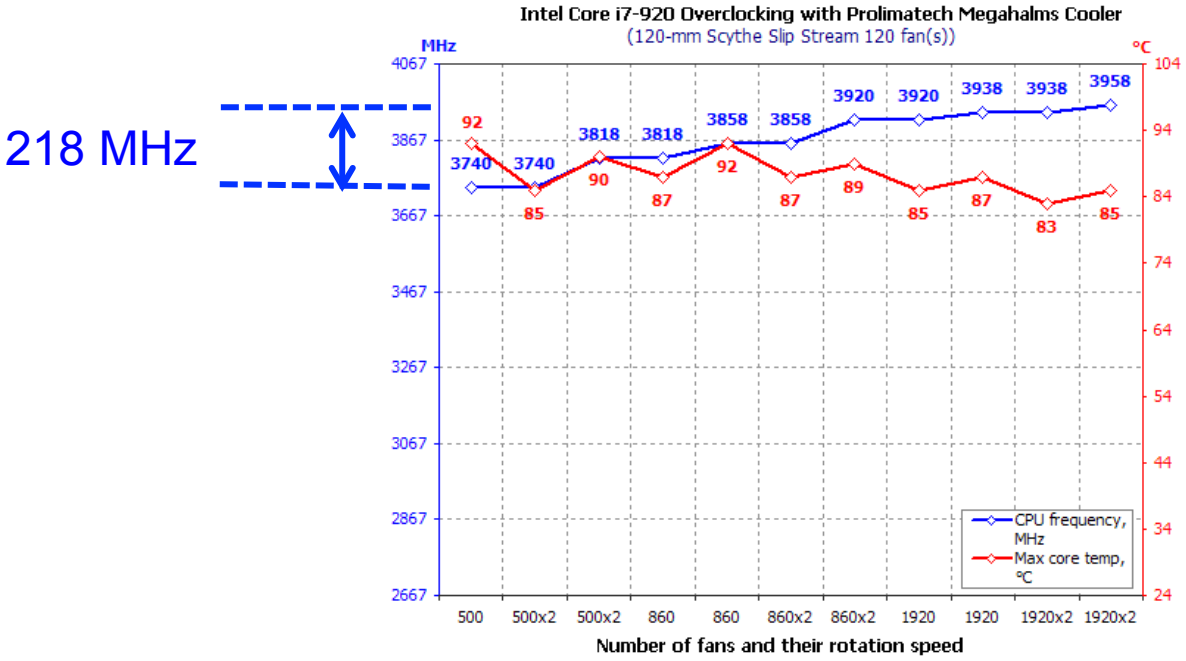
(d) Jaguar

OS noise fluctuations hamper reproducibility

Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)

Impact of temperature to applications

- CPU temperature v.s. CPU frequency



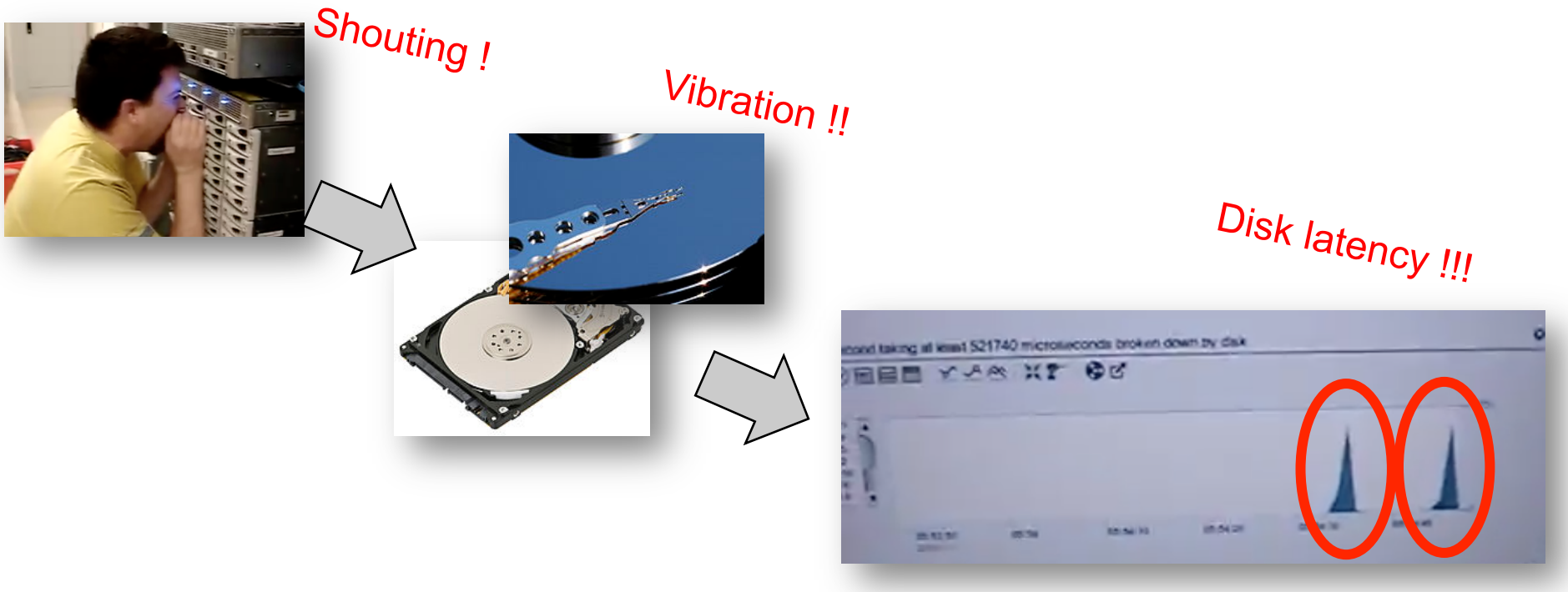
Source: http://www.xbitlabs.com/articles/coolers/display/core-i7-coolers-roundup_17.html

Fan speed: A single fan x 500RPM ➔ Two fans x 192RPM

Temperature hampers reproducibility

Impact of “human” noise to applications

1. The guys shout to the disk drives
2. The disk drives detect vibration, and suspend the disk head
3. The latencies rise



Human noise may hamper reproducibility

What hampers reproducibility in computation ?

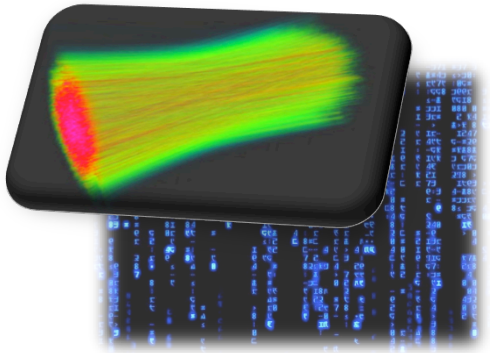
Experimental environment

Hardware

- CPU/GPU model
- Memory
- Network
- Storage system
- and others …

Software

- Kernel version
- Compiler version & option
- Library version
- and others …



Different behaviors
Different numerical results

Other factors

- System noises
- Temperature
- Physical noises

Out of our control

Several factors make MPI applications non-deterministic and irreproducible

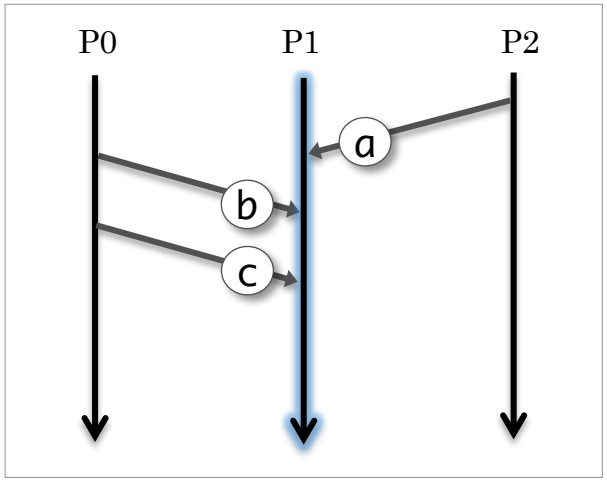
Now, reproducibility is a common issue in HPC

BoF	Tue. Nov 17, 15:30-17:00	Performance Reproducibility in HPC - Challenges and State-of-the-Art	13B
Invited talk	Wed, Nov 18, 13:30 – 14:15	Reproducibility in High Performance Computing	Ballroom D
Tech. paper	Thu, Nov 19 11:00 – 11:30	Clock Delta Compression for Scalable Order-Replay of Non-Deterministic Parallel Applications	18AB
BoF	Thu, Nov 19, 12:15 – 13:15	Reproducibility of High Performance Codes and Simulations – Tools, Techniques, Debugging	17AB
Tech. paper	Thu, Nov 19, 13:30 – 14:00	Scientific Benchmarking of Parallel Computing Systems	18AB
Workshop	Fri, Nov 20, 8:00 – 12:00	NRE2015: Numerical Reproducibility at Exascale	Hilton 400 - 402

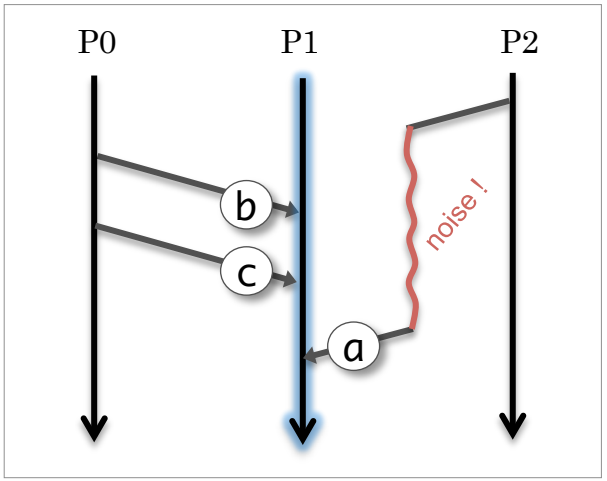
Student Cluster Competition	* SC16 Student Cluster Competition will be abont reproducibility

What is MPI non-determinism (ND) ?

- Message receive orders can be different across executions
  - Unpredictable system noise (e.g. network, system daemon & OS jitter)
- Arithmetic orders can also change across executions



Execution A: (a+b)+c

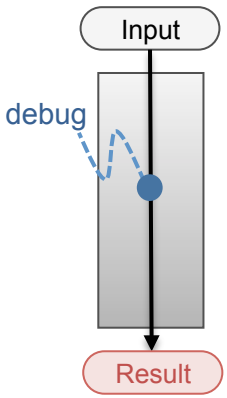


Execution B: a+(b+c)

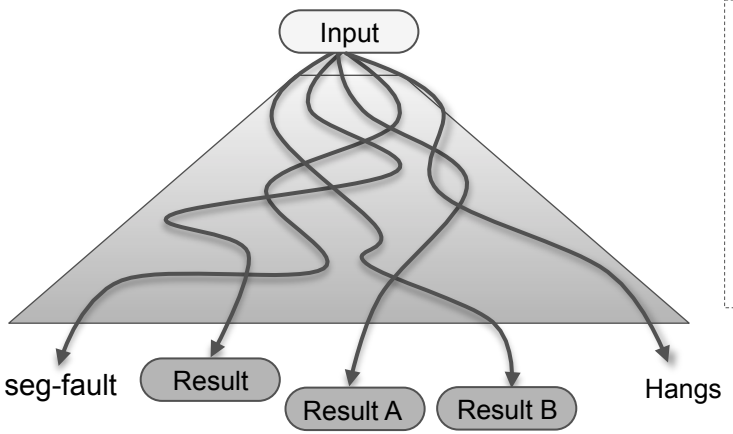
Non-determinism also increases debugging cost

- Control flows of an application can change across different runs

Deterministic apps



Non-deterministic apps



- Non-deterministic control flow
  - Successful run, seg-fault or hang
- Non-deterministic numerical results
  - Floating-point arithmetic is "NOT" necessarily associative

$$(a+b)+c \neq a+(b+c)$$

➔ Developers need to do debug runs until the same bug is reproduced

➔ Running as intended ? Application bugs ? Silent data corruption ?

In ND applications, it's hard to reproduce bugs and incorrect results,
It costs excessive amounts of time for "reproducing", finding and fixing bugs

"On average, software developers spend 50% of their programming time finding and fixing bugs."[1]

UNIVERSITY OF CAMBRIDGE Judge Business School

RogueWave SOFTWARE

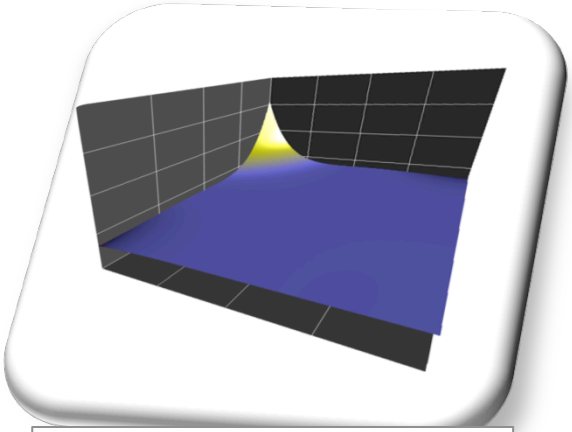
Irreproducible numerical result

--- Case study: “Monte Carlo Simulation” (MCB)

- CORAL proxy application
- MPI non-determinism

Table 1: Catalyst Specification

Nodes	304 batch nodes
CPU	2.4 GHz Intel Xeon E5-2695 v2 (24 cores in total)
Memory	128 GB
Interconnect	InfiniBand QDR (QLogic)
Local Storage	Intel SSD 910 Series (PCIe 2.0, MLC)



MCB: Monte Carlo Benchmark

Final numerical results are different between 1st and 2nd run

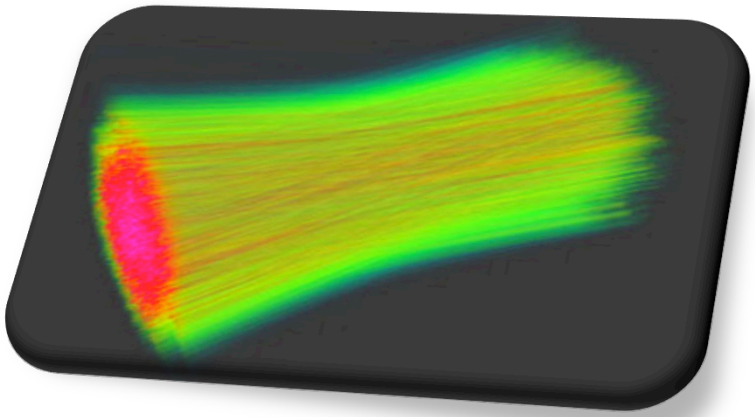
```
$ diff result_run1.out result_run2.out
result_run1.out:< IMC E_RR_total  -3.3140234409e-05  -8.3026937374e-08  2.9153322360e-08  -4.8198506756e-06  2.3113821822e-06
result_run2.out:> IMC E_RR_total  -3.3140234410e-05  -8.3026937376e-08  2.9153322360e-08  -4.8198506757e-06  2.3113821821e-06
```

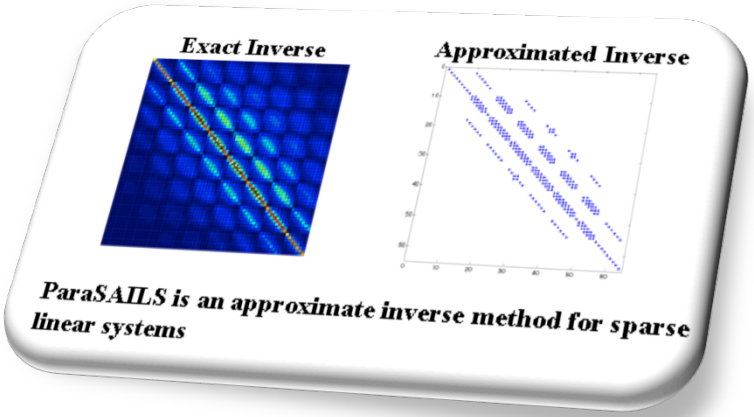
09e-05 10e-05 | 74e-08 76e-08 | 56e-06 57e-06 | 22e-06 21e-06

Irreproducible bugs

--- Case study: Pf3d and Diablo/Hypre 2.10.1

- Computational Scientists are suffering from irreproducible hangs





- **Pf3d** – hung only when scaling to half a million MPI processes
- Scientist refused to debug for 6 months …

- **Diablo** - hung in one of Hypre functions
- Scientist spent 2 months in the period of 18 months

Hypre is an MPI-based library for solving large, sparse linear systems of equations on massively parallel computers

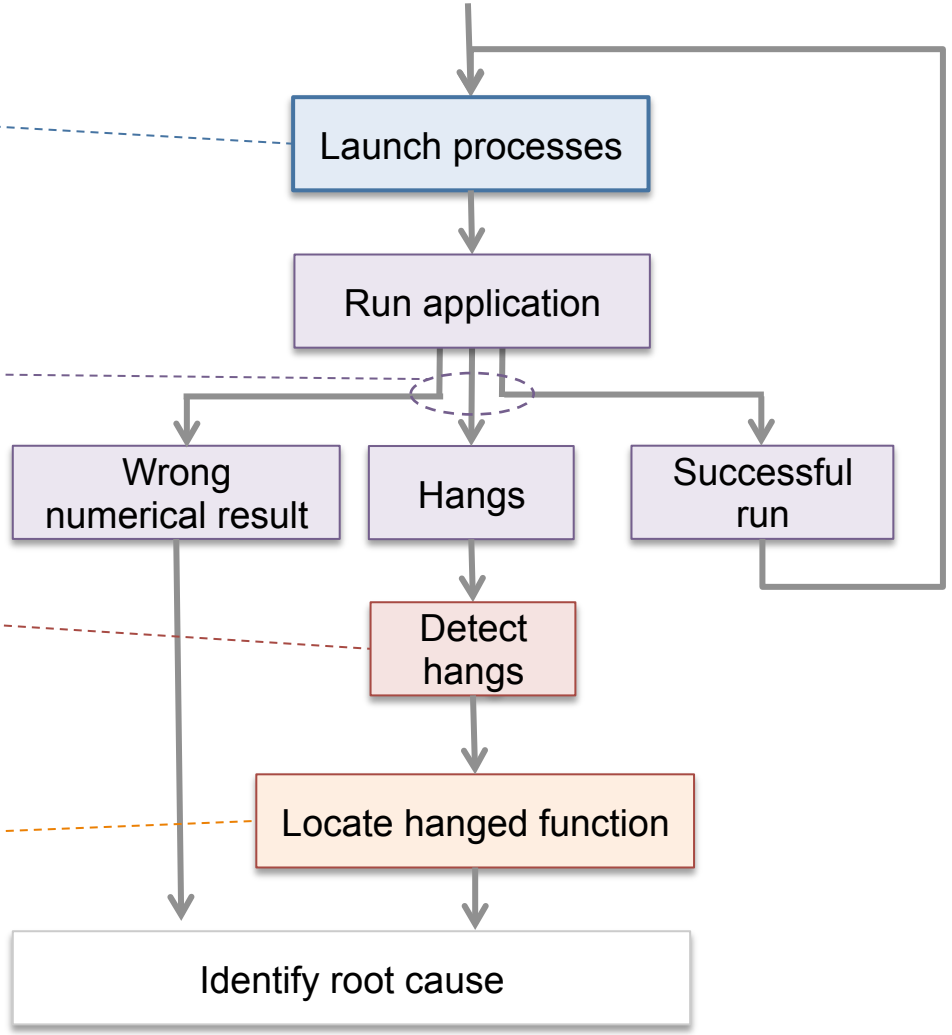
Typical debugging cycles for irreproducible bugs

Launching large application is expensive

MPI behaviors are irreproducible (Scientists have to execute multiple time)

Waste computational resources until allocated time/walltime ends

Analyzing every single of stack traces are not practical



Outline

- Irreproducibility in HPC
- Existing toolset for irreproducibility bugs
  - Spindle
  - ReMPI (MPI record-and-replay)
  - lo-watchdog
  - STAT
- Clock delta compression for ReMPI

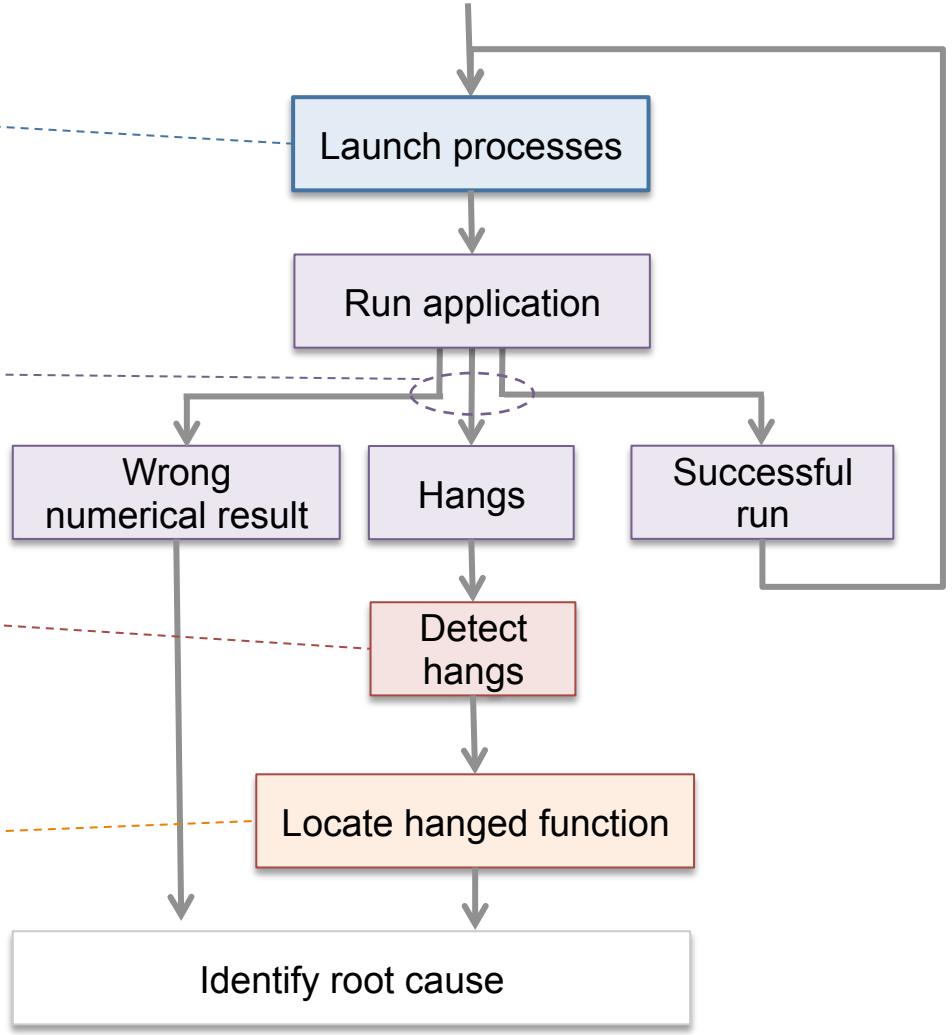
Typical debugging cycles for irreproducible bugs

Launching large application is expensive

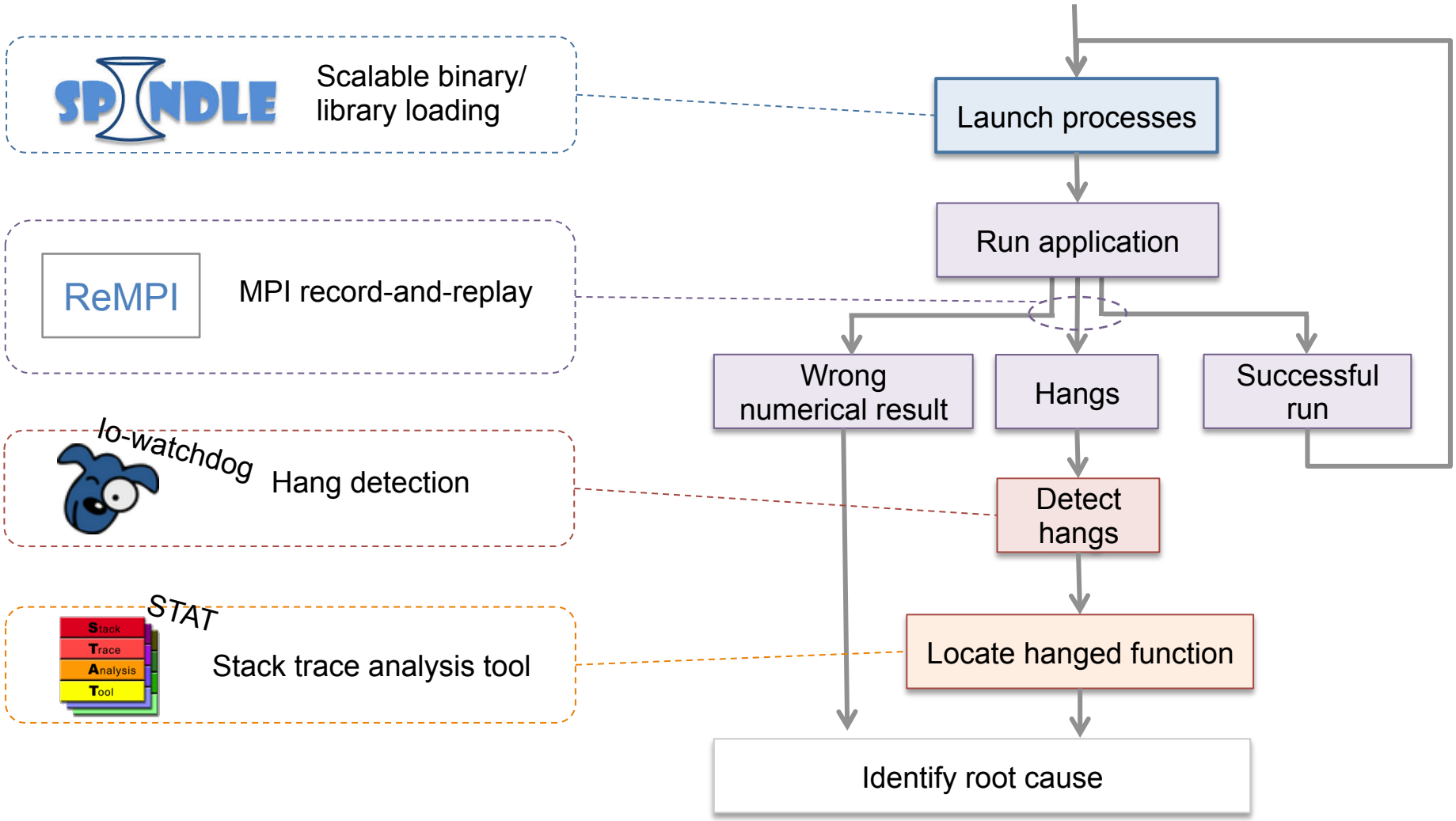
MPI behaviors are irreproducible (Scientists have to execute multiple time)

Waste computational resources until allocated time/walltime ends

Analyzing every single of one of stack traces are not practical



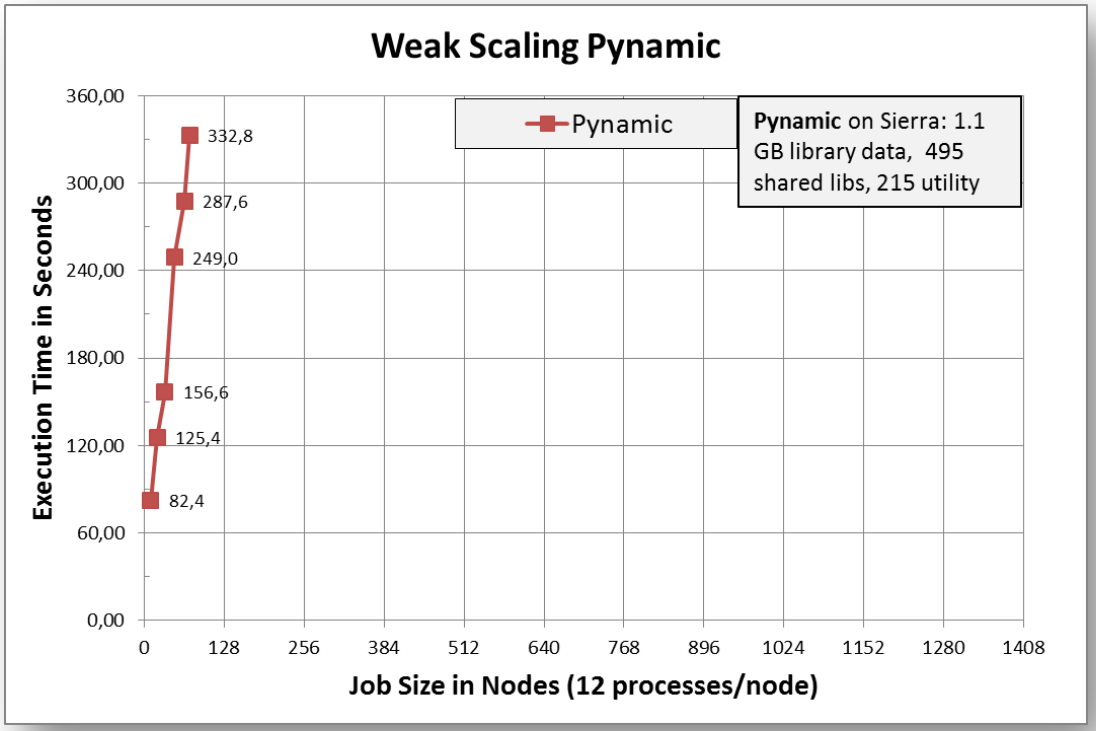
Useful toolset for irreproducible bugs



Dynamic Linking Causes Major Disruption at Scale

- **Multi-physics applications at LLNL**
  - 848 shared library files
  - Load time on BG/P:
    2k tasks → 1 hour
    16k tasks → 10 hours

- **Pynamic**
  - LLNL Benchmark
  - Loads shared libraries and python files
  - 495 shared objects → 1.1 GB



Pynamic running on LLNL Sierra Cluster
1944 nodes, 12 tasks/node, NFS

Challenges Arise from File Access Storms

SPINDLE

- Caused by dynamic linker
  - **searching** and
  - **loading** dynamic linked libraries
- Formulas:

File metadata operations:

# of tests = # of processes
x # of locations
x # of libraries

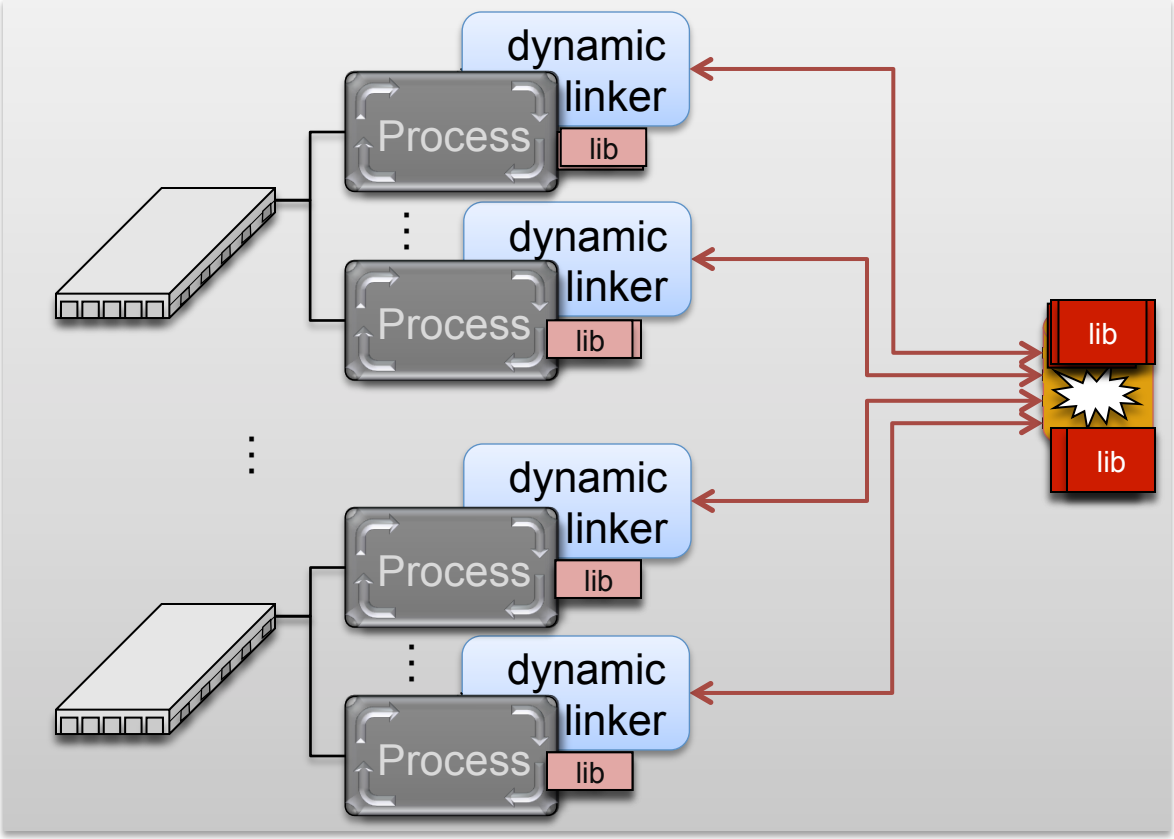
File read operations:

# of reads = # of processes
x # of libraries

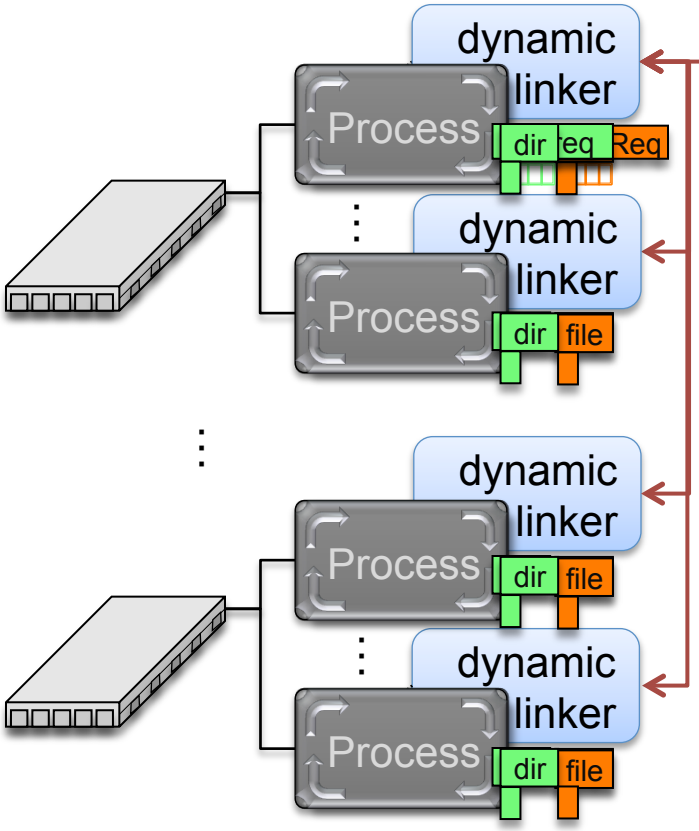
- Example: Pynamic Benchmark on Sierra-Cluster
  - serial (1 task): 5,671 open/stat calls
  - parallel (23,328 tasks) : 132,293,088 open/stat calls

File Access is Uncoordinated!

- Loading is nearly unchanged since 1964 (MULTICS)
- Each process independently read shared libraries
  - ld-linux.so uses serial POSIX file operations that are not coordinated among process.



How SPINDLE Works



Requesting dir/file:

✓✓1. Request from leader

✓✓2. Leader reads from disk

✓✓3. Leader distributes to peers

Scalable binominal tree broadcast
Cache metadata/libraries in ramdisk

File metadata operations:
# of tests = *# of locations*

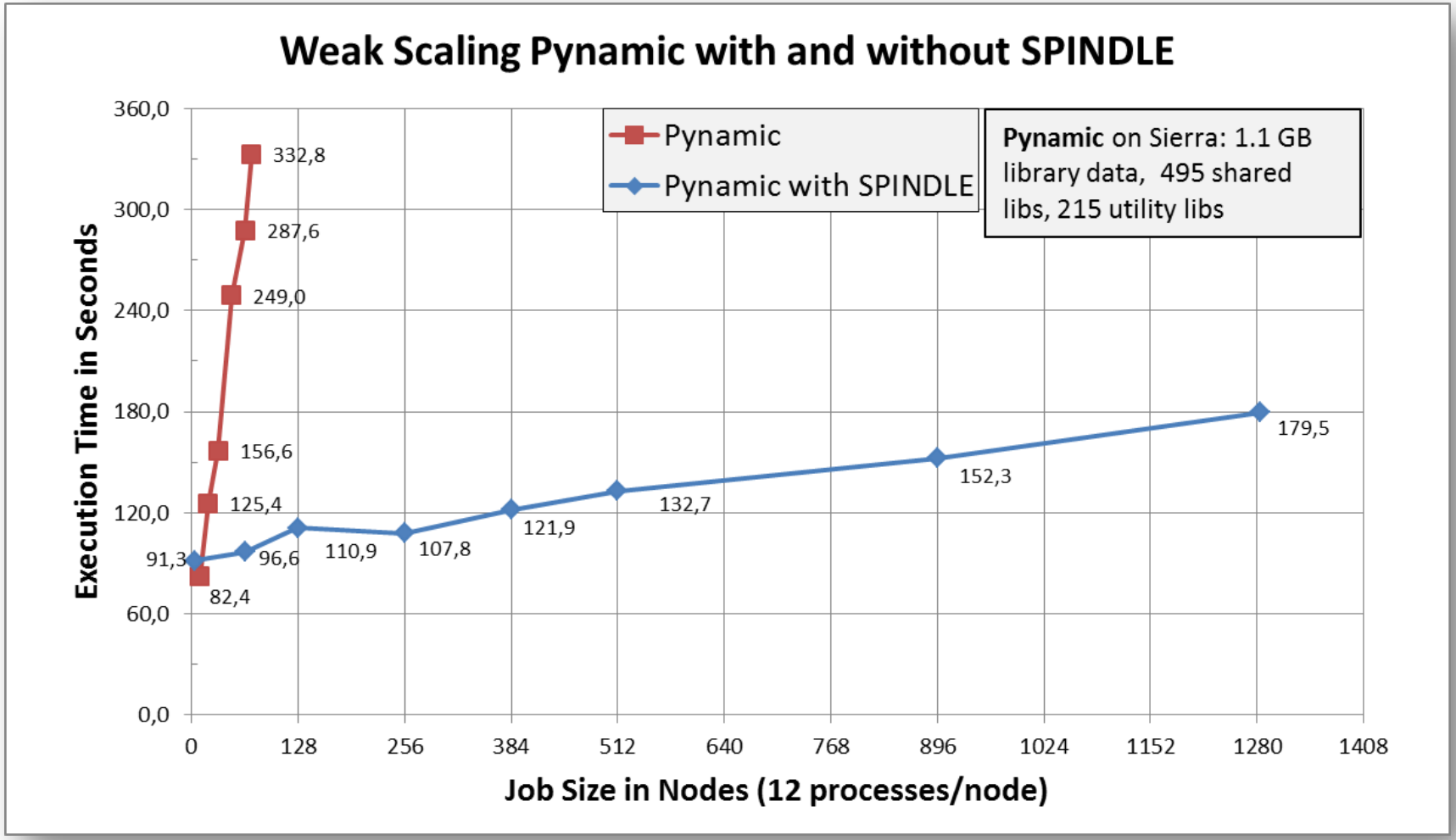
File read operations:
# of reads = *# of libraries*

SPINDLE: *Scalable Parallel Input Network for Dynamic Load Environments*

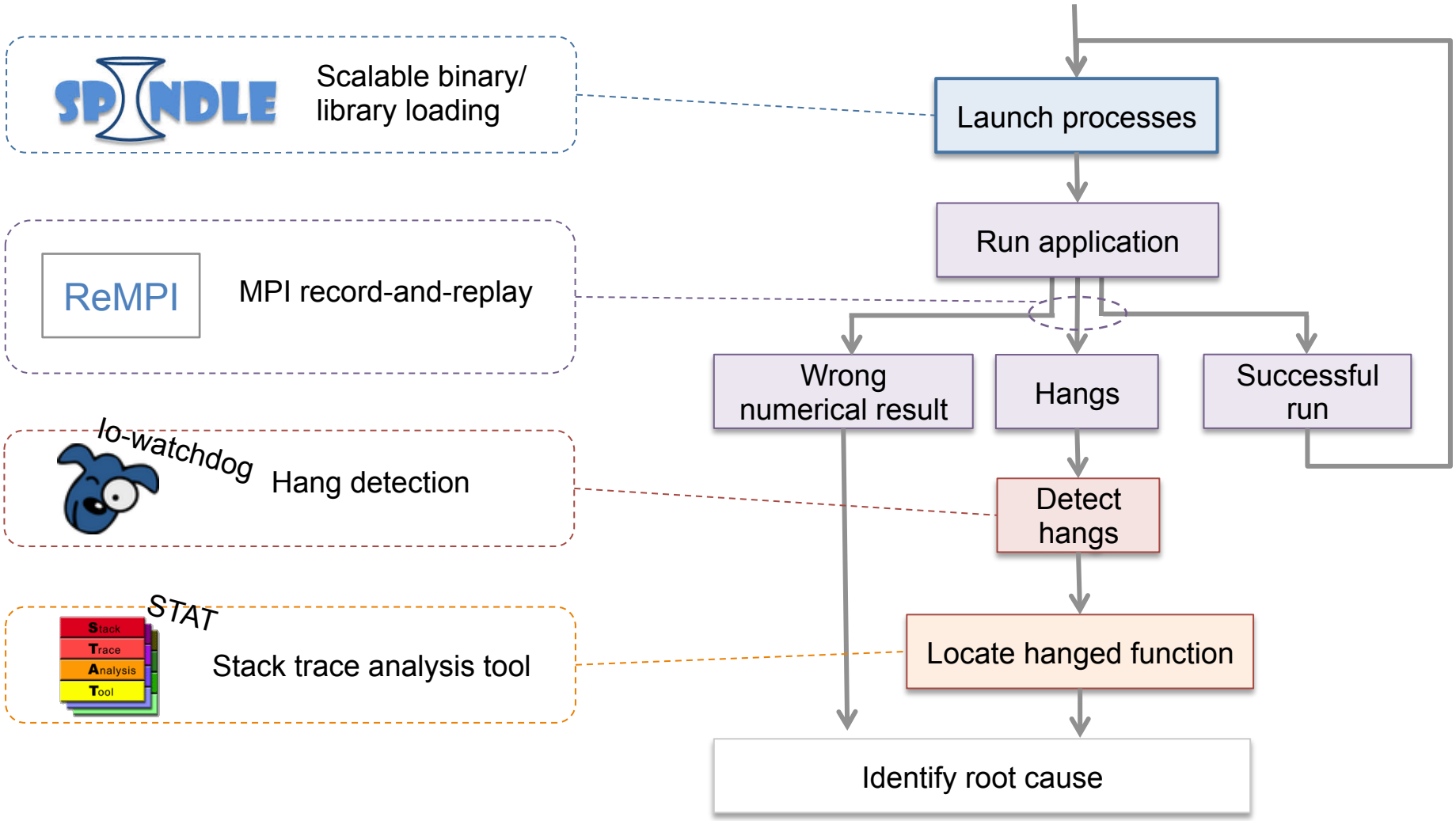
SPINDLE Intercepts Dynamic Linker Transparently

- rtld-audit interface of GNU linker
  - rtld-audit provides an auditing API that allows an application to be notified when various dynamic linking events occur
  - Redirect library loads to libraries read by leader
- Specify dynamic library with audit interface in `LD_AUDIT` environment variable
- Server caching
  - Stores metadata/libraries in Ramdisk
  - Server do not send the same requests to filesystems

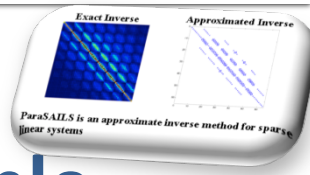
SPINDLE's Performance



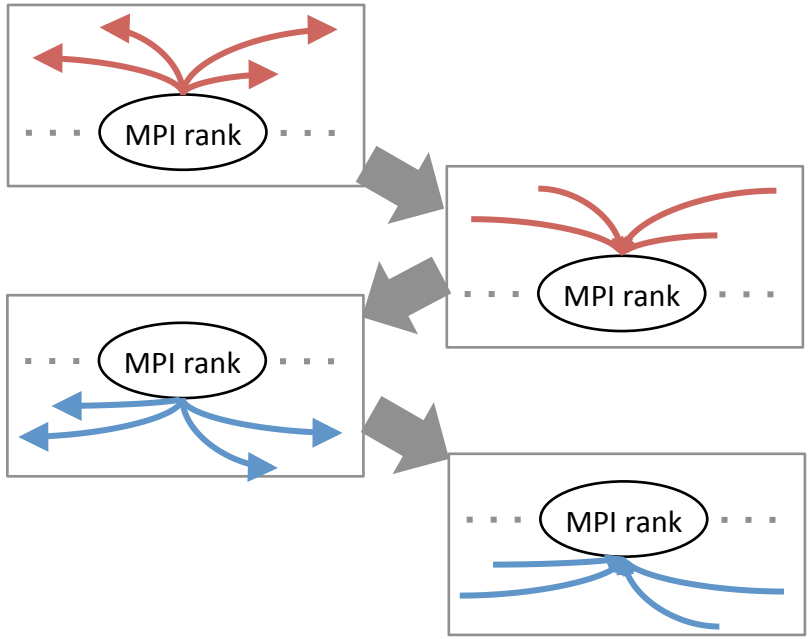
Useful toolset for irreproducible bugs



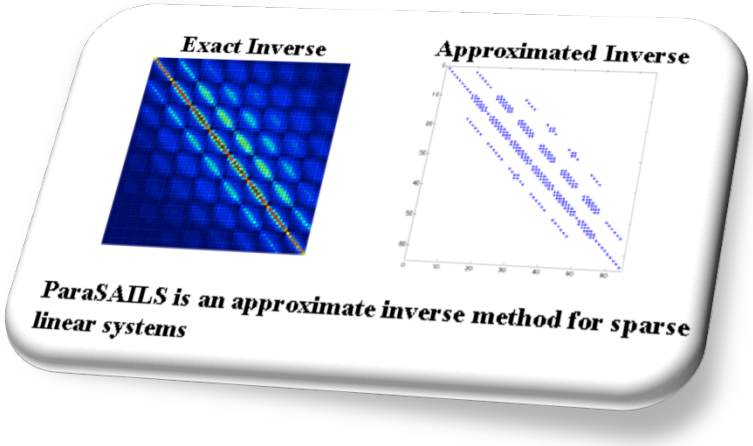
Difficult to find root cause of bugs --- HYPRE 2.10.1 example



- Func ParaSailsSetup (…)
  - Func ParaSailsSetupPattern (…)
    - MPI_Isend (tag = 222)
    - MPI_Allreduce (SUM)
    - MPI_Probe (ANY_SOURCE, tag=222)
    - MPI_Recv (tag = 222)
    - MPI_Isend (tag = 223)
    - MPI_Probe (ANY_SOURCE, tag = 223)
    - MPI_Recv (tag = 223)
  - Func ParaSailsSetupValues (…)
    - MPI_Isend (tag = 222)
    - MPI_Allreduce (SUM)
    - MPI_Probe (ANY_SOURCE, tag=222)
    - MPI_Recv (tag = 222)
    - MPI_Isend (tag = 223)
    - MPI_Probe (ANY_SOURCE, tag = 223)
    - MPI_Recv (tag = 223)



Irreproducible bug In HYPRE 2.10.1



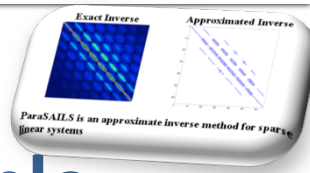
- **Diablo** - hung in one of Hypre functions

- Hangs are totally irreproducible
  - Hangs only a few times per 30 runs
- Hangs after running for a few hours
  - Checkpoint/Restart just before the hang does not work

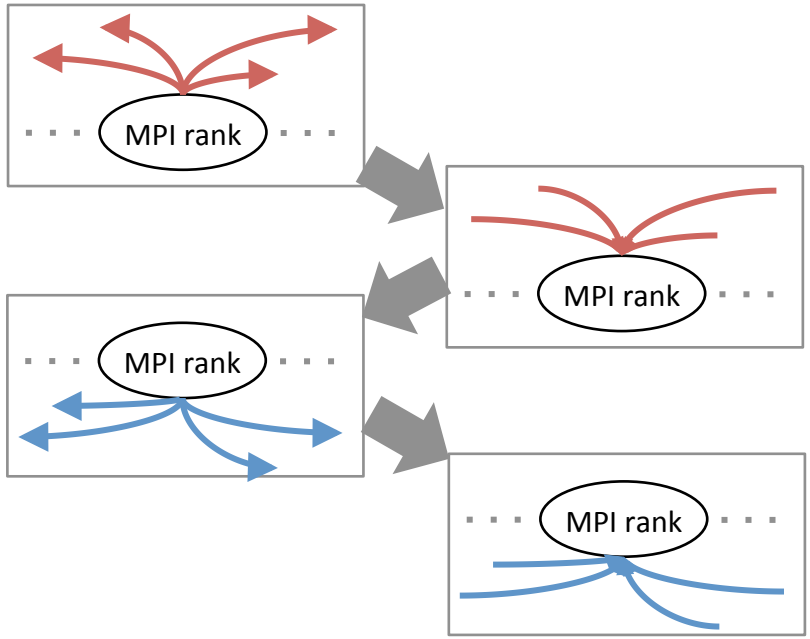
  ➔ We had to wait for a few hours more from the last checkpoint
- Hangs in particular parallelisms
  - 64 procs ➔ Hangs
  - Certain level of parallelism➔ Never hang

The scientists spent 2 months in the period of 18 months, then gave up debugging

Difficult to find root cause of bugs --- HYPRE 2.10.1 example

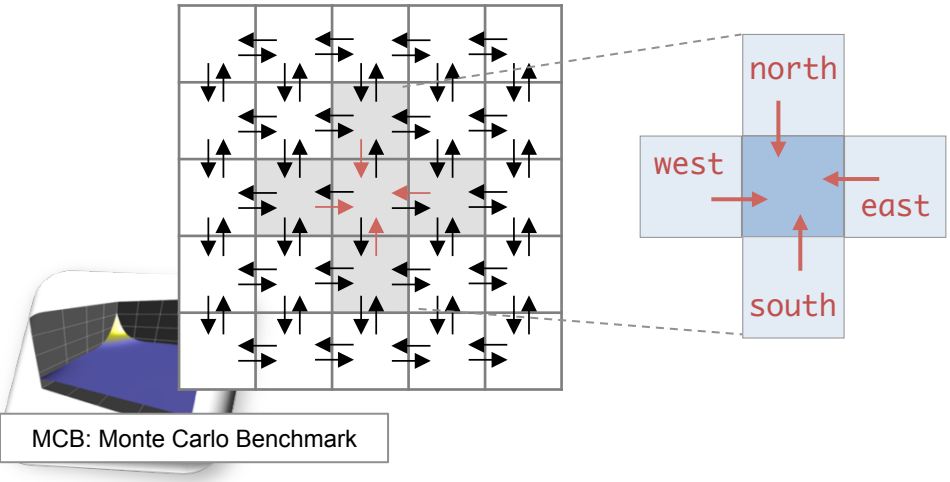


- Func ParaSailsSetup (…)
  - Func ParaSailsSetupPattern (…)
    - MPI_Isend (tag = 222)
    - MPI_Allreduce (SUM)
    - MPI_Probe (ANY_SOURCE, tag=222)
    - MPI_Recv (tag = 222)
    - MPI_Isend (tag = 223)
    - MPI_Probe (ANY_SOURCE, tag = 223)
    - MPI_Recv (tag = 223)
  - Func ParaSailsSetupValues (…)
    - MPI_Isend (tag = 222)
    - MPI_Allreduce (SUM)
    - MPI_Probe (ANY_SOURCE, tag=222)
    - MPI_Recv (tag = 222)
    - MPI_Isend (tag = 223)
    - MPI_Probe (ANY_SOURCE, tag = 223)
    - MPI_Recv (tag = 223)



Why MPI non-determinism occurs ?

- In such non-deterministic applications, each process doesn't know which rank will send message
  - e.g.) Particle simulation
- Messages can arrive in any order from neighbors ➔ inconsistent message arrivals



MCB: Monte Carlo Benchmark

Typical MPI non-deterministic code

```
MPI_Irecv(…, MPI_ANY_SOURCE, …);
while(1) {
  MPI_Test(flag);
  if (flag) {
     <computation>
     MPI_Irecv(…, MPI_ANY_SOURCE, …);
  }
}
```

Source of MPI non-determinism

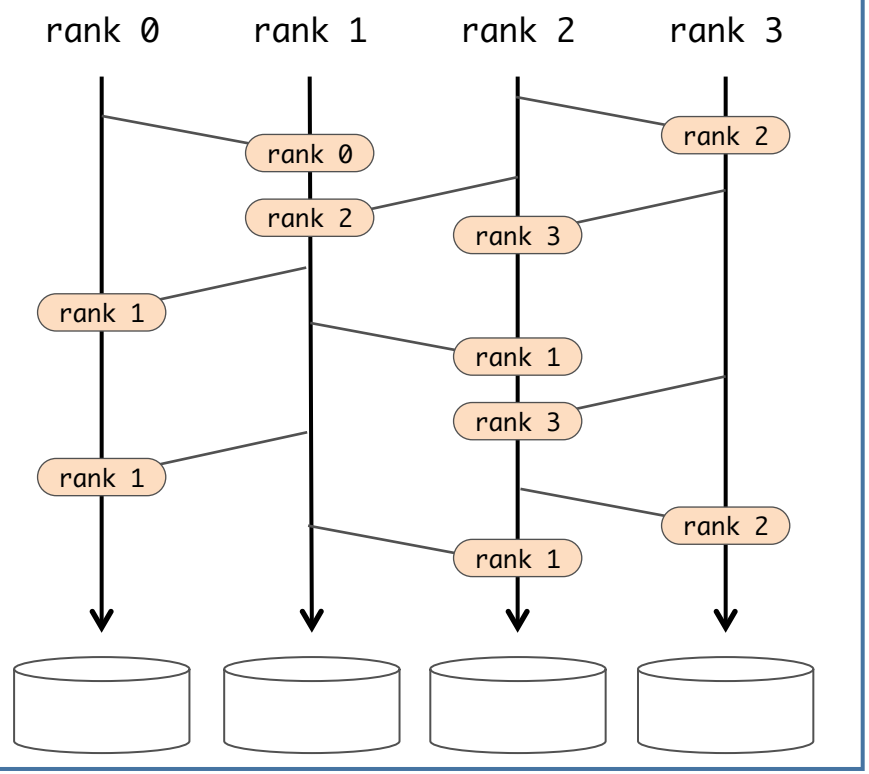
MPI matching functions

	Wait family	Test family
single	MPI_Wait	MPI_Test
any	MPI_Waitany	MPI_Testany
some	MPI_Waitsome	MPI_Testsome
all	MPI_Waitall	MPI_Testall

ReMPI can reproduce message matching

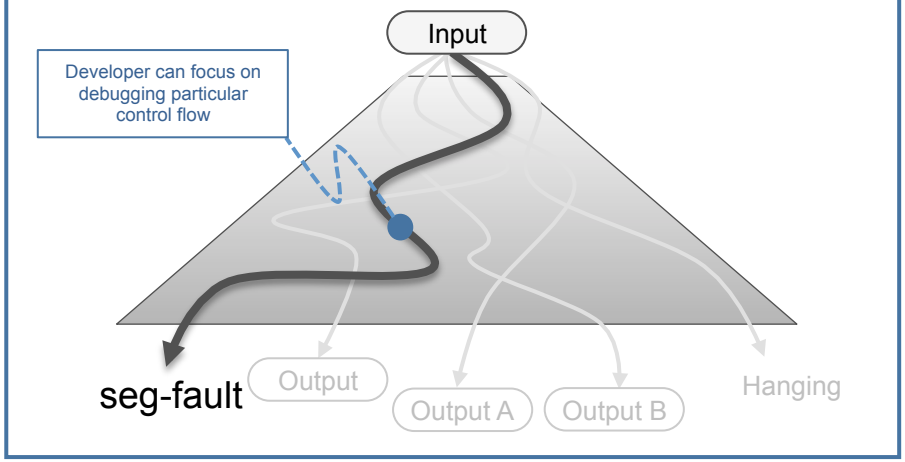
- ReMPI can reproduce message matching by using record-and-replay technique

Record-and-replay



- Traces, records message receive orders in a run, and replays the orders in successive runs for debugging
  - Record-and-replay can reproduce a target control flow
  - Developers can focus on debugging a particular control flow

Debugging a particular control flow in replay



ReMPI: Implementation

- PMPI wrapper for record and replay

Record mode

```
MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
{
    int ret;
    ret = PMPI_Test(…, flag, status);
    <Write record “flag” and “status.MPI_SOURCE” >
}
```

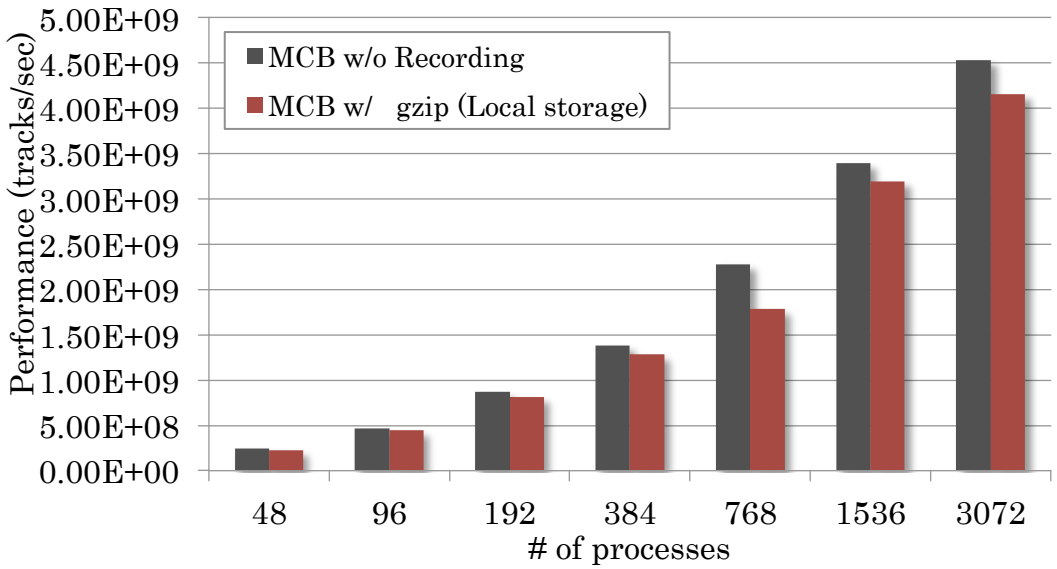
Replay mode

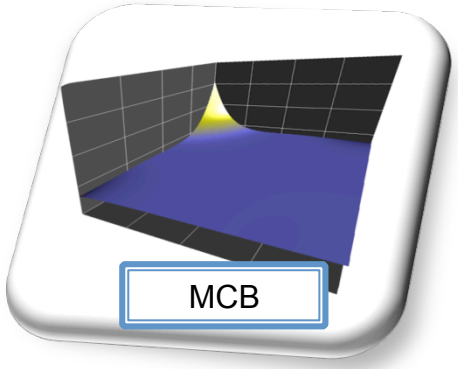
```
MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
{
    int ret;
    <Read record “flag” and “status.MPI_SOURCE” >
    ret = PMPI_Probe(…);
    ret = PMPI_Recv(…);
}
```

Record overhead to performance

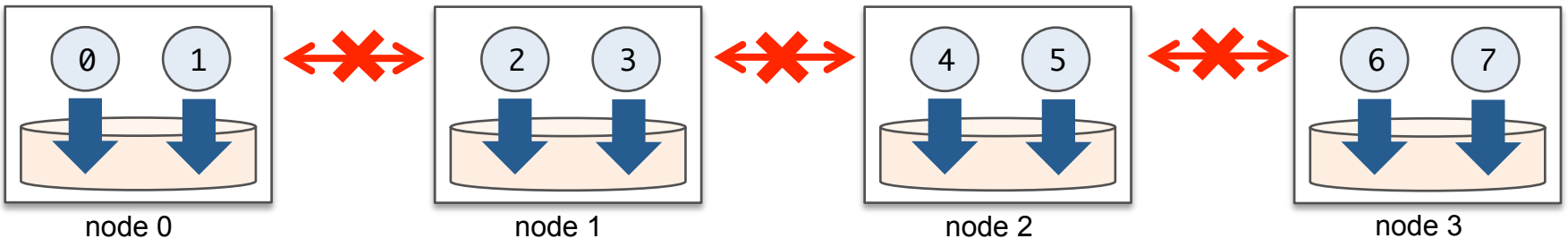
ReMPI

- Performance metric: how may particles are tracked per second





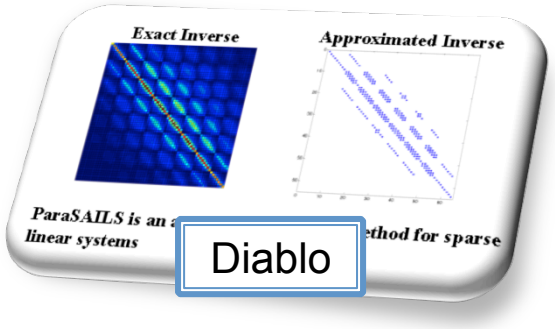
- ReMPI becomes scalable by recording to local memory/storage
  - Each rank independently write record → No communication across MPI ranks



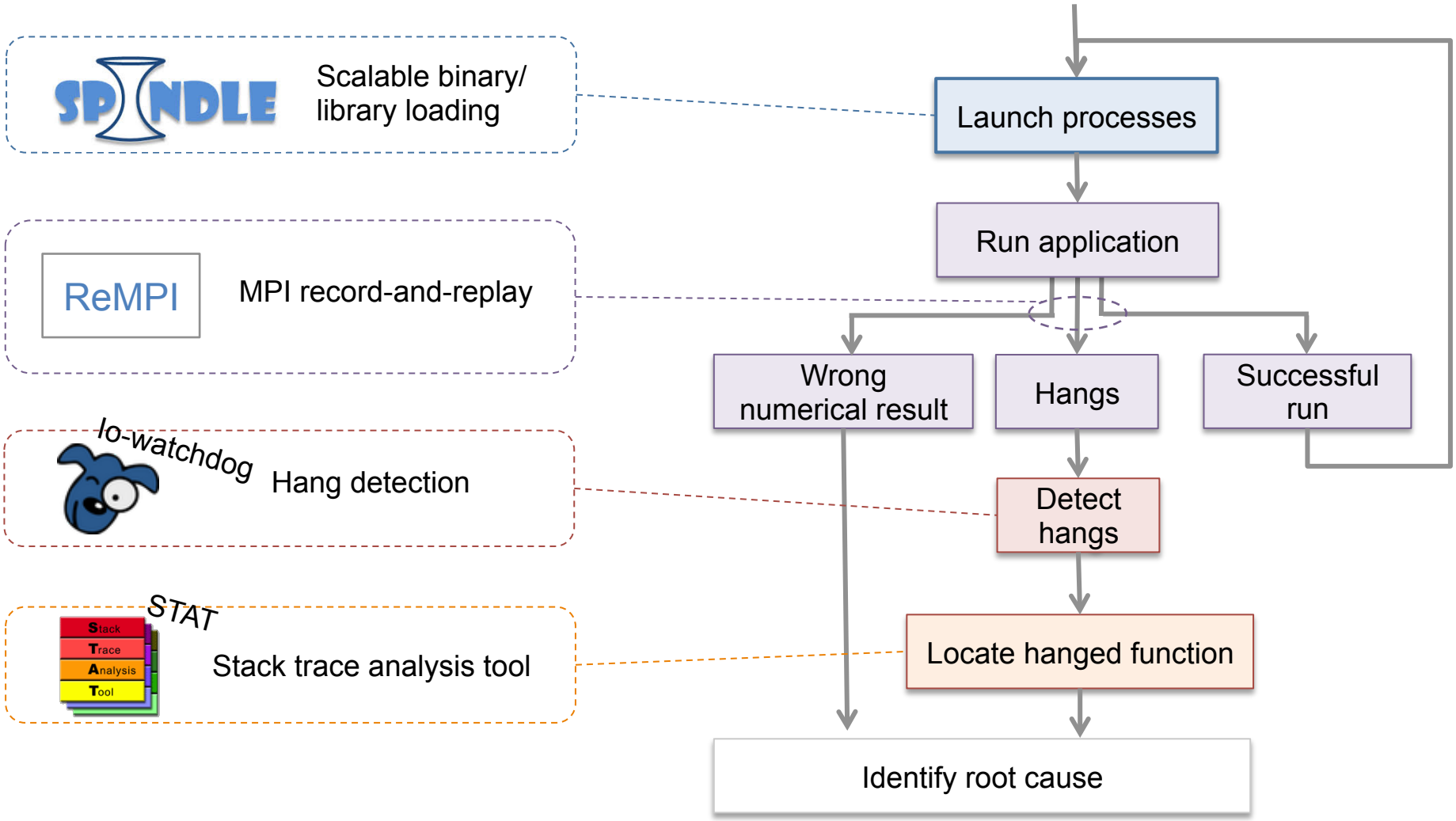
ReMPI captures the irreproducible bug in Diablo

- Diablo originalyl hangs 4 – 6 times out of 30 runs
- Even with ReMPI, we can still record hanging MPI behavirs

	# of hangs out of 30 runs
w/o ReMPI	4-6 times
w/ ReMPI	3 times



Useful toolset for irreproducible bugs



IO-Watchdog: detecting applications' hangs

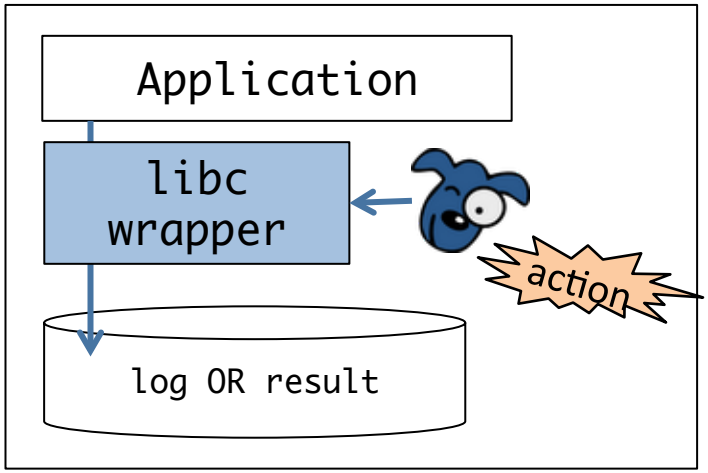
- IO-Watchdog detect applications' hangs
- When detected, io-watchdog triggers a set of user-defined action (script)
- Actions
  - Killing procceses
  - Email to the usr
  - running debugger (e.g. STAT)

```
search /dir/path/to/actions
timeout = 20m
actions = STAT, kill
```

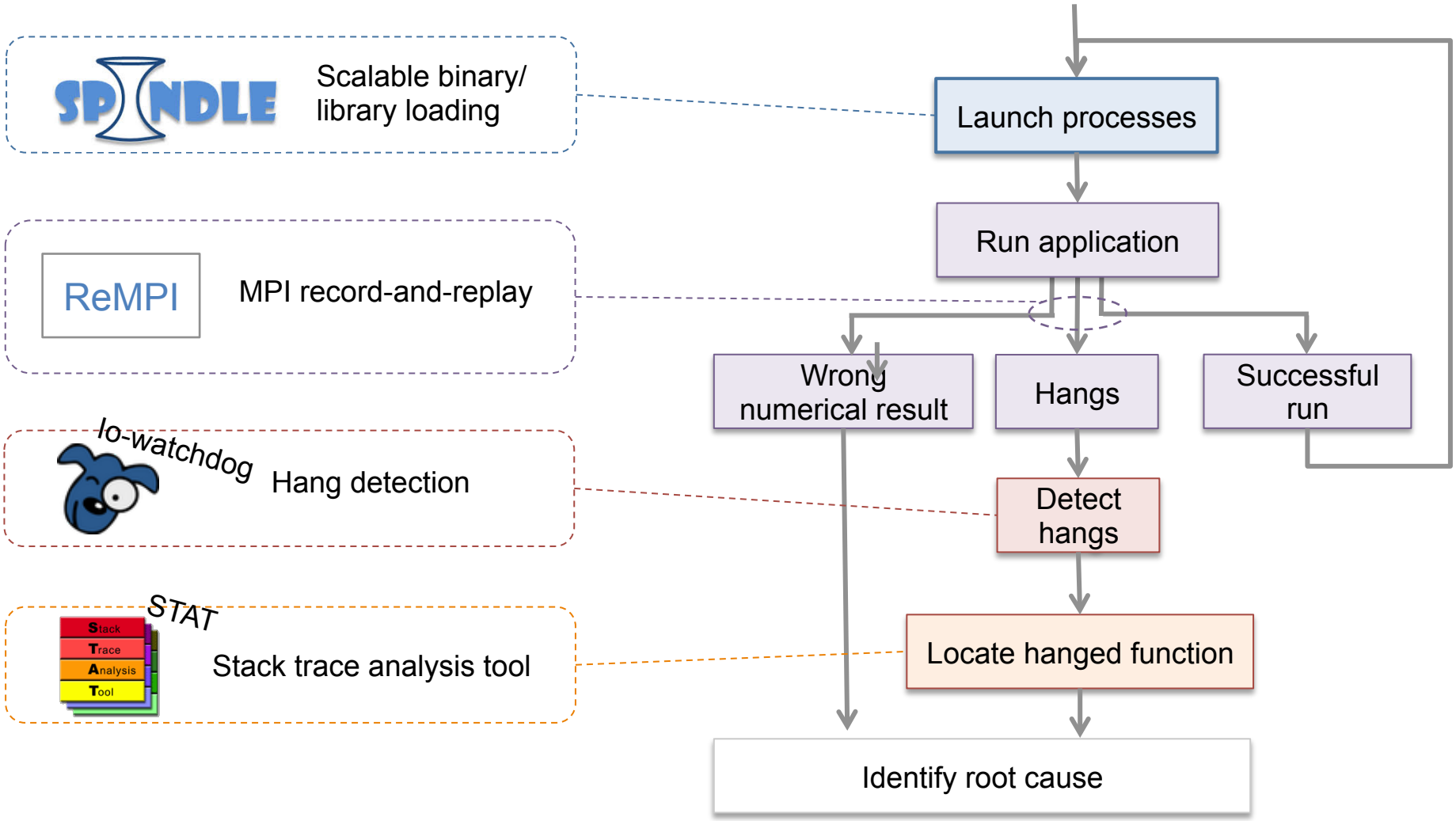
IO-Watchdog watches IOs

- Typical applications periodically write something to a log or data file

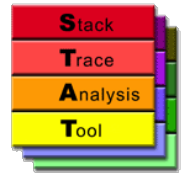
- Monitor I/O via libc wrapper
  1. Preload io-watchdog wrapper via LD_PRELOAD
  2. Monitor all write/output-related calls in libc from an application
  3. Watchdog daemon periodically wakes up
  4. Ensures that the application has written something during the last timeout period
  5. Triggers actions



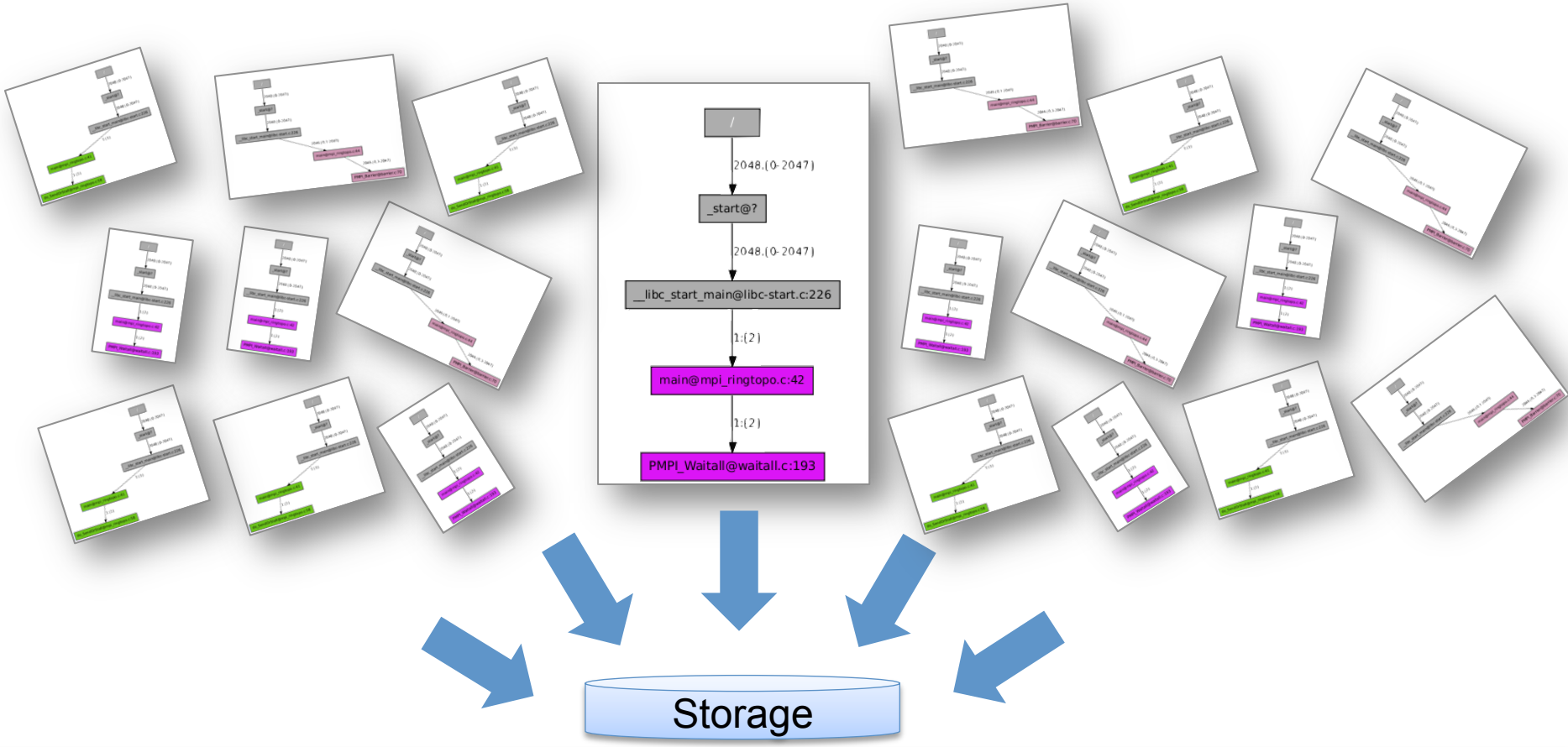
Useful toolset for irreproducible bugs



STAT: Stack Trace Analysis tool



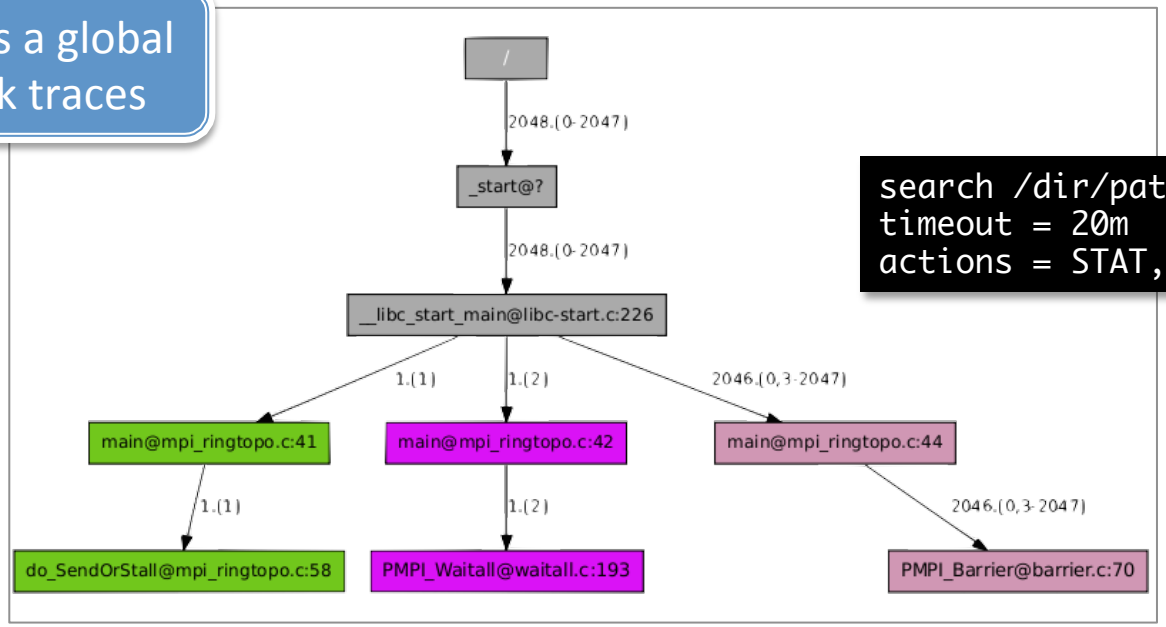
- Stack traces are useful to find hanging location
- But, … at large scale
  - Analyzing all traces is not practical
  - Producing a large mount of trace data



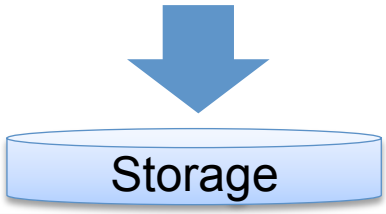
STAT: Stack Trace Analysis tool

- Stack traces are useful to find hanging location
- But, … at large scale
  - Analyzing all traces is not practical
  - Producing a large mount of trace data

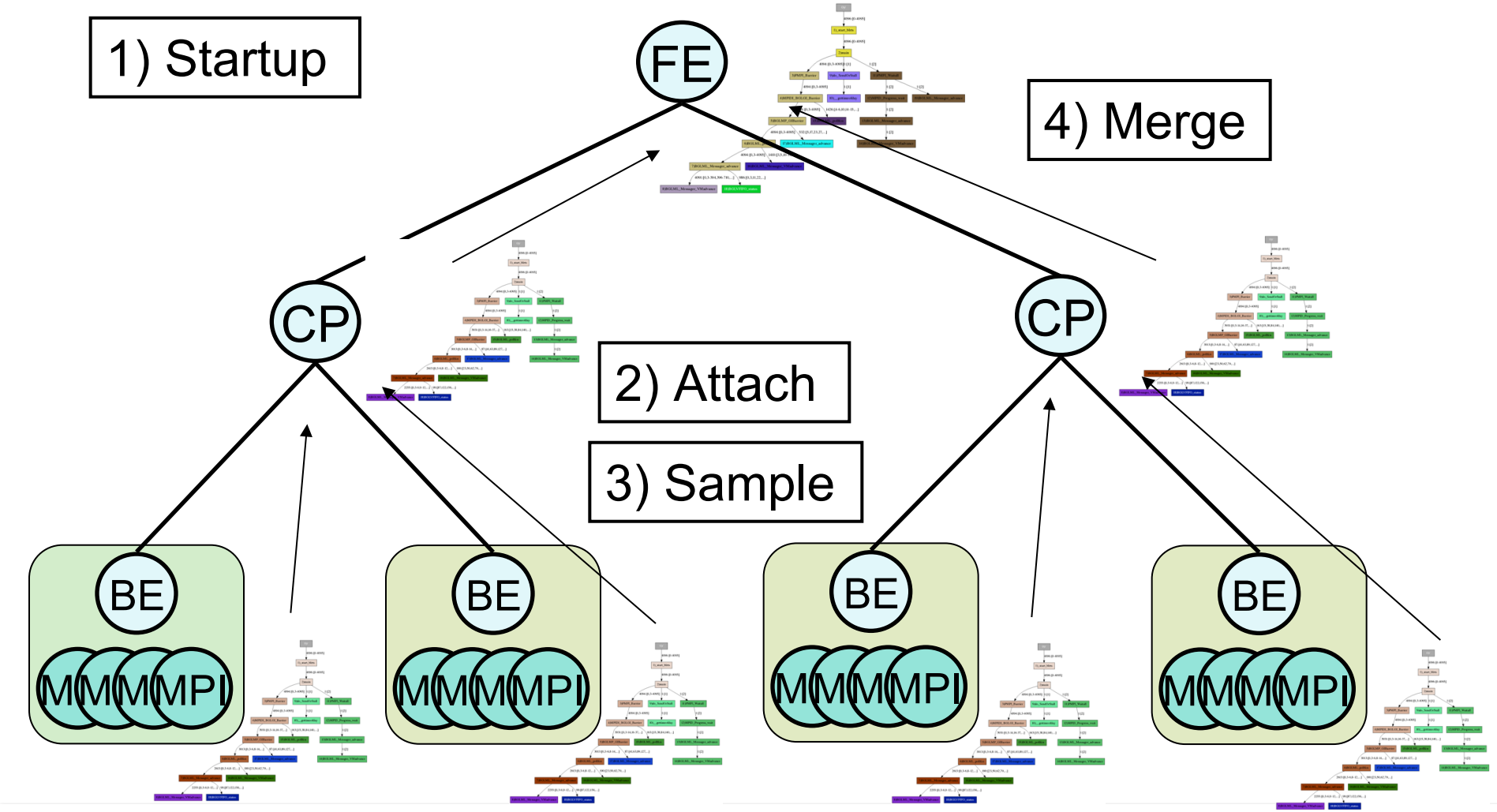
STAT provides a global view of stack traces




```
search /dir/path/to/actions
timeout = 20m
actions = STAT, kill
```

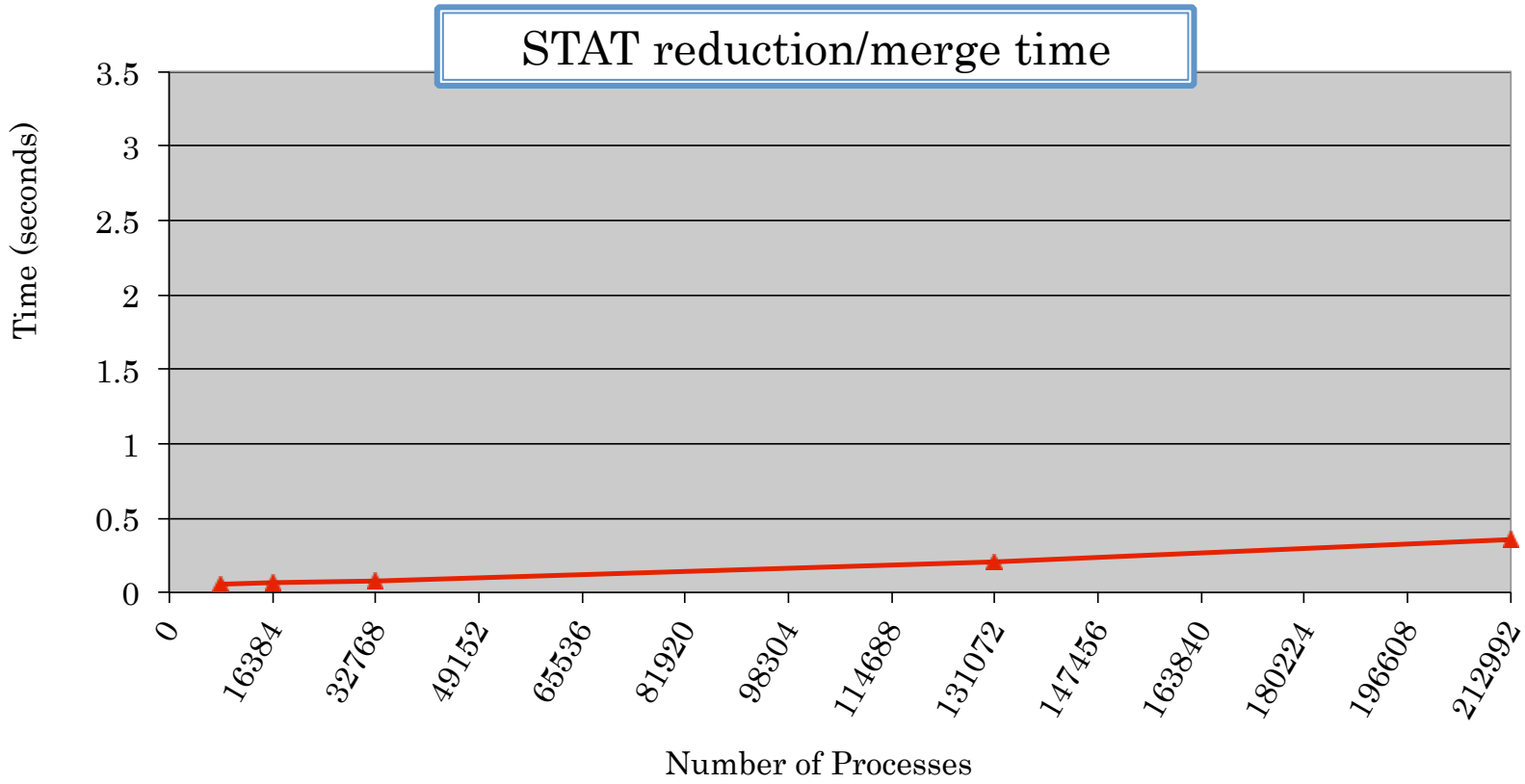


STAT provides scalable engine supporting this model



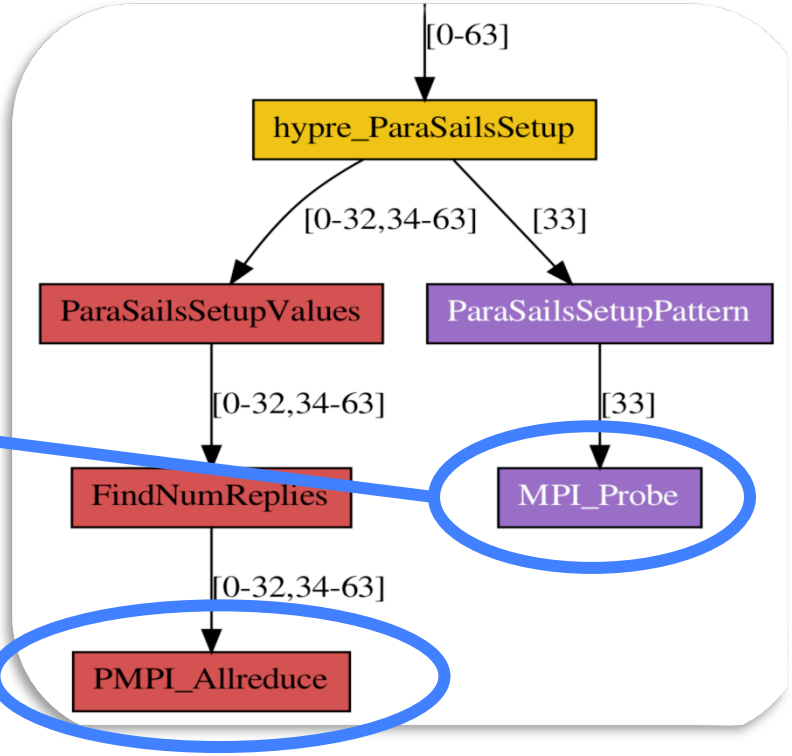
STAT is lightweight as well as scalable

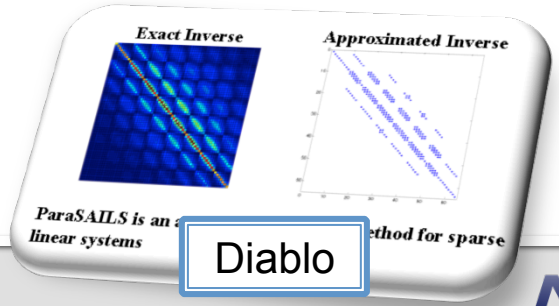
- Simple MPI benchmark (MPI ring topology communication)
- BG/L: 16K to 212K procs



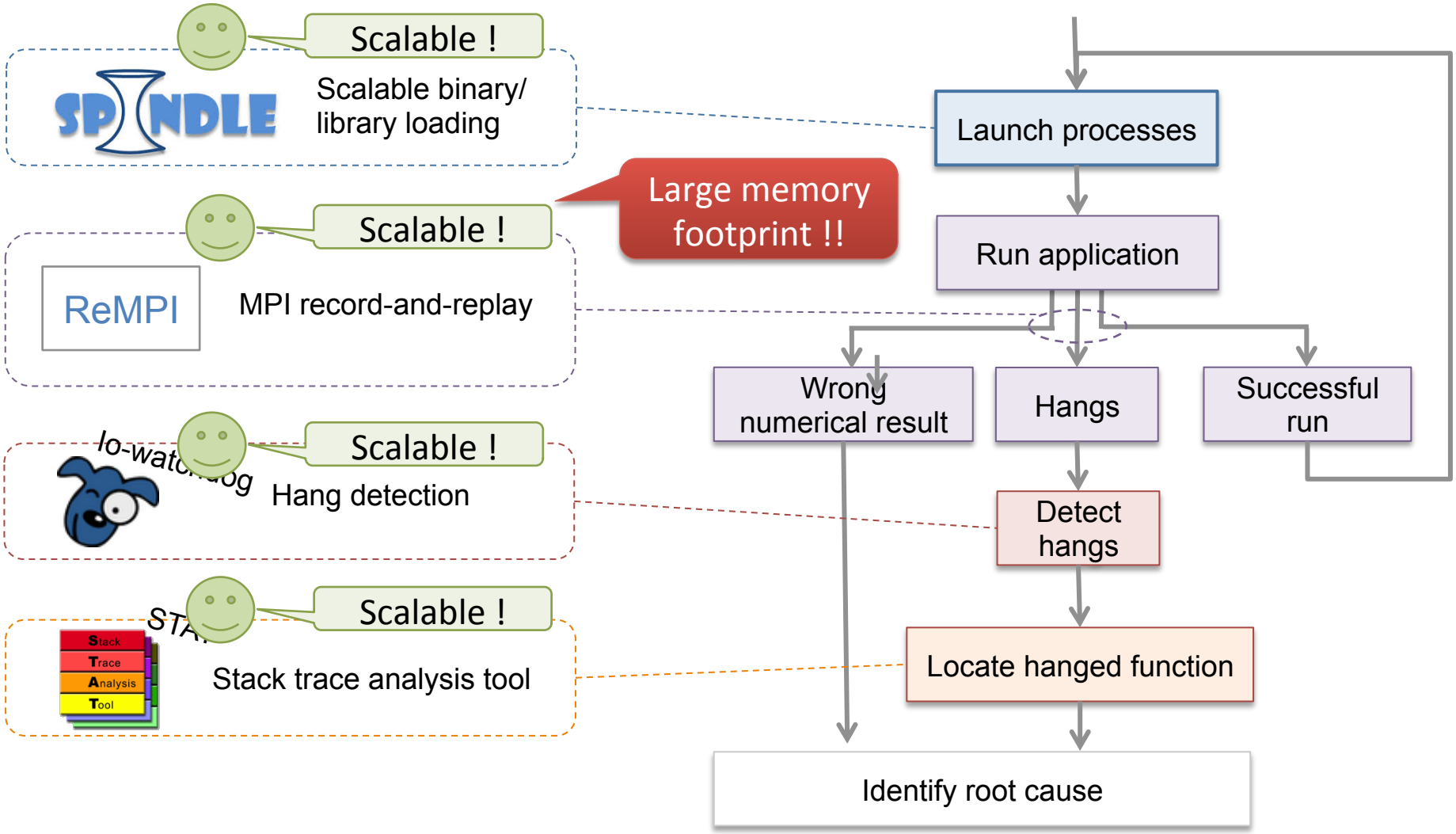
STAT manifests Diablo/HYPRE hangs

- Func ParaSailsSetup (…)
  - Func ParaSailsSetupPattern (…)
    - MPI_Isend (tag = 222)
    - MPI_Allreduce (SUM)
    - MPI_Probe (ANY_SOURCE, tag=222)
    - MPI_Recv (tag = 222)
    - MPI_Isend (tag = 223)
    - MPI_Probe (ANY_SOURCE, tag = 223)
    - MPI_Recv (tag = 223)
  - Func ParaSailsSetupValues (…)
    - MPI_Isend (tag = 222)
    - MPI_Allreduce (SUM)
    - MPI_Probe (ANY_SOURCE, tag=222)
    - MPI_Recv (tag = 222)
    - MPI_Isend (tag = 223)
    - MPI_Probe (ANY_SOURCE, tag = 223)
    - MPI_Recv (tag = 223)





Scalable toolset for irreproducible bugs

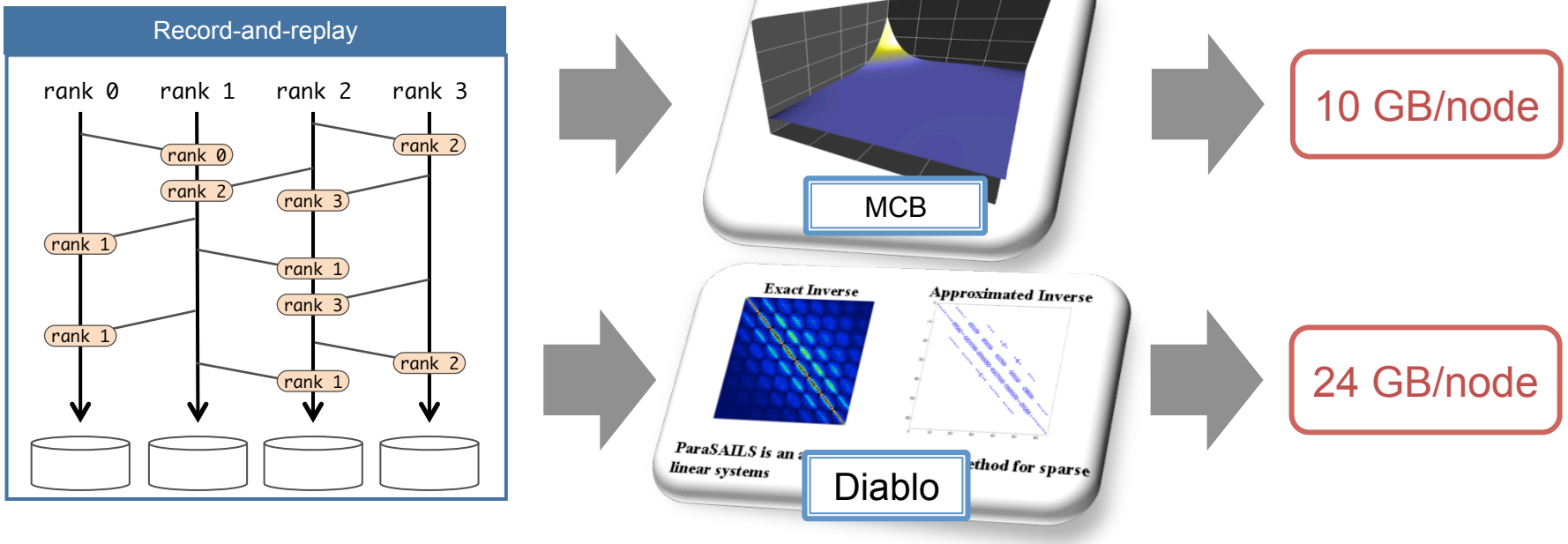


Outline

- Irreproducibility in HPC
- Existing toolset for irreproducibility bugs
  - Spindle
  - ReMPI (MPI record-and-replay)
  - lo-watchdog
  - STAT
- Clock delta compression for ReMPI

Record-and-replay won't work at scale

- Record-and-replay produces large amount of recording data
  - Over "10 GB/node" per day in MCB
  - Over "24 GB/node" per day in Diablo
- For scalable record-replay with low overhead, the record data must fit into local memory, but capacity is limited
  - Storing in shared/parallel file system is not scalable approach
  - Not necessary that the systems have fast local storage



Challenges Record size reduction for scalable record-replay

Overview of Clock Delta Compression (CDC)

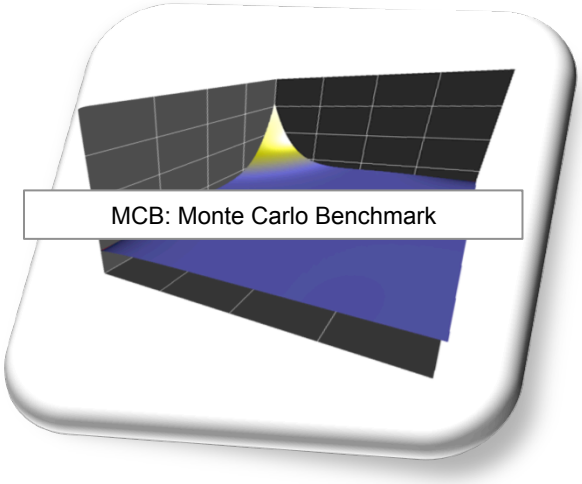
- Putting logical-clock (Lamport clock) into each MPI message

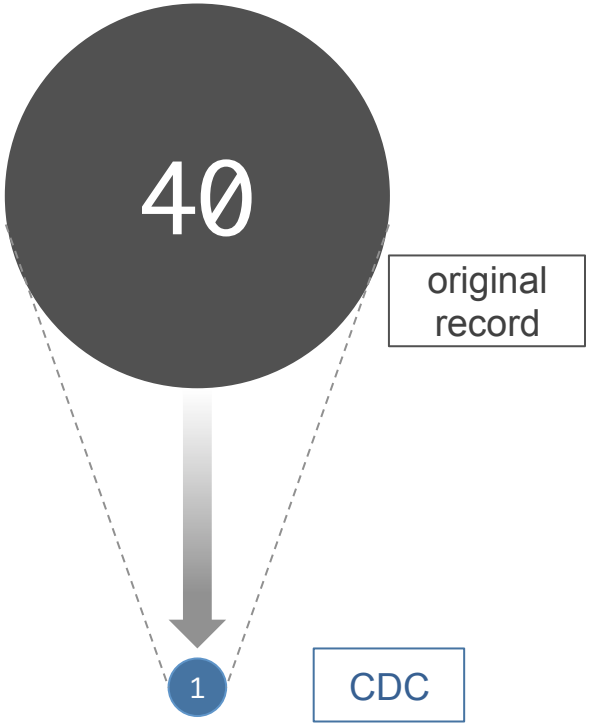
- Actual message receive orders (i.e. wall-clock orders) are very similar to logical clock orders in each MPI rank
  - MPI messages are received in almost monotonically increasing logical-clock order

- CDC records only the order differences between the wall-clock order and the logical-clock order without recording the entire message order

Result in MCB

- 40 times smaller than the one w/o compression





How to record-and-replay MPI applications ?

- Source of MPI non-determinism is these matching functions
  - "Replaying these matching functions' behavior" ➔ "Replaying MPI application's behavior"

Matching functions in MPI

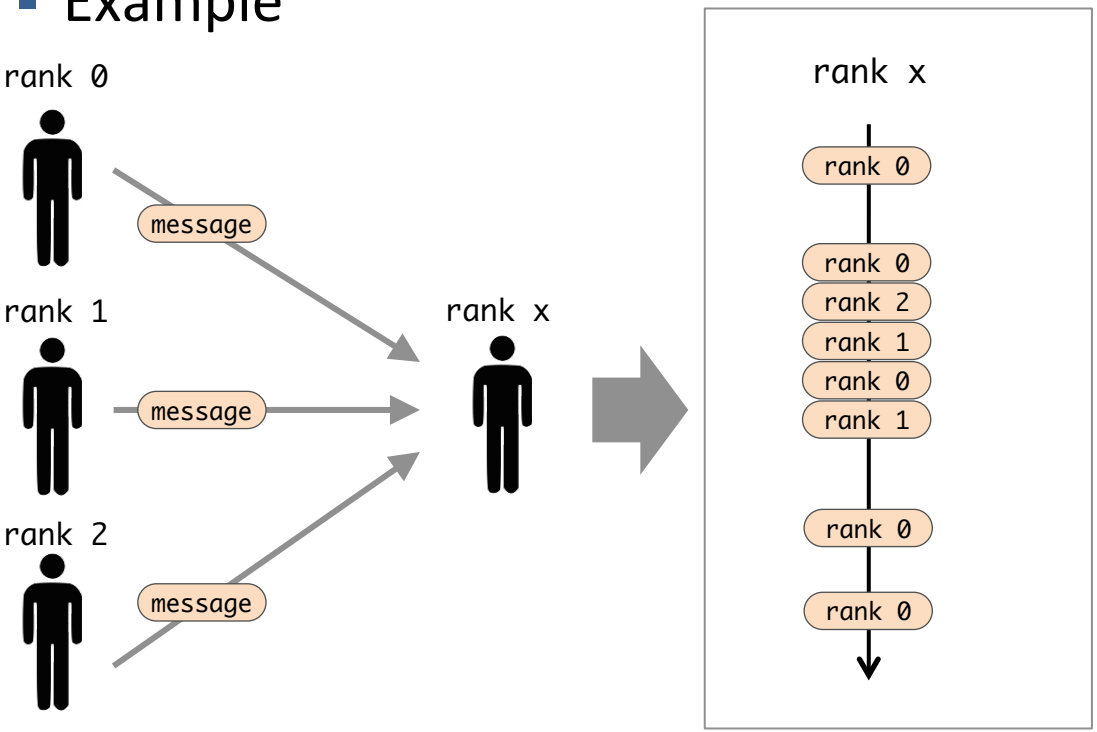
	Wait family	Test family
single	MPI_Wait	MPI_Test
any	MPI_Waitany	MPI_Testany
some	MPI_Waitsome	MPI_Testsome
all	MPI_Waitall	MPI_Testall

Source of MPI non-determinism

Questions What information need to be recorded for replaying these matching functions ?

Necessary values to be recorded for correct replay

- Example

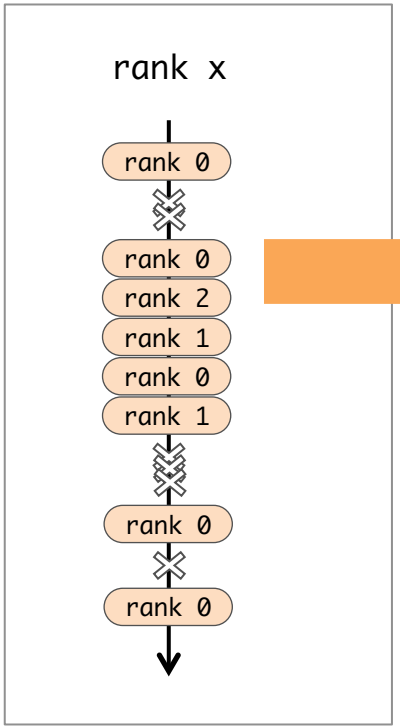


Necessary values for correct replay

Matching functions in MPI

	Wait family	Test family
single	MPI_Wait	MPI_Test
any	MPI_Waitany	MPI_Testany
some	MPI_Waitsome	MPI_Testsome
all	MPI_Waitall	MPI_Testall

- rank
  - Who send the messages?
- count & flag
  - For MPI_Test family
    - flag: Matched or unmatched ?
    - count: How many time unmatched ?
- id
  - For application-level out-of-order
- with_next
  - For matching some/all functions



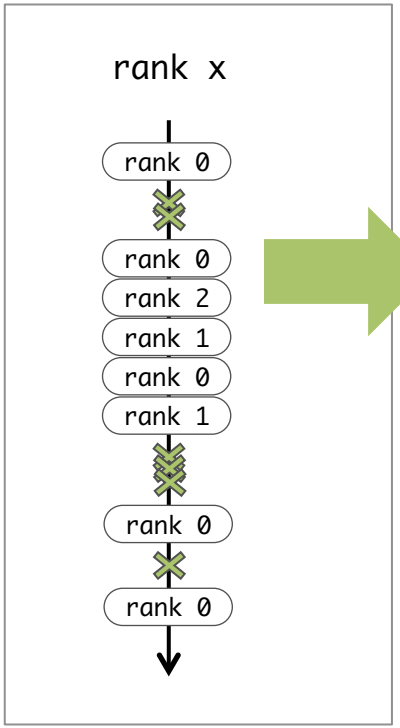
count	flag	rank
--	1	0
2	0	--
		0
		2
--	1	1
--	1	0
--	1	1
3	0	--
--	1	0
--	0	--
1	1	0

Necessary values for correct replay

Matching functions in MPI

	Wait family	Test family
single	MPI_Wait	MPI_Test
any	MPI_Waitany	MPI_Testany
some	MPI_Waitsome	MPI_Testsome
all	MPI_Waitall	MPI_Testall

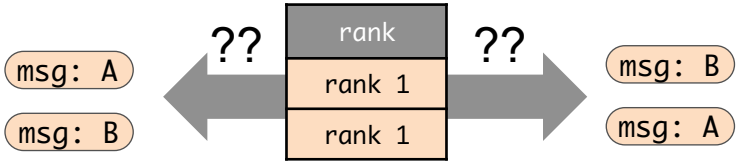
- rank
  - Who send the messages?
- count & flag
  - For MPI_Test family
    - flag: Matched or unmatched ?
    - count: How many time unmatched ?
- id
  - For application-level out-of-order
- with_next
  - For matching some/all functions

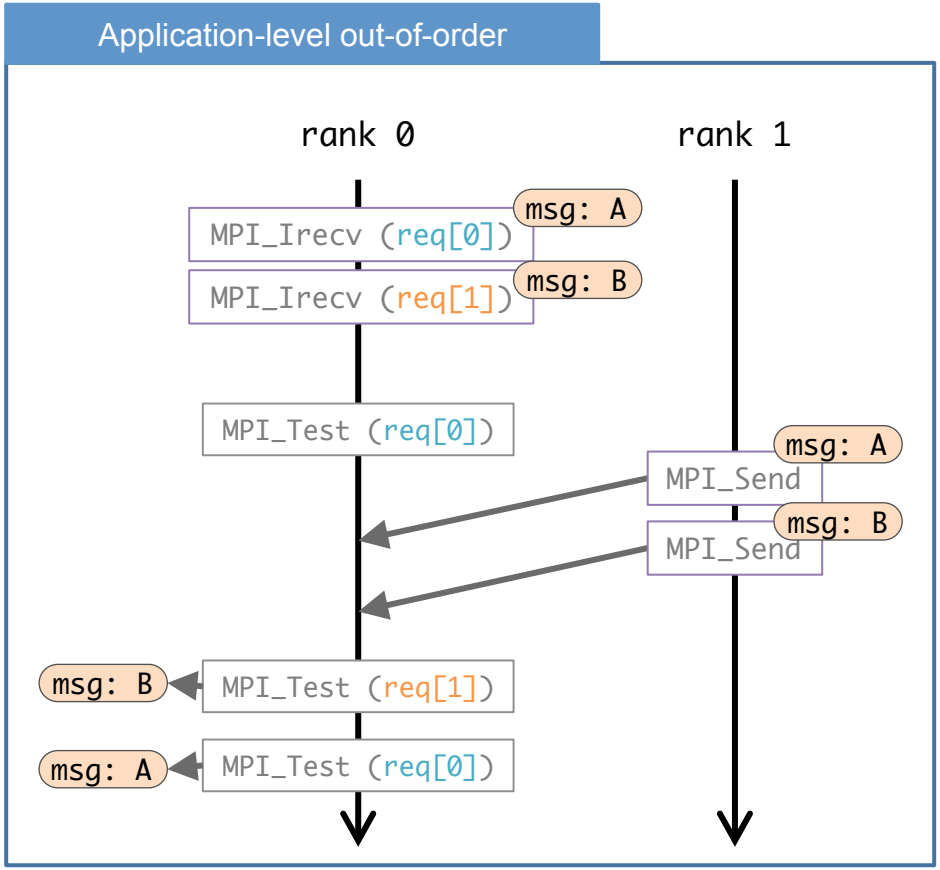


count	flag	rank
--	1	0
2	0	--
--	1	0
--	1	2
--	1	1
--	1	0
--	1	1
3	0	--
--	1	0
--	0	--
1	1	0

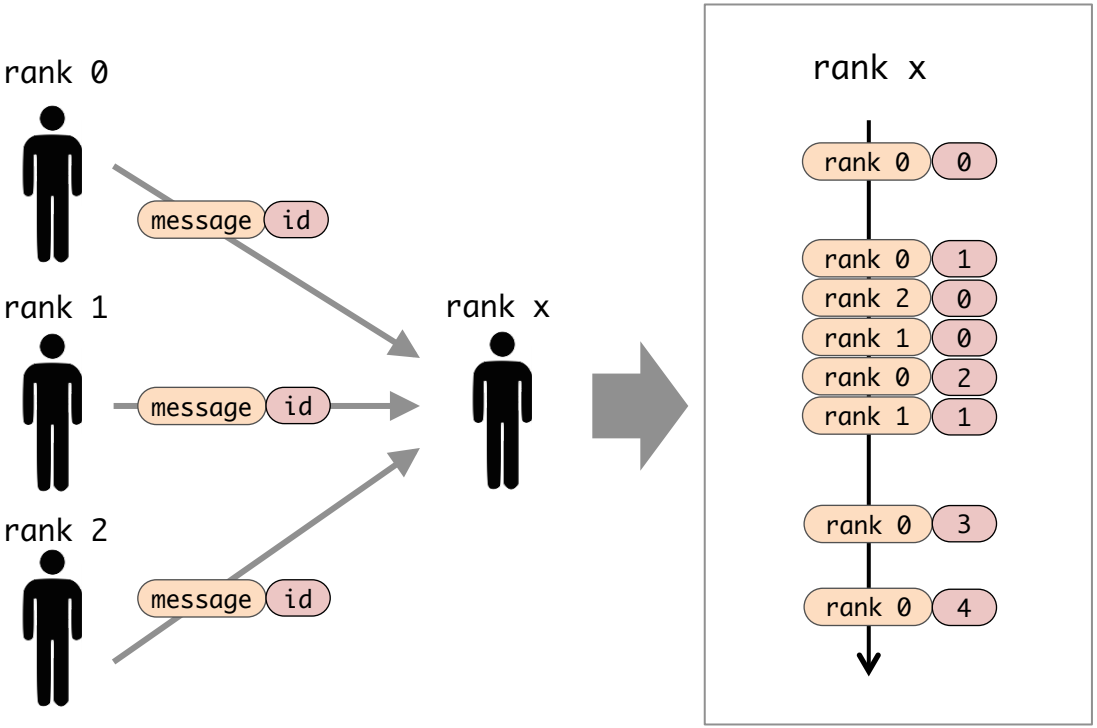
Application-level out-of-order

- MPI guarantees that any two communications executed by a process are ordered
  - Send order: msg A ➔ msg B
  - Recv order: msg A ➔ msg B
- However, timing of matching function calls depends on an application
  - Message receive order is not necessary equal to message send order
- For example,
  - "msg: B" may matches earlier than "msg: A"
- Recording only "rank" cannot distinguish between "msg A ➔ msg B" and " msg B ➔ msg A"





Each rank need to assign “id” number to each message

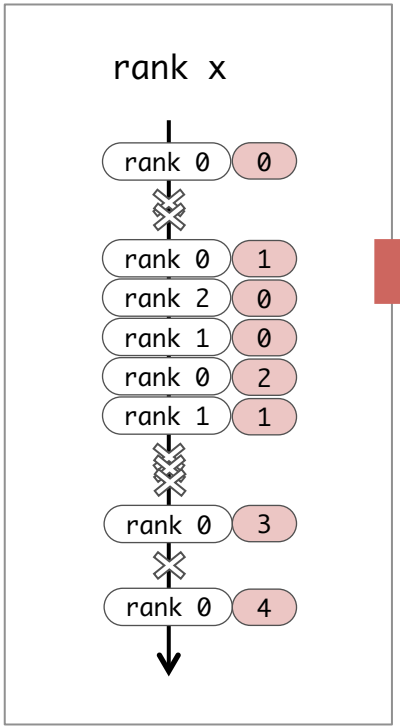


Necessary values for correct replay

Matching functions in MPI

	Wait family	Test family
single	MPI_Wait	MPI_Test
any	MPI_Waitany	MPI_Testany
some	MPI_Waitsome	MPI_Testsome
all	MPI_Waitall	MPI_Testall

- rank
  - Who send the messages?
- count & flag
  - For MPI_Test family
    - flag: Matched or unmatched ?
    - count: How many time unmatched ?
- id
  - For application-level out-of-order
- with_next
  - For matching some/all functions



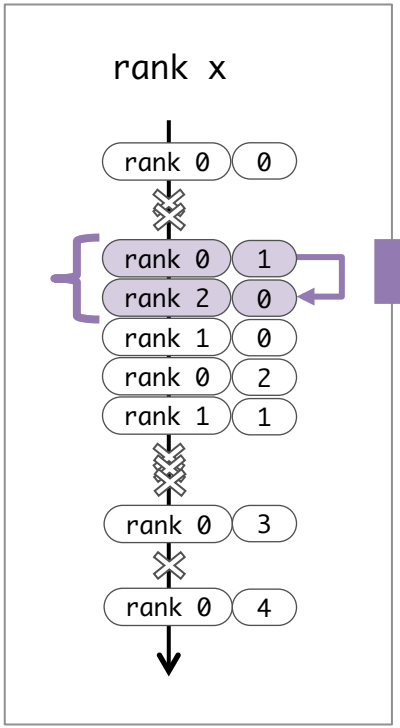
count	flag	rank	id
--	1	0	0
2	0	--	--
			1
			0
--	1	1	0
--	1	0	2
--	1	1	1
3	0	--	--
--	1	0	3
--	0	--	--
1	1	0	4

Necessary values for correct replay

Matching functions in MPI

	Wait family	Test family
single	MPI_Wait	MPI_Test
any	MPI_Waitany	MPI_Testany
some	MPI_Waitsome	MPI_Testsome
all	MPI_Waitall	MPI_Testall

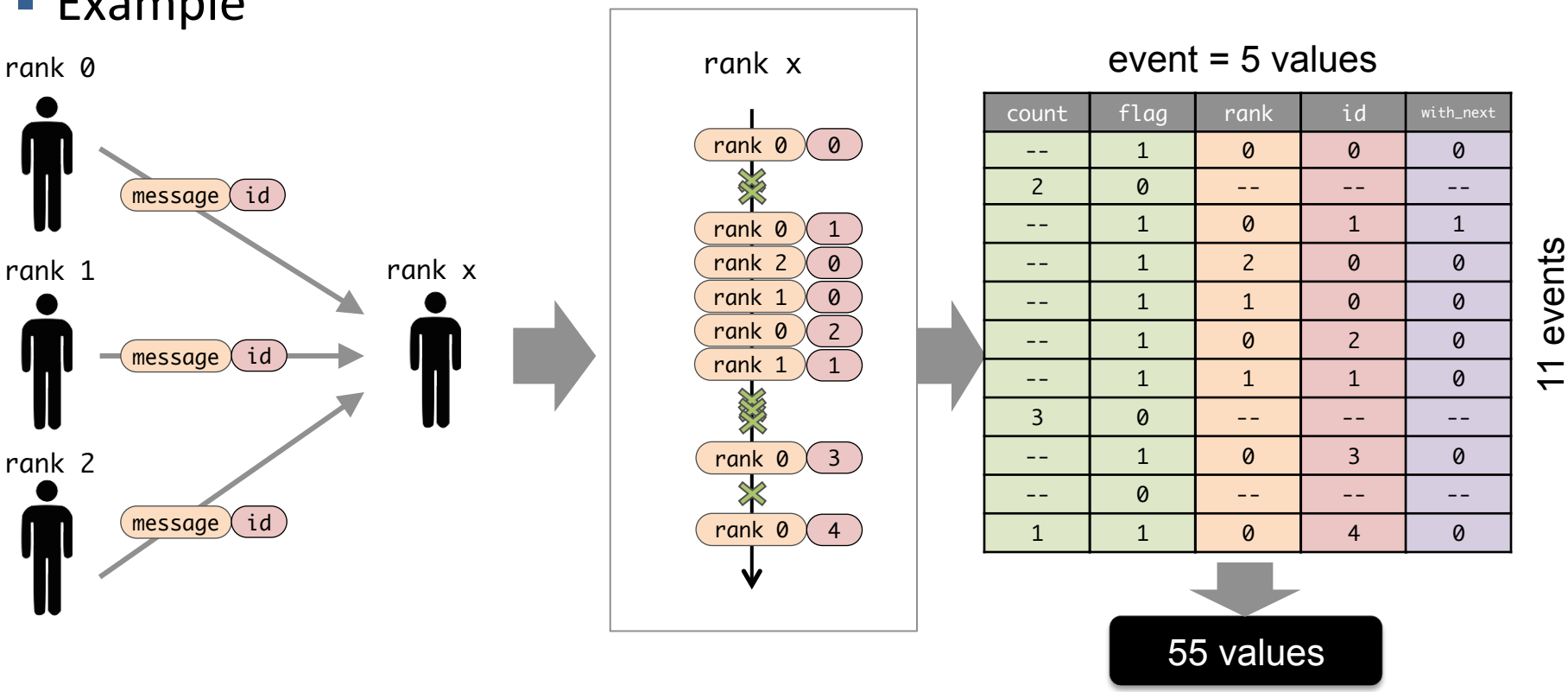
- rank
  - Who send the messages?
- count & flag
  - For MPI_Test family
    - flag: Matched or unmatched ?
    - count: How many time unmatched ?
- id
  - For application-level out-of-order
- with_next
  - For matching some/all functions



count	flag	rank	id	with_next
--	1	0	0	0
2	0	--	--	--
				1
				0
--	1	1	0	0
--	1	0	2	0
--	1	1	1	0
3	0	--	--	--
--	1	0	3	0
--	0	--	--	--
1	1	0	4	0

Necessary values for correct replay

- Example



CDC: Clock delta compression

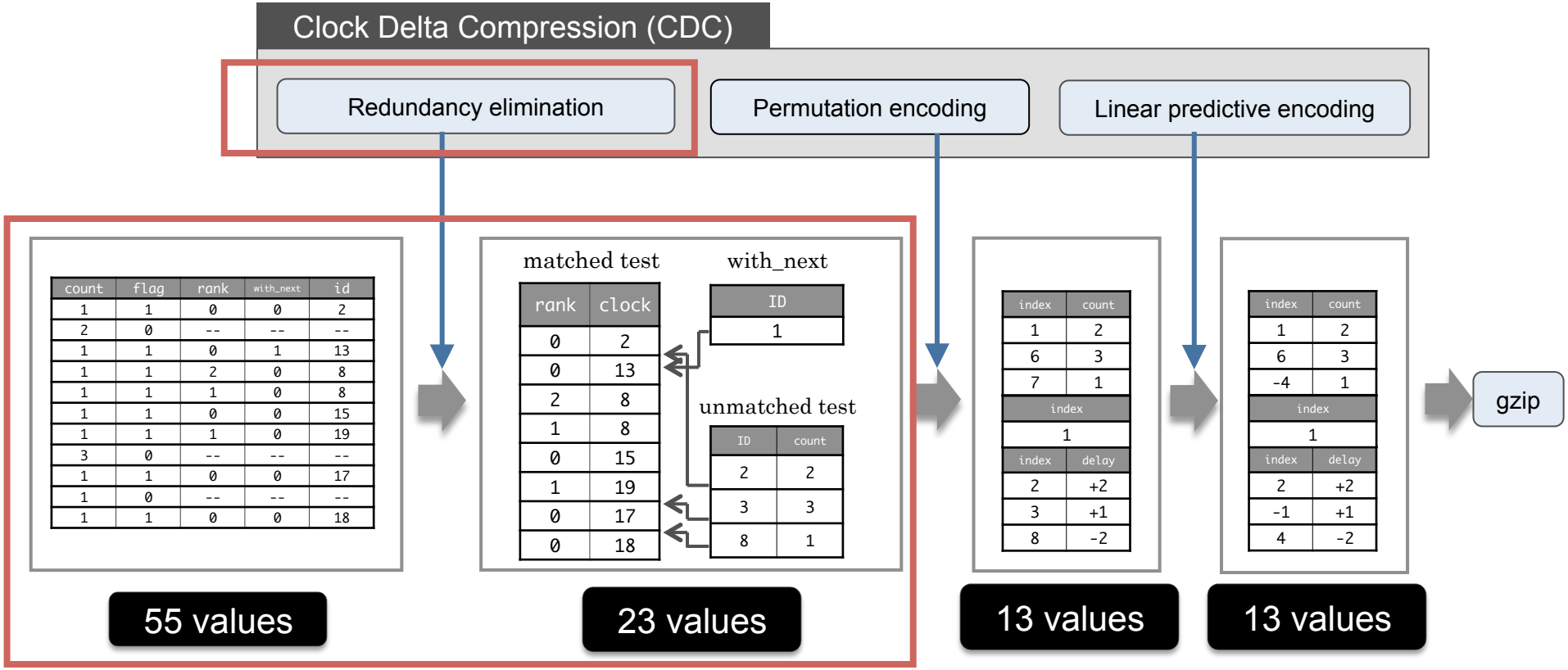
Clock Delta Compression (CDC)

- Redundancy elimination
- Permutation encoding
- Linear predictive encoding

count	flag	rank	with_next	id
1	1	0	0	2
2	0	--	--	--
1	1	0	1	13
1	1	2	0	8
1	1	1	0	8
1	1	0	0	15
1	1	1	0	19
3	0	--	--	--
1	1	0	0	17
1	0	--	--	--
1	1	0	0	18

55 values

CDC: Clock delta compression



count	flag	rank	with_next	id
1	1	0	0	2
2	0	--	--	--
1	1	0	1	13
1	1	2	0	8
1	1	1	0	8
1	1	0	0	15
1	1	1	0	19
3	0	--	--	--
1	1	0	0	17
1	0	--	--	--
1	1	0	0	18

rank	clock
0	2
0	13
2	8
1	8
0	15
1	19
0	17
0	18

ID
1

ID	count
2	2
3	3
8	1

index	count
1	2
6	3
7	1

index
1

index	delay
2	+2
3	+1
8	-2

index	count
1	2
6	3
-4	1

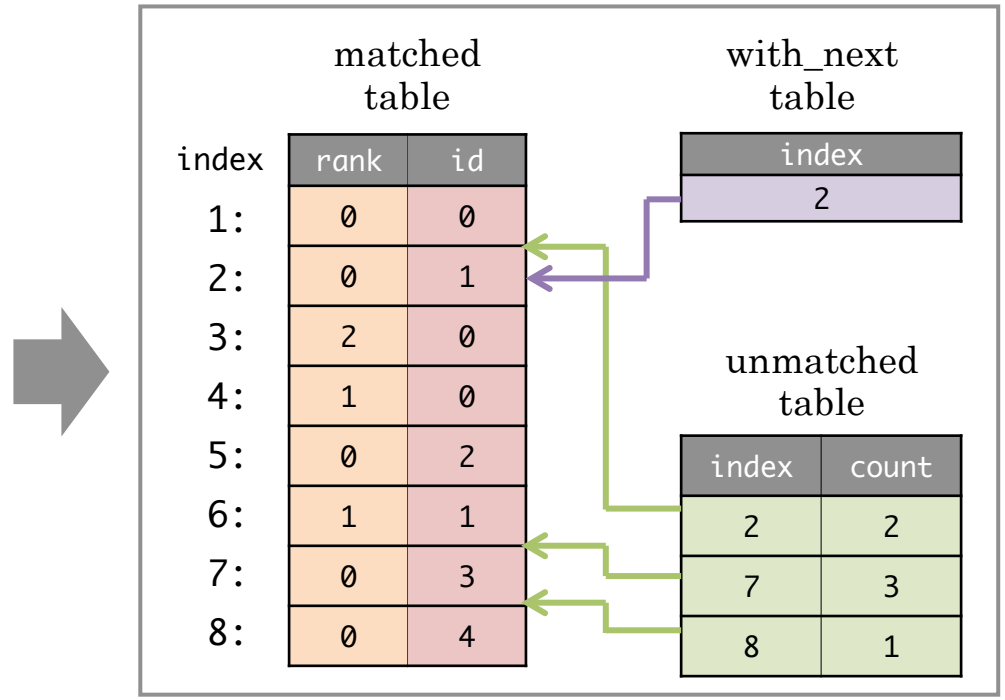
index
1

index	delay
2	+2
-1	+1
4	-2

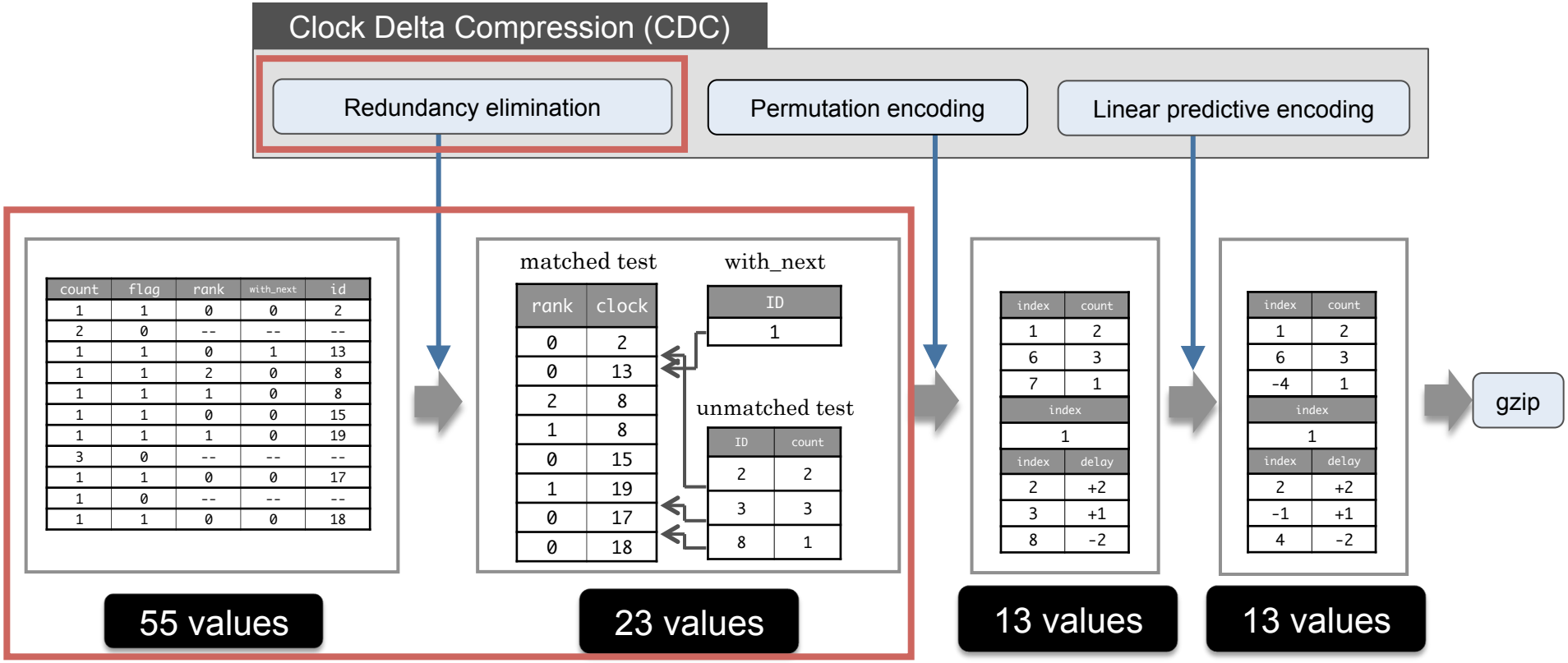
Redundancy elimination

- The base record has redundancy
- To eliminate redundancy, and we divide the original table into three tables
  - matched events table (rank & id)
  - unmatched events table (count & flag)
  - with_next table (with_next)

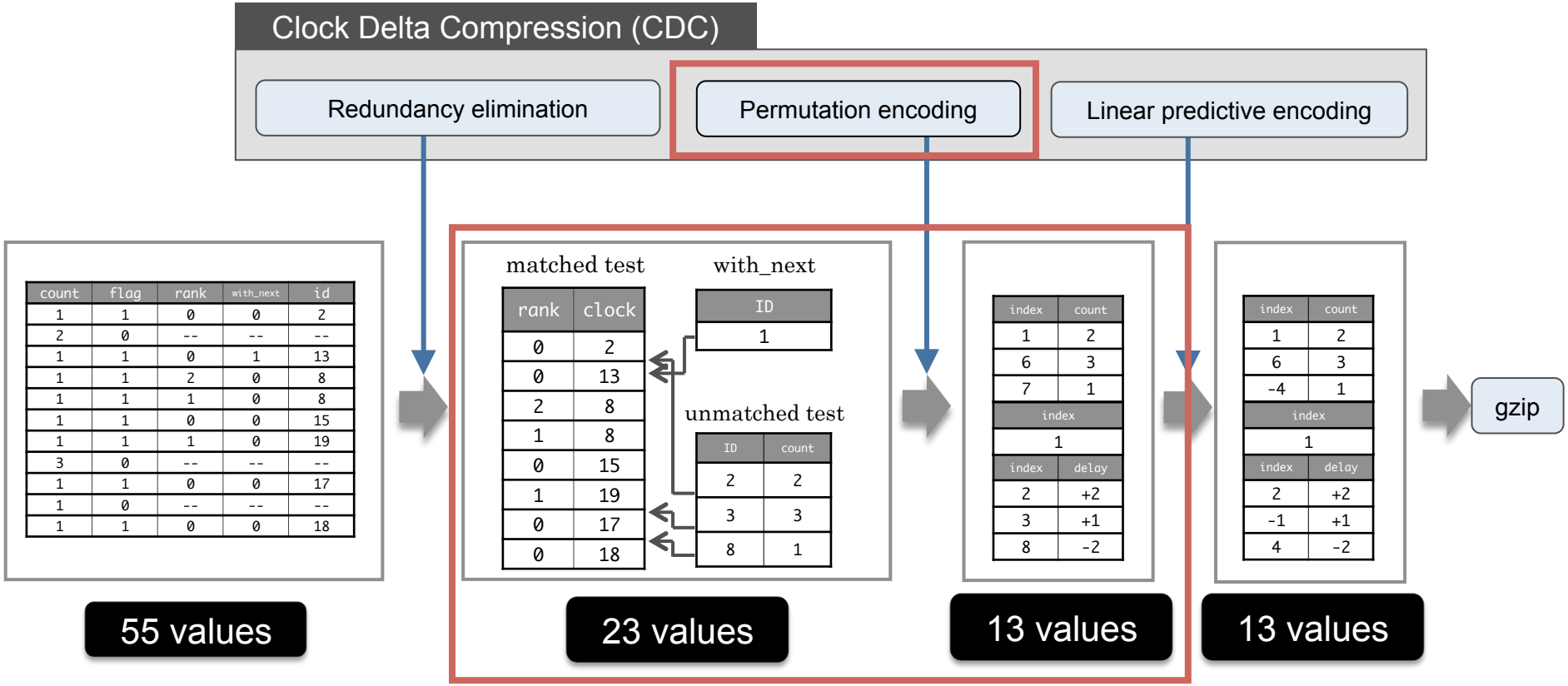
unmatched table		matched table		with_next table
count	flag	rank	id	with_next
--	1	0	0	0
2	0	--	--	--
--	1	0	1	1
--	1	2	0	0
--	1	1	0	0
--	1	0	2	0
--	1	1	1	0
3	0	--	--	--
--	1	0	3	0
1	0	--	--	--
--	1	0	4	0



CDC: Clock delta compression



CDC: Clock delta compression



count	flag	rank	with_next	id
1	1	0	0	2
2	0	--	--	--
1	1	0	1	13
1	1	2	0	8
1	1	1	0	8
1	1	0	0	15
1	1	1	0	19
3	0	--	--	--
1	1	0	0	17
1	0	--	--	--
1	1	0	0	18

rank	clock
0	2
0	13
2	8
1	8
0	15
1	19
0	17
0	18

ID
1

ID	count
2	2
3	3
8	1

index	count
1	2
6	3
7	1

index
1

index	delay
2	+2
3	+1
8	-2

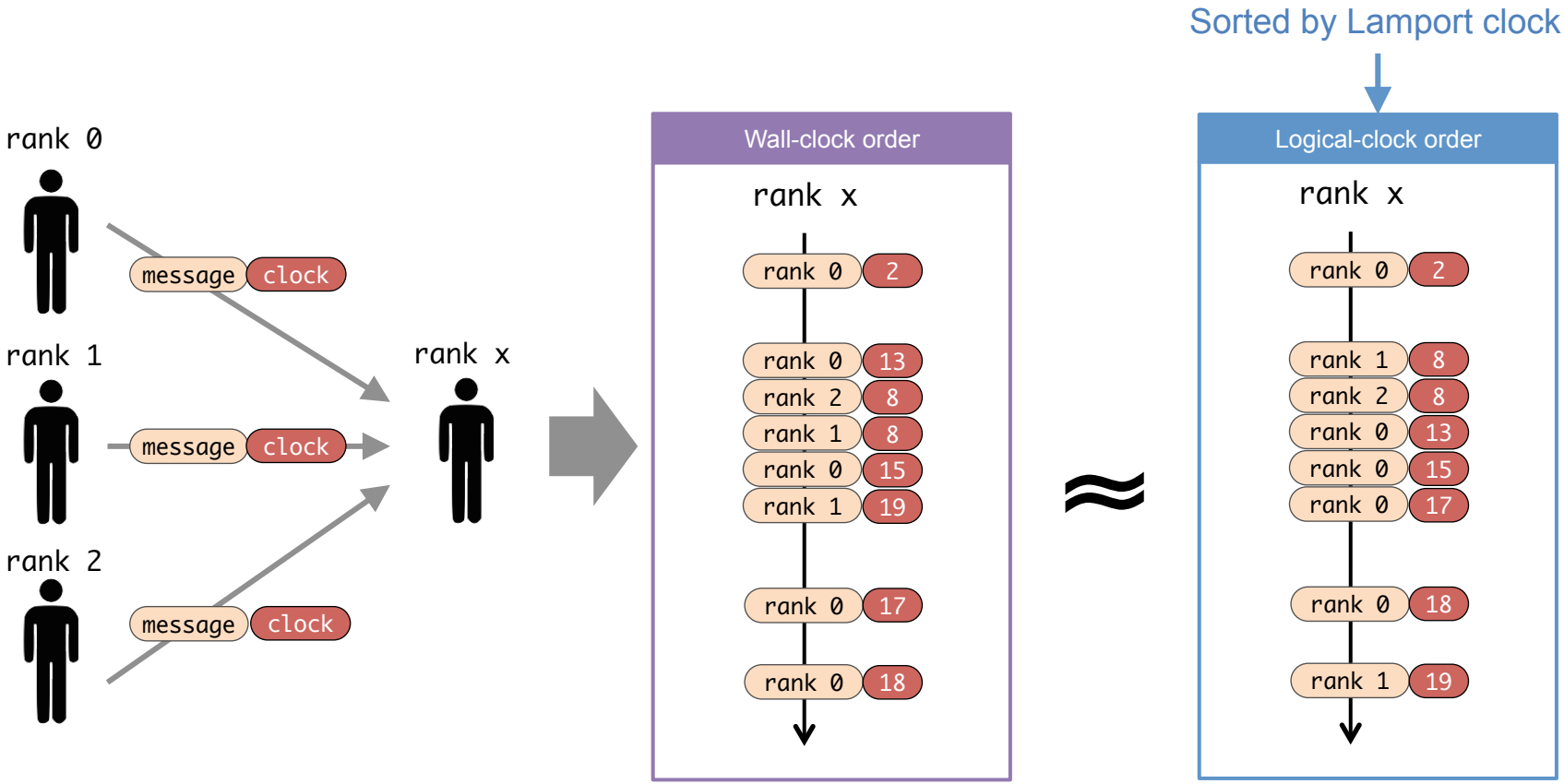
index	count
1	2
6	3
-4	1

index
1

index	delay
2	+2
-1	+1
4	-2

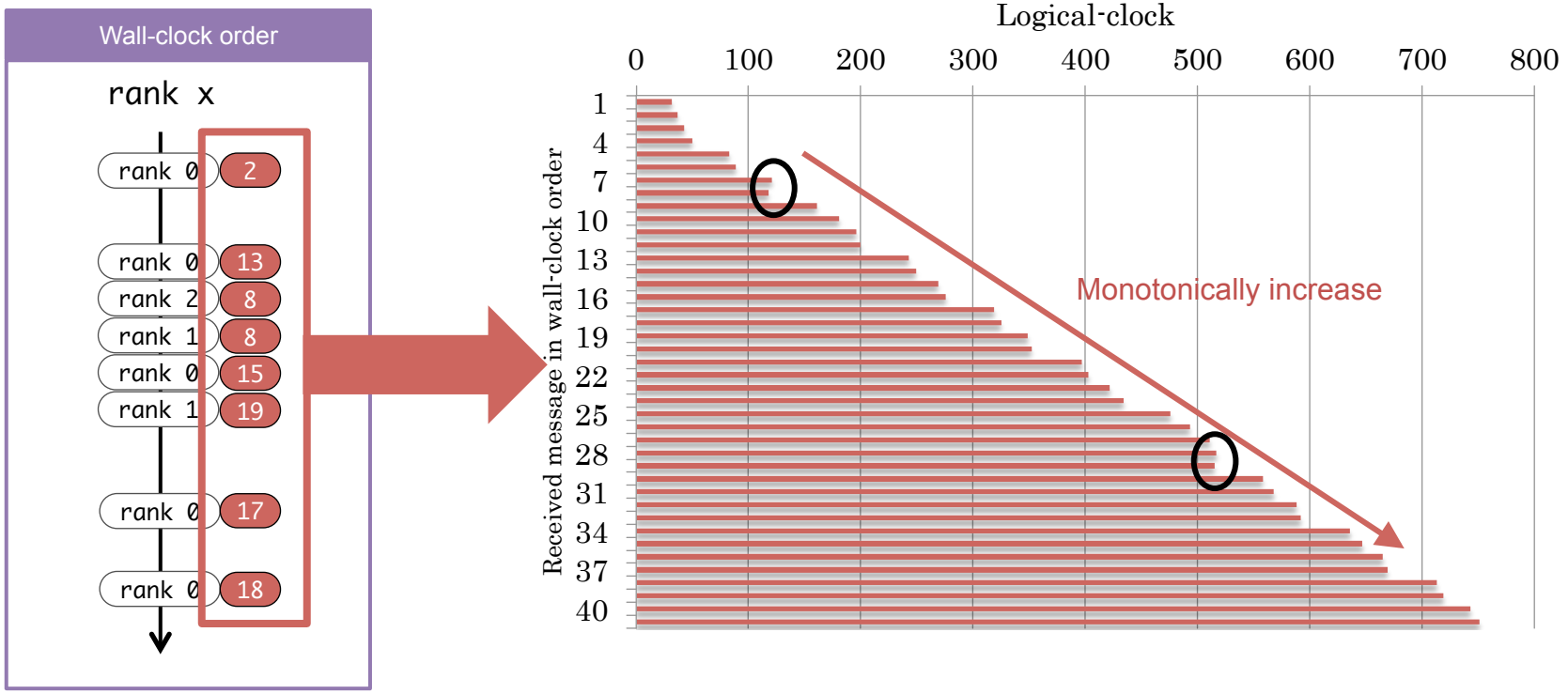
Key observation in communications

- Received order (Wall-clock order) are very similar to Logical-clock order
  - Put "Lamport clock" instead of msg "id" when sending a message



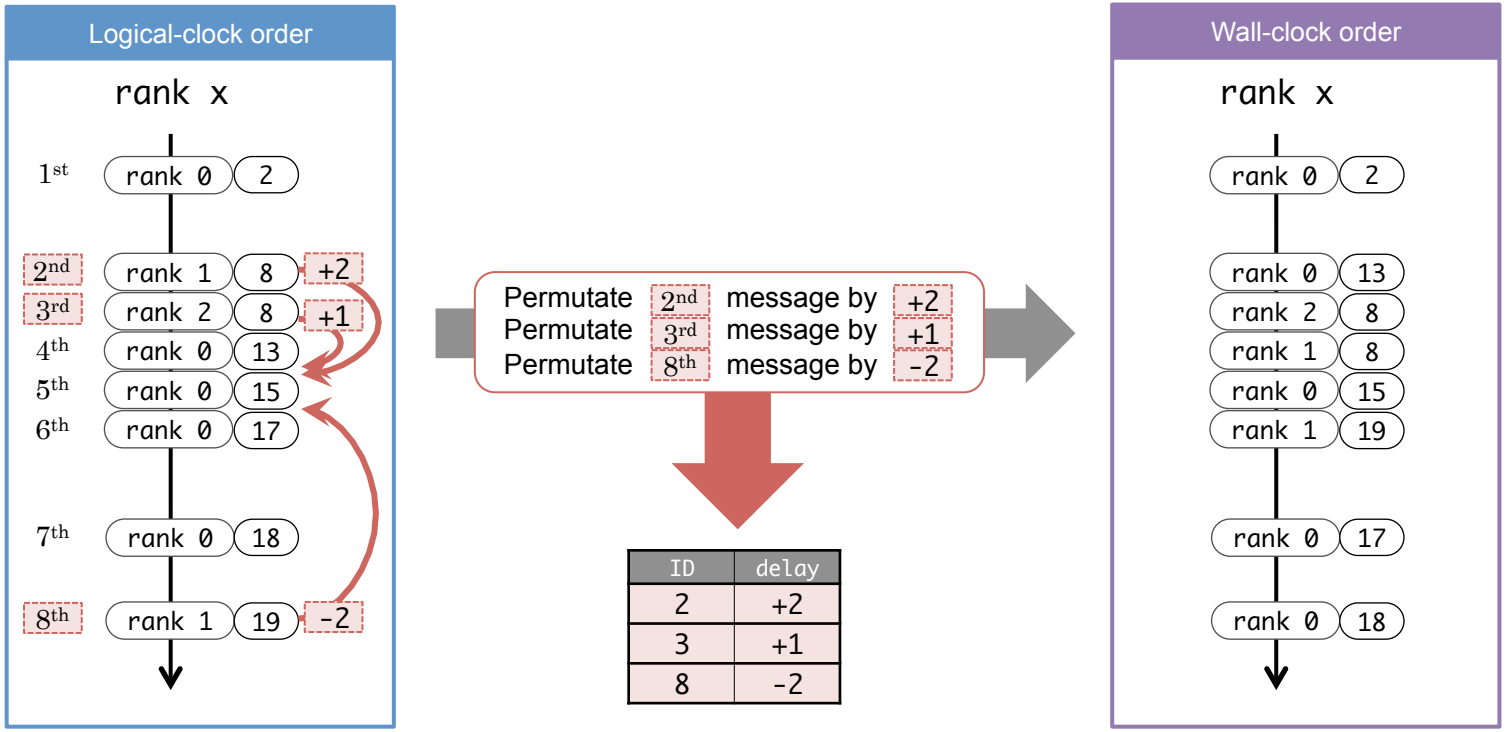
Case study: Received logical-clock values in MCB

- Received logical-clock values in a received order
  - Almost monotonically increase ➔ received order == logical-clock order



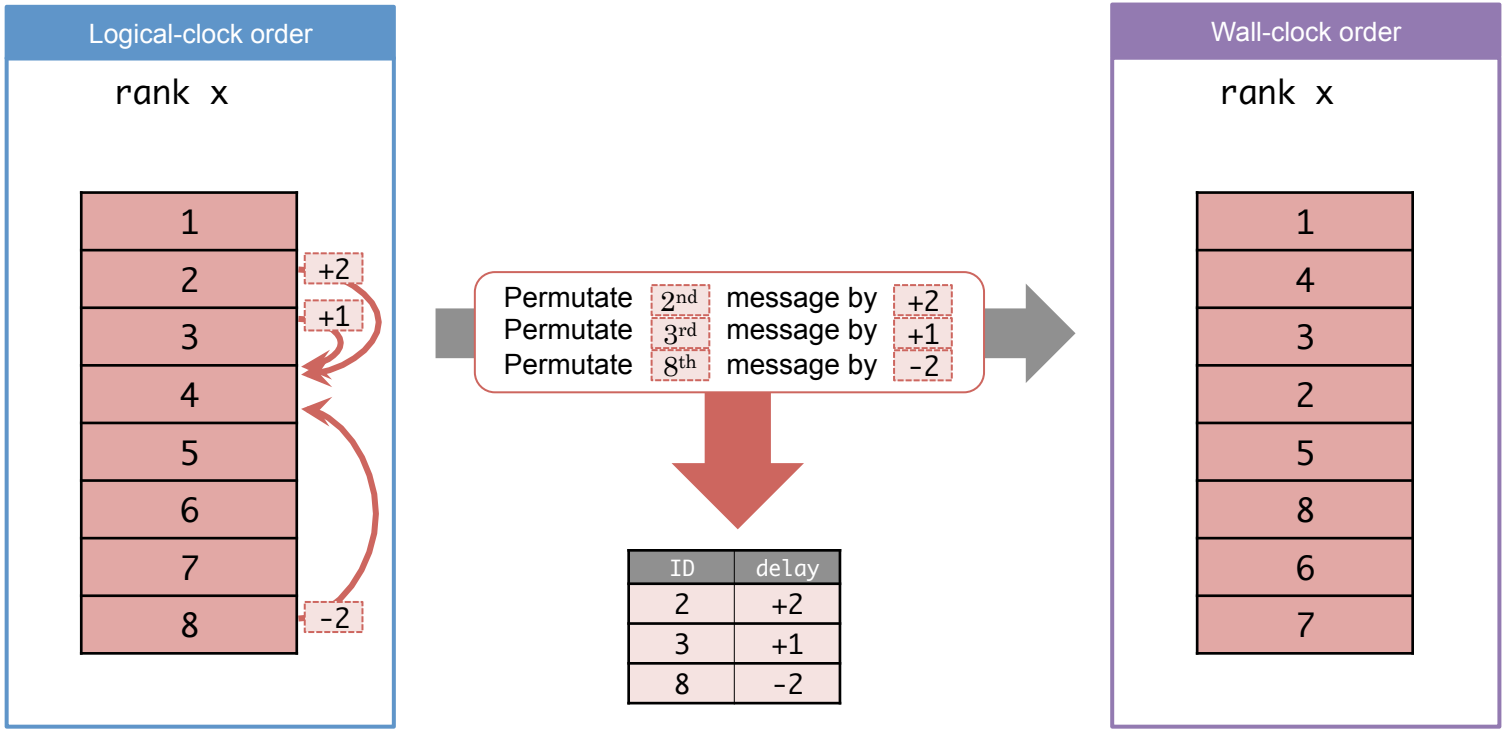
Permutation encoding

- We only records the difference between wall-order and logical-order instead of recording entire received order



Permutation encoding

- Permutation encoding can be regarded as an edit distance problem computing minimal permutations to create from sequential numbers to observed wall-clock order

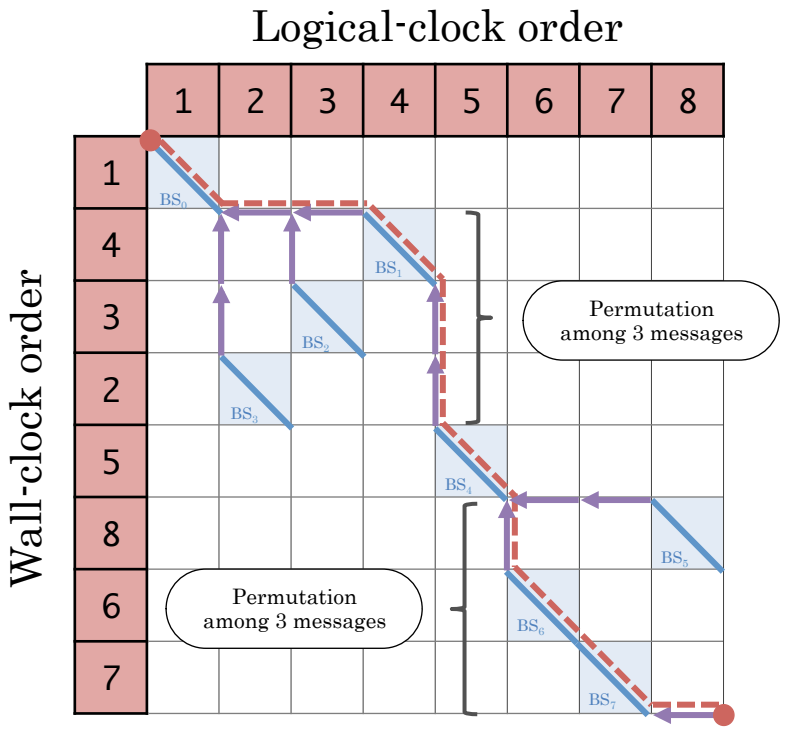


Edit distance algorithm

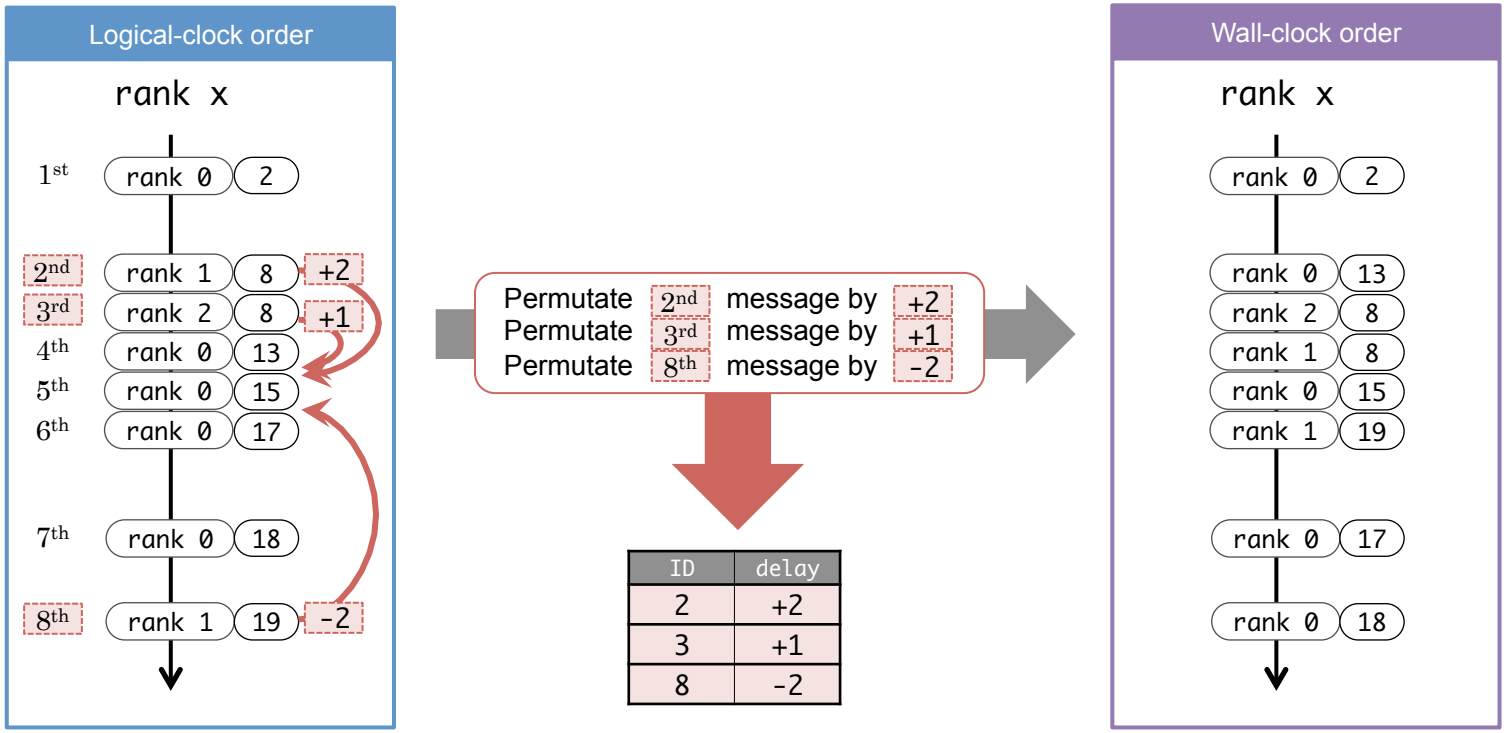
- Edit distance algorithm
  - Compute similarity between two strings
    - Wall-clock order
    - Logical-clock order
  - Time complexity: $O(N^2)$
    - N: length of the strings

- Special conditions in CDC
  1. Logical-clock order is sequential numbers
  2. Wall-clock order is created by permutations of Logical-clock

  ➔ Time complexity: O(N+D)
    - N: Length of the strings
    - D: Edit distance

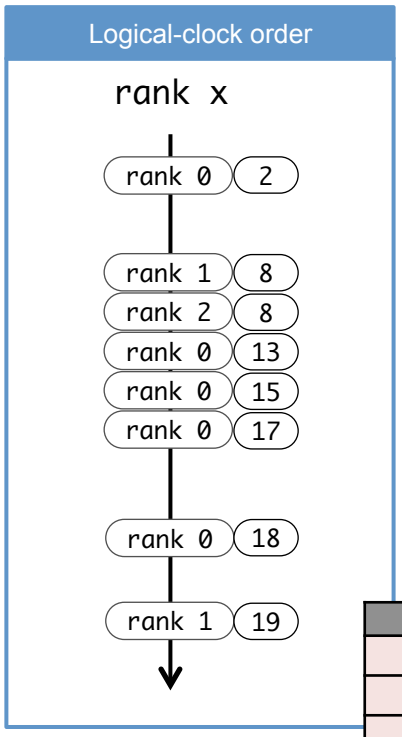


Why Logical-clock order is not recorded ?

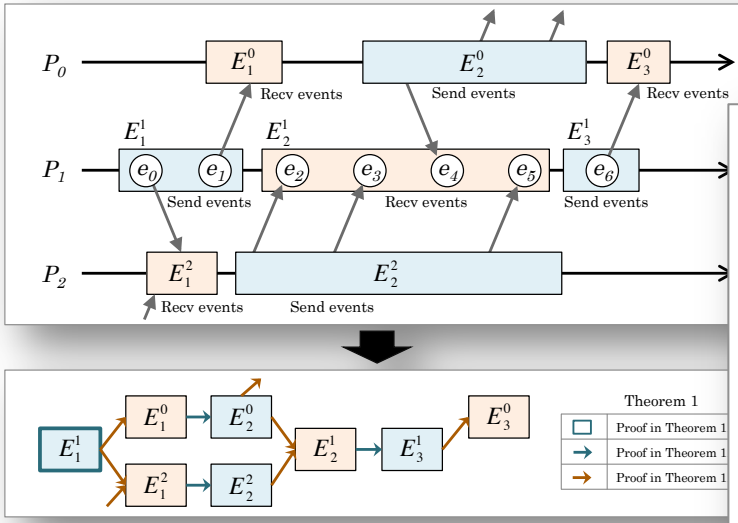


Logical-clock order is reproducible

- Logical-clock order is always reproducible, so CDC only records the permutation difference



ID	delay
2	+2
3	+1
8	-2



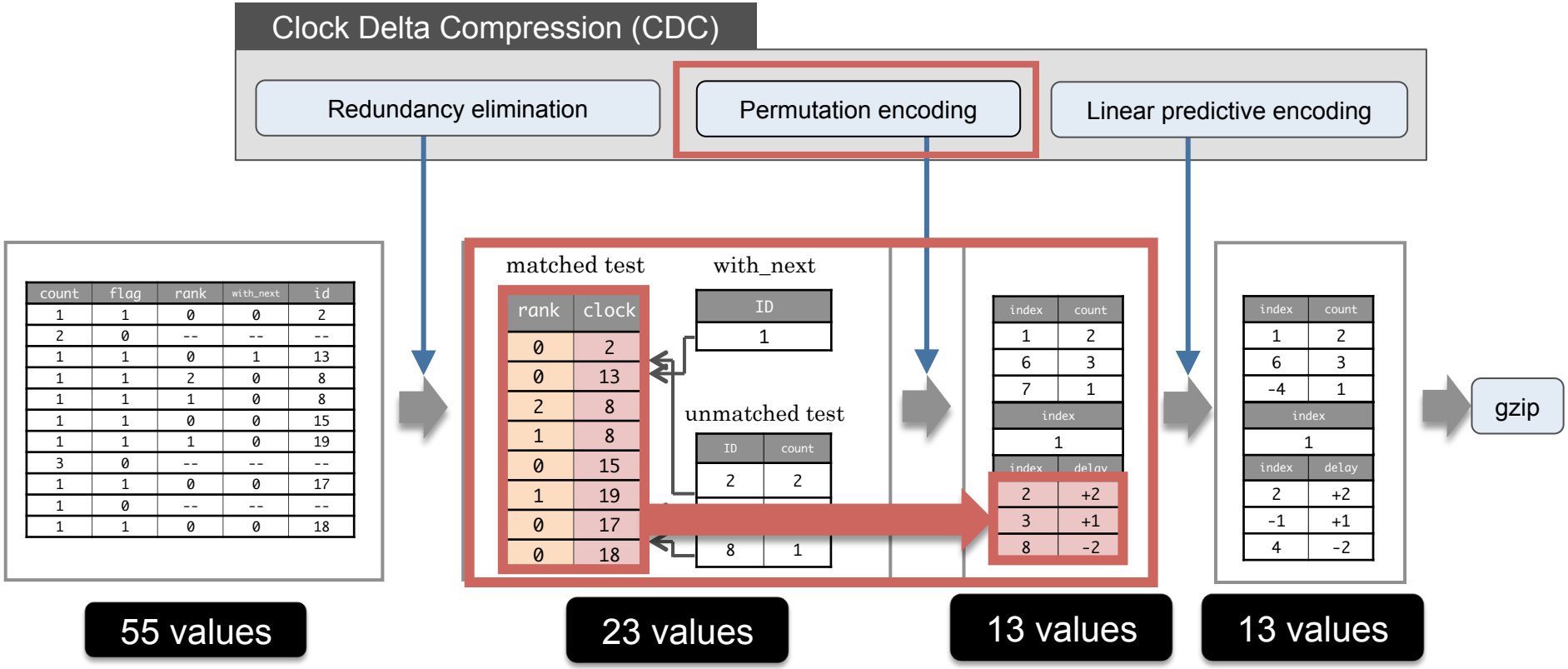
Theorem 1. CDC can correctly replay message events, that is, $\boldsymbol{E} = \hat{\boldsymbol{E}}$ where $\boldsymbol{E}$ and $\hat{\boldsymbol{E}}$ are ordered sets of events for a record and a replay mode.

PROOF (MATHEMATICAL INDUCTION). **(i) Basis**: Show the first send events are replayable, i.e., $^{\forall}x$ s.t. "E_1^x is send events" $\Rightarrow$ "E_1^x is replayable". As defined in Definition 7.(i) E_1^x is deterministic, that is , E_1^x is always replayed. In Figure 12, E_1^1 is deterministic, that is, is always replayed. **(ii) Inductive step for send events**: Show send events are replayable if the all previous message events are replayed, i.e., "$^{\forall}\boldsymbol{E} \rightarrow E$ s.t. $\boldsymbol{E}$ is replayed, E is send event set" $\Rightarrow$ "E is replayable". As defined in Definition 7.(ii), E is deterministic, that is, E is always replayed. **(iii) Inductive step for receive events**: Show receive events are replayable if the all previous message events are replayed, i.e., "$^{\forall}\boldsymbol{E} \rightarrow E$ s.t. $\boldsymbol{E}$ is replayed, E is receive event set" $\Rightarrow$ "E is replayable". As proofed in Proposition 1, all message receives in E can be replayed by CDC. Therefore, all of the events can be replayed, i.e., $\boldsymbol{E} = \hat{\boldsymbol{E}}$. (Mathematical induction processes are graphically shown in Figure 12.) ■

Theorem 2. CDC can replay piggyback clocks.

PROOF. As proved in Theorem 1, since CDC can replay all message events, send events and clock ticking are replayed. Thus, CDC can replay piggyback clock sends. ■

CDC: Clock delta compression



count	flag	rank	with_next	id
1	1	0	0	2
2	0	--	--	--
1	1	0	1	13
1	1	2	0	8
1	1	1	0	8
1	1	0	0	15
1	1	1	0	19
3	0	--	--	--
1	1	0	0	17
1	0	--	--	--
1	1	0	0	18

rank	clock
0	2
0	13
2	8
1	8
0	15
1	19
0	17
0	18

ID
1

ID	count
2	2
8	1

index	count
1	2
6	3
7	1

index
1

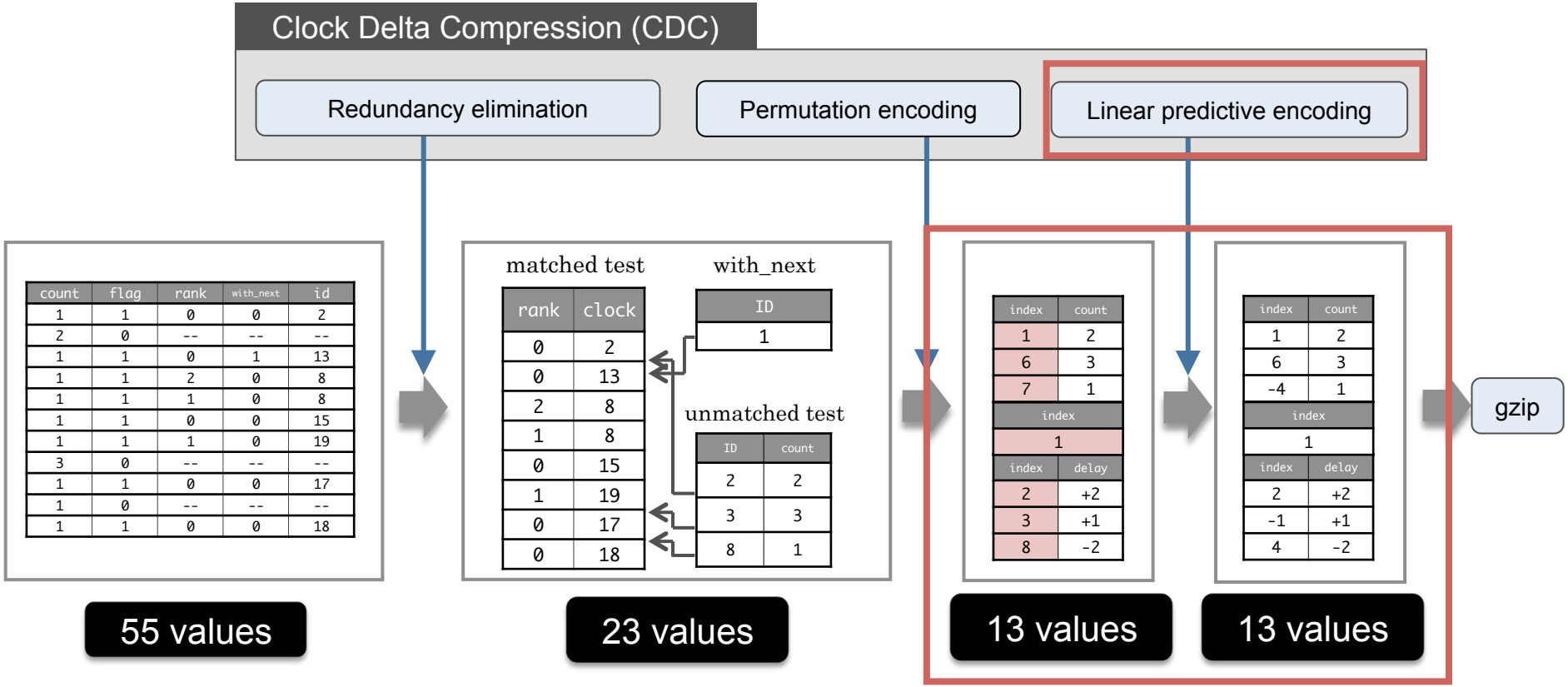
index	delay
2	+2
3	+1
8	-2

index	count
1	2
6	3
-4	1

index
1

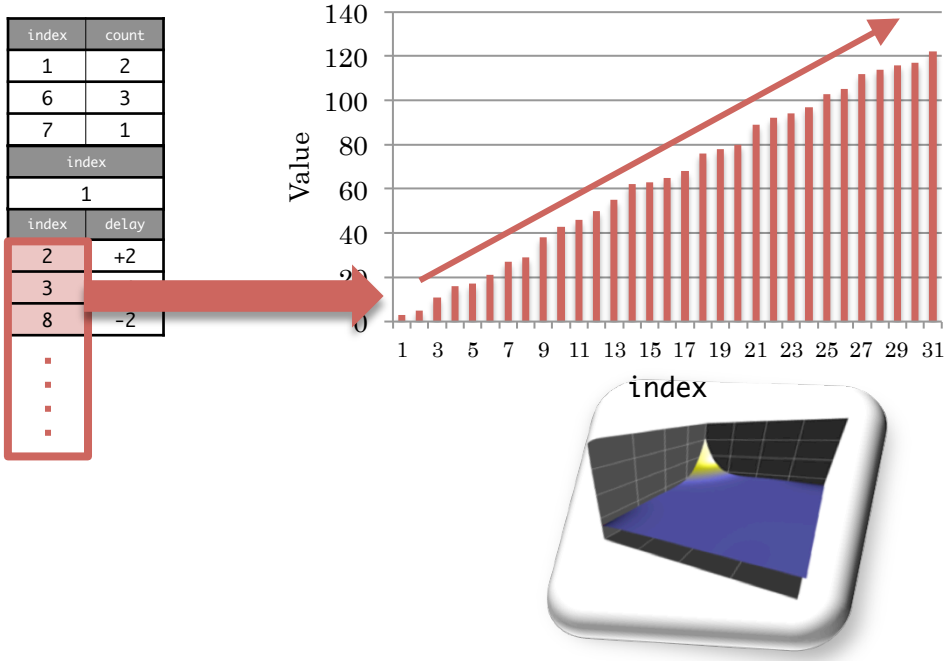
index	delay
2	+2
-1	+1
4	-2

CDC: Clock delta compression



Case study: index values in MCB

- Problem in the format: `index` values linearly increase as CDC records events
- Compression rate by gzip becomes worse as the table size increases
  - gzip encodes frequent sequence of bits into shorter bits
  - If we can encode these values into close to zero, gzip can give a high compression rate



Linear predictive (LP) encoding

- LP encoding is used for compressing sequence of values, such as audio data
- When encoding $\{x_1, x_2, \ldots, x_N\}$ LP encoding predicts each value x_n from the past p number of values assuming the sequence is linear, and store errors $\{e_1, e_2, \ldots, e_N\}$.

$$\hat{x}_n = a_1 x_{n-1} + a_2 x_{n-2} + \quad \ldots \quad + a_p x_{n-p}$$

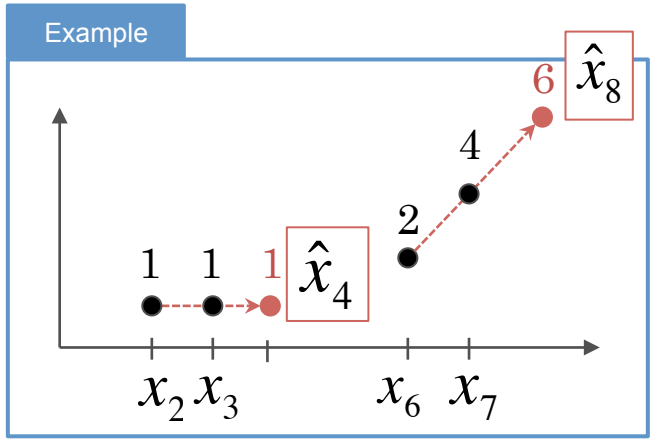
$$e_n = x_n - \hat{x}_n$$

If you give a good prediction, the index values become close to zero

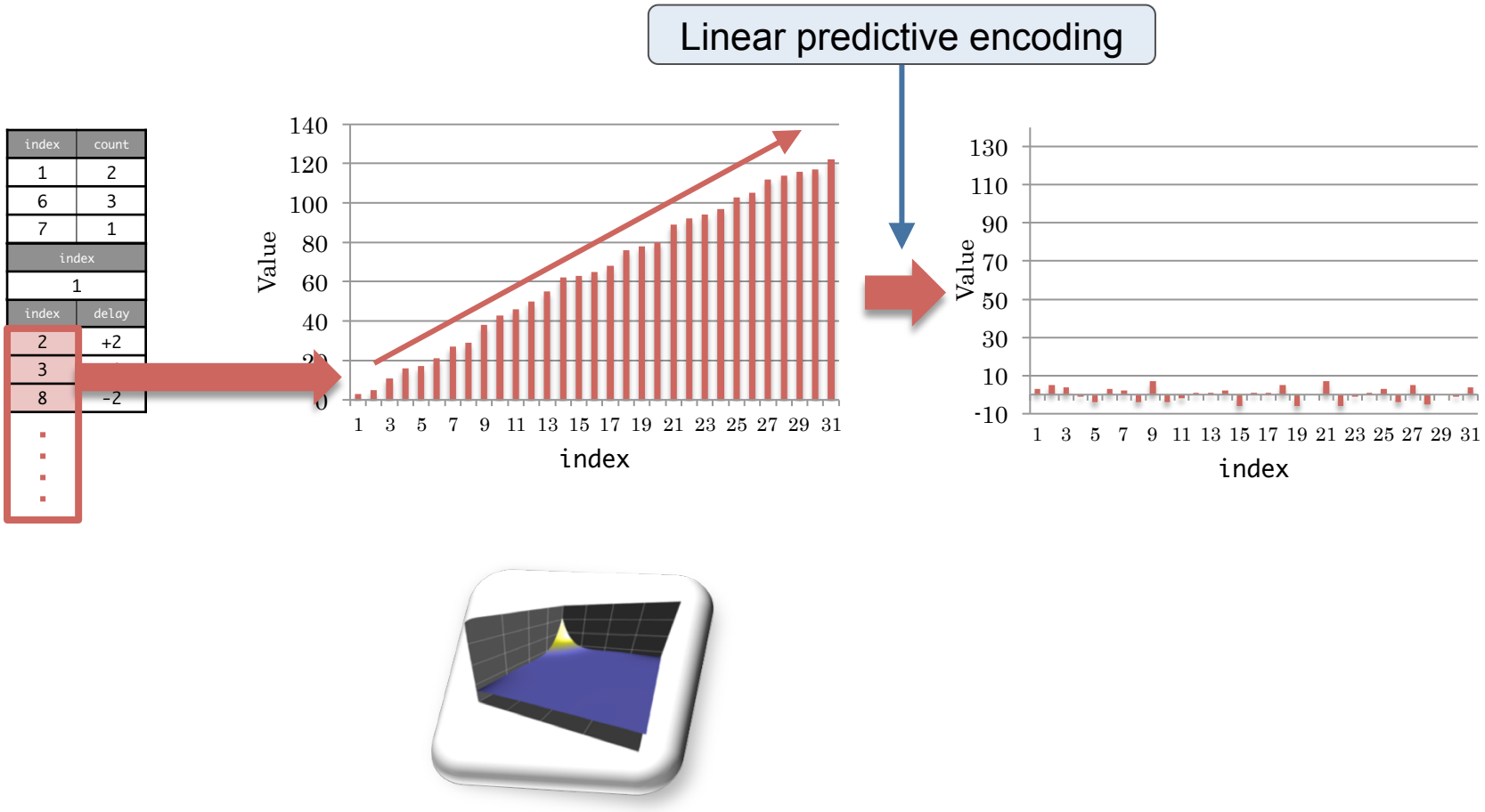
- Choice of p, and co-efficients, $\{a_1, a_2, \ldots, a_p\}$, affects accuracy of prediction
- In CDC, we predict x_n is on an extension of a line created by x_{n-1}, x_{n-2}

$$p = 2$$
$$\{a_1, a_2\} = \{2, -1\}$$

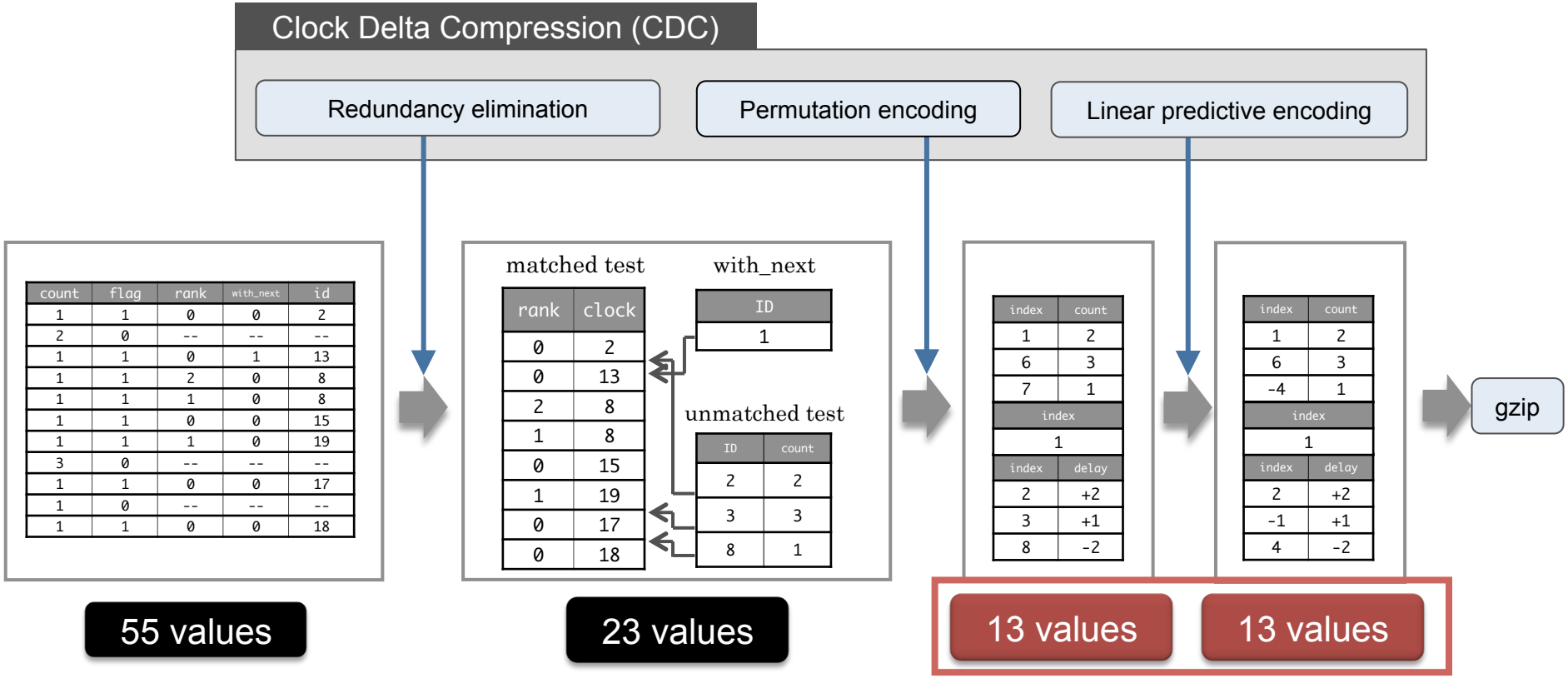
Example



Case study: Linear predictive encoding in MCB

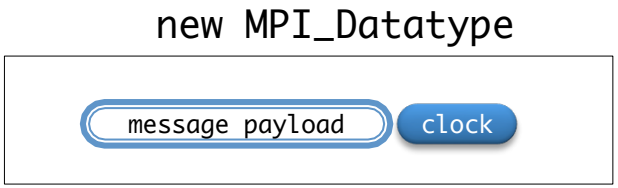


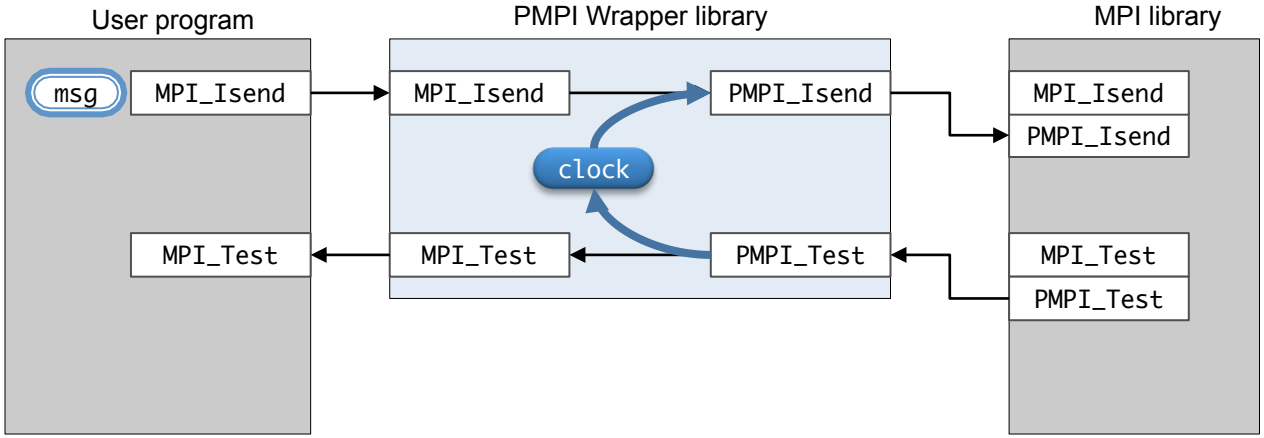
CDC: Clock delta compression



Implementation: Clock piggybacking [1]

- We use PMPI wrapper to record
  - events and clock piggybacking
- Clock piggybacking
  - MPI_Send/Isend:
    - When sending MPI message, the PMPI wrapper define new MPI_Datatype that combining message payload & clock
  - MPI Test/Wait family:
    - Retrieve the clock value, and synchronize the local Lamport clock
    - Pass record data to CDC thread

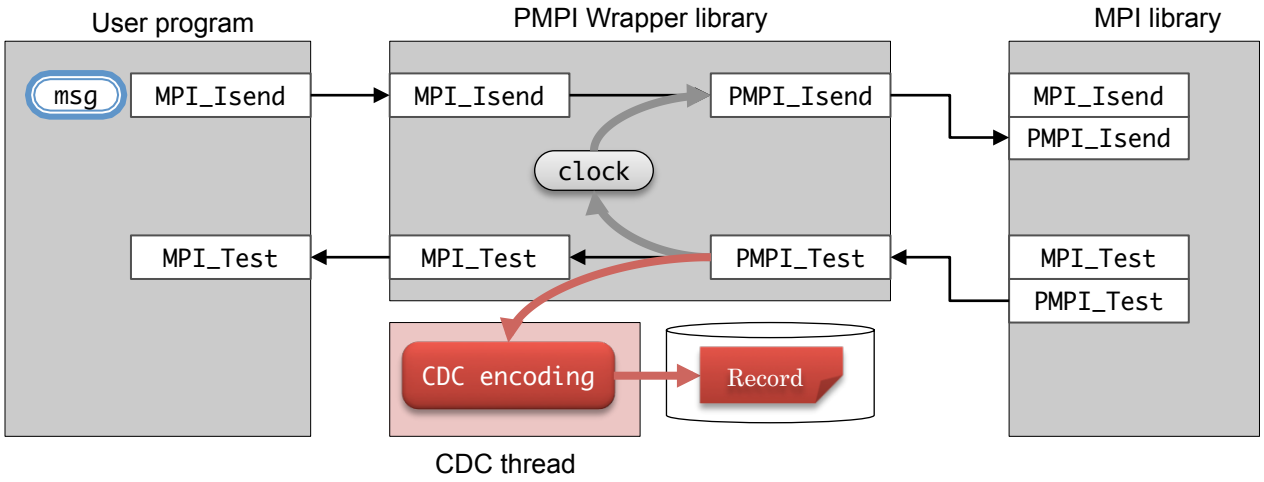




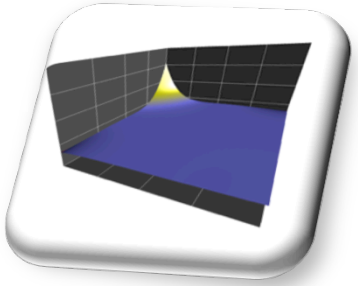
[1] M. Schulz, G. Bronevetsky, and B. R. Supinski. On the Performance of Transparent MPI Piggyback Messages. In Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 194–201, Berlin, Heidelberg, 2008. Springer-Verlag.

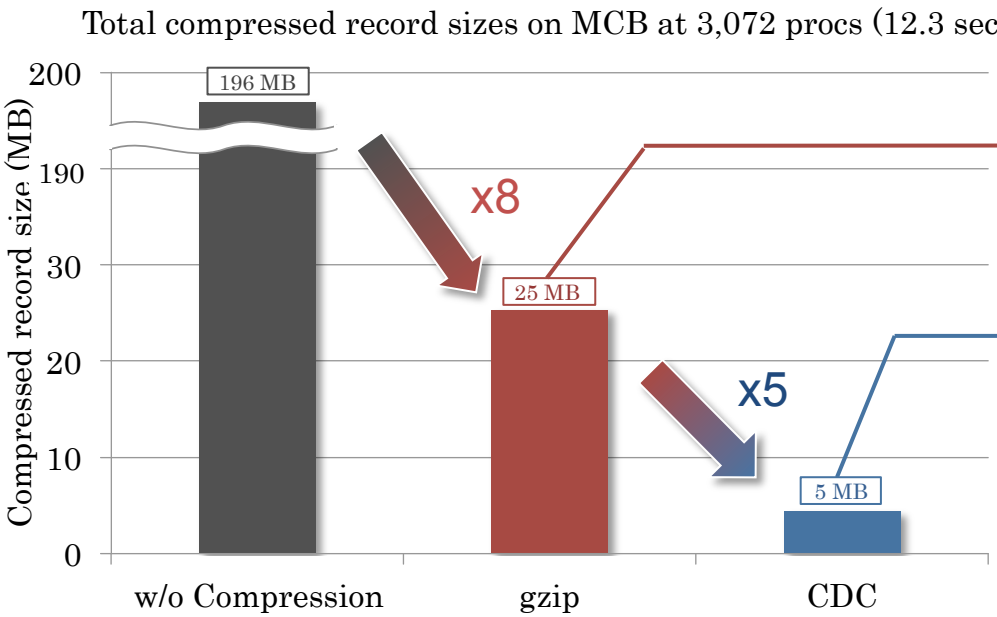
Asynchronous encoding

- CDC-dedicated thread is running
- Asynchronously compress and record events



Compression improvement in MCB





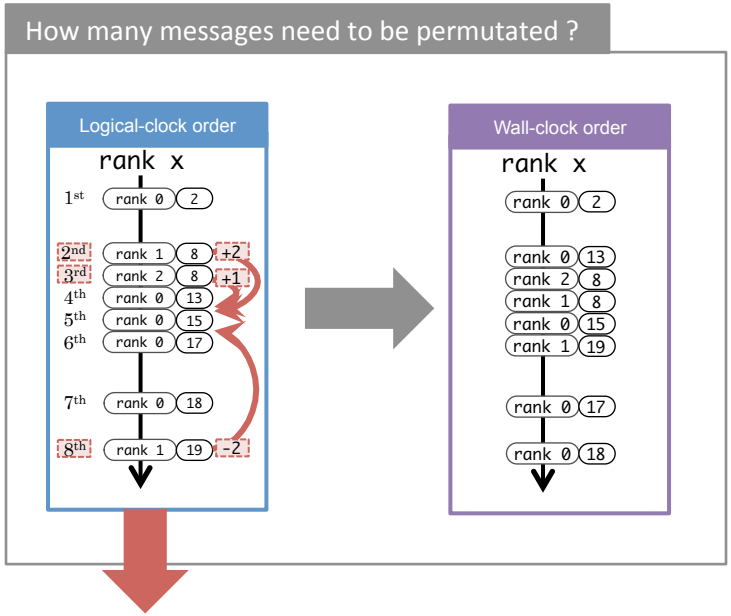
gzip itself can reduce the original format by 8x

5x more reduction

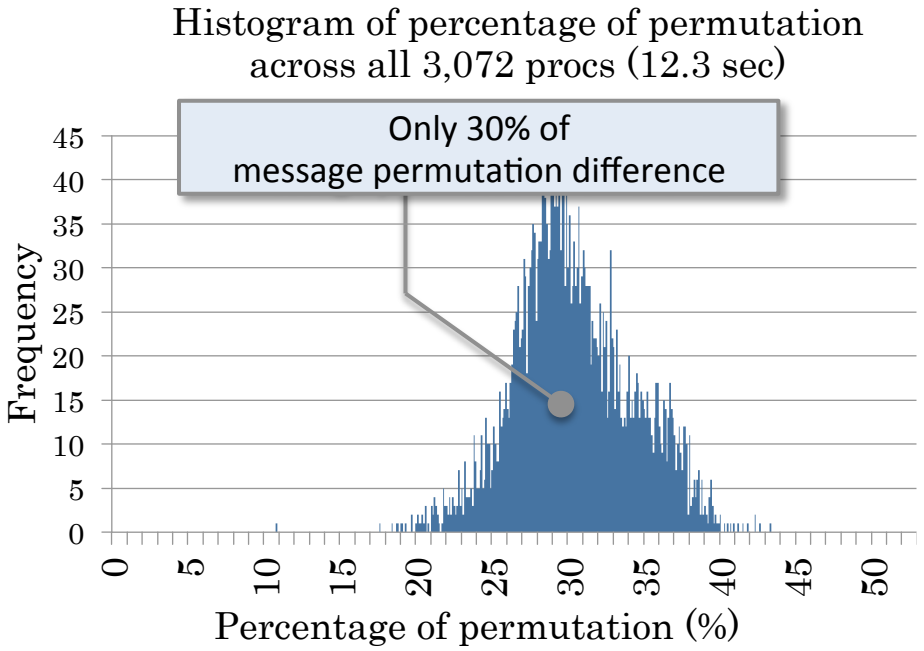
- For example, if 1GB of memory per node for record-and-replay …
  - w/o compression: 2 hours
  - gzip: 19 hours
  - CDC: 4 days

High compression Compressed size becomes 40x smaller than original size

Similarity between wall-clock and logical-clock order

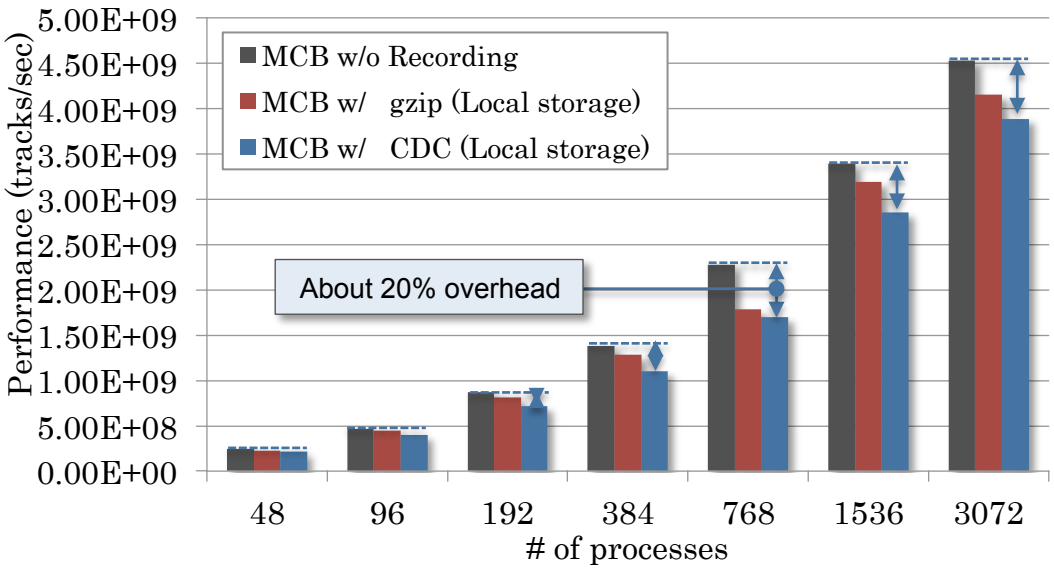


3 messages out of 8 = 37% of similarity



Compression overhead to performance

- Performance metric: how may particles are tracked per second



In both gzip and CDC, compression is asynchronously done.
The overhead to applications is minimized

CDC executes more complicated compression algorithm. CDC overhead becomes a little higher than gzip

In practice, capacity of local memory is limited. Because all record data must fit in local memory for scalability, high compression rate is more important than lower overhead

Low overhead CDC overhead are about 20% on average

Summary

- Irreproducibility is a common issue in HPC
- Existing toolset helps to track irreproducibility bugs
  - Spindle, ReMPI, io-watchdog and STAT
- Especially, ReMPI can help to reproduce buggy MPI behaviors
  - However, it produces large amount of data
  - This hampers scalability of the tool
- Clock Delta Compression (CDC)
  - Only record difference between wall-clock order and logical-clock order
    - Logical-clock order is always reproducible
- Our group will eradicate irreproducibility issues for computational science in future ☺

Toolset

- Spindle
  - http://computation.llnl.gov/projects/spindle
- ReMPI
  - (Preparing for software release)
- IO-watchdog
  - https://code.google.com/archive/p/io-watchdog/
- STAT
  - https://computing.llnl.gov/code/STAT/

Thanks !

Speaker:

Kento Sato (佐藤 賢斗)
Lawrence Livermore National Laboratory
https://kento.github.io

Acknowledgement

Dong H. Ahn, Ignacio Laguna, Gregory L. Lee,
Martin Schulz and Chambreau, Chris

Spindle
http://computation.llnl.gov/projects/spindle
ReMPI
(Preparing for software release)
IO-watchdog
https://code.google.com/archive/p/io-watchdog/
STAT
https://computing.llnl.gov/code/STAT/

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-PRES-681938).


Lawrence Livermore
National Laboratory