Clock Delta Compression for Scalable Order-Replay of Non-Deterministic Parallel Applications

SC15

Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, Martin Schulz

November 19th, 2015

LLNL-PRES-679294

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

Debugging large-scale applications is becoming problematic

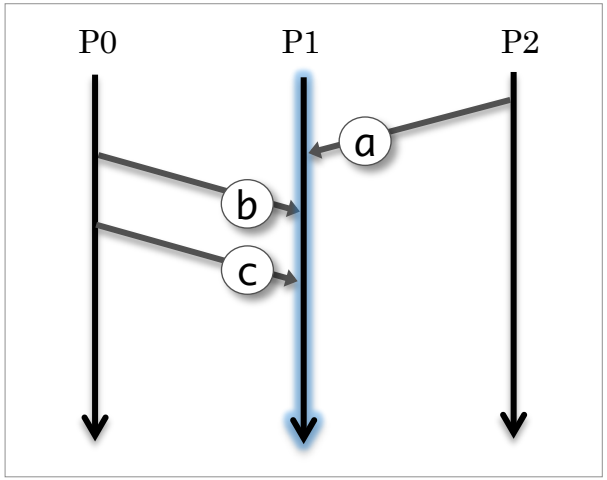
"On average, software developers spend 50% of their programming time finding and fixing bugs."[1]

With trends towards asynchronous communication patterns in MPI applications, **MPI non-determinism** will significantly increase debugging cost

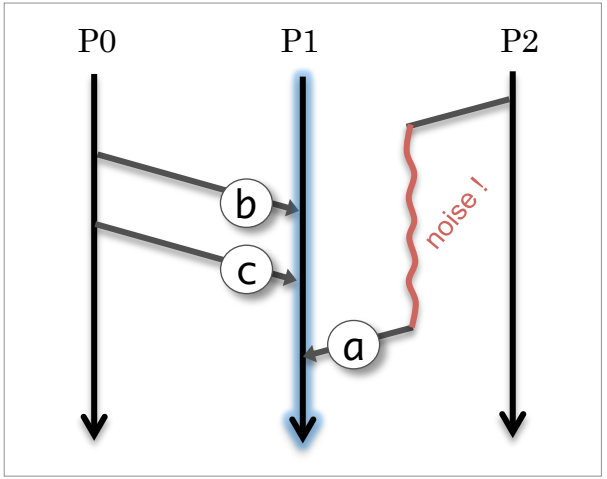
[1] Source: http://www.prweb.com/releases/2013/1/prweb10298185.htm, CAMBRIDGE, UK (PRWEB) JANUARY 08, 2013

What is MPI non-determinism (ND) ?

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



Execution A: (a+b)+c

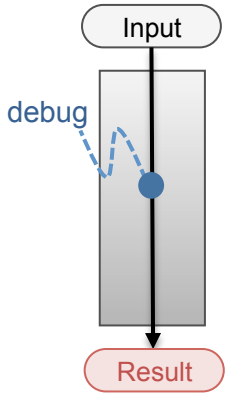


Execution B: a+(b+c)

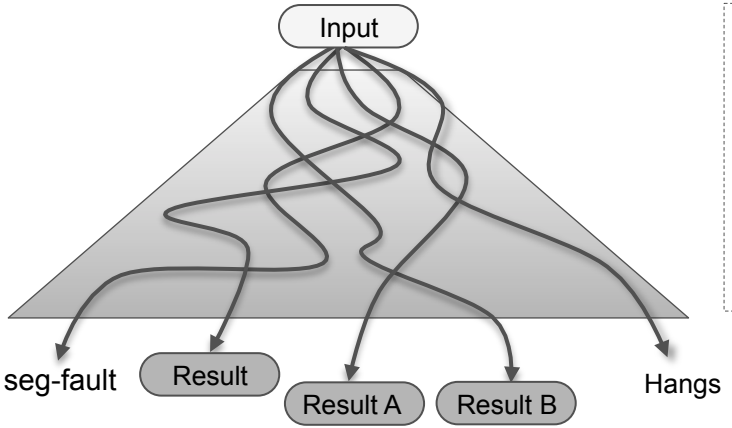
MPI non-determinism significantly increases debugging cost

- 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

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

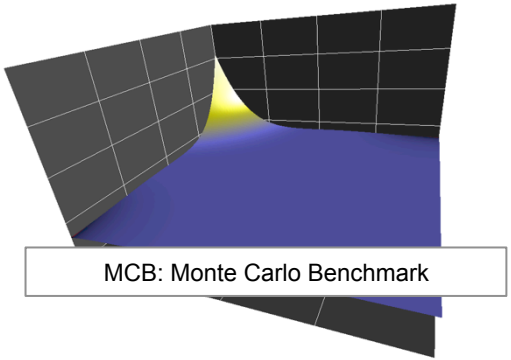
➔ Developers need to do debug runs until the same bug is reproduced

➔ 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

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

- CORAL proxy application
- MPI non-determinism



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

```
$ diff result_run1.out result_run2.out
result_run1.out:< IMC E_RR_total   -3.31402344[09e-05   -8.30269374[74e-08   2.9153322360e-08   -4.8198506[56e-06   2.3113821[22e-06
result_run2.out:> IMC E_RR_total   -3.31402344[10e-05   -8.30269376[76e-08   2.9153322360e-08   -4.8198506[57e-06   2.3113821[21e-06
```

09e-05
10e-05

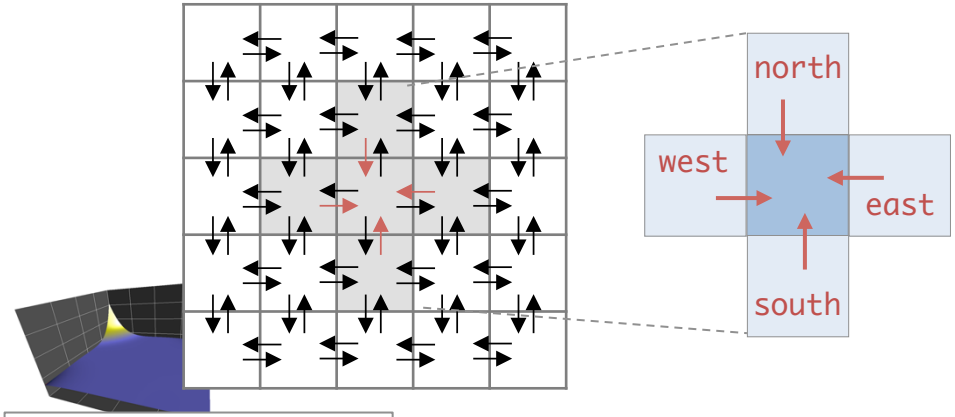
74e-08
76e-08

56e-06
57e-06

22e-06
21e-06

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, …);
  }
}
```

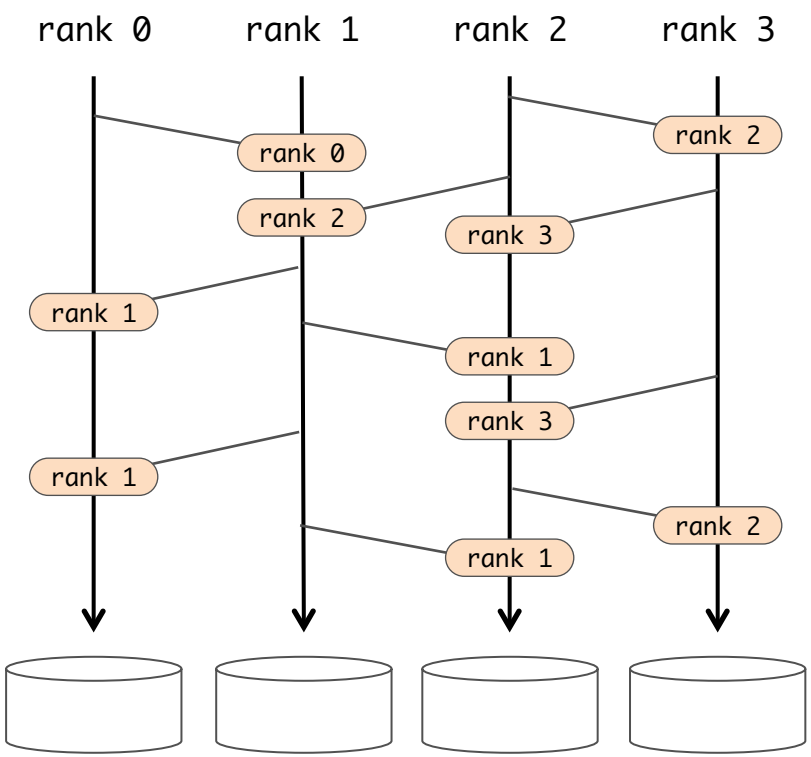
Source of MPI non-determinism

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

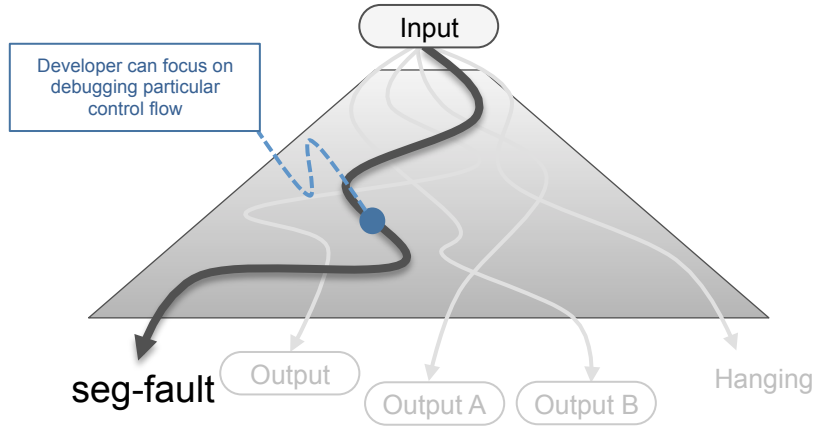
State-of-the-art approach: Record-and-replay

Record-and-replay



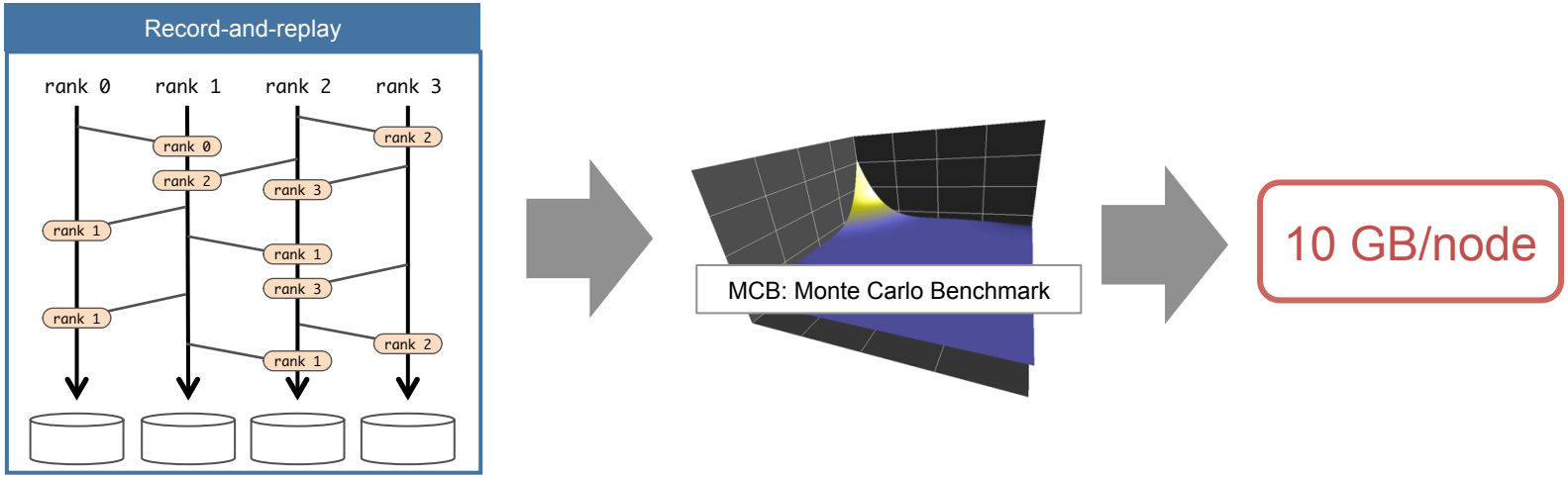
- Traces, records message receive orders in a run, and replays the orders in successive runs for debugging
  - Record-and-replay can reproduce a target control flow
  - Developers can focus on debugging a particular control flow

Debugging a particular control flow in replay



Record-and-replay won't work at scale

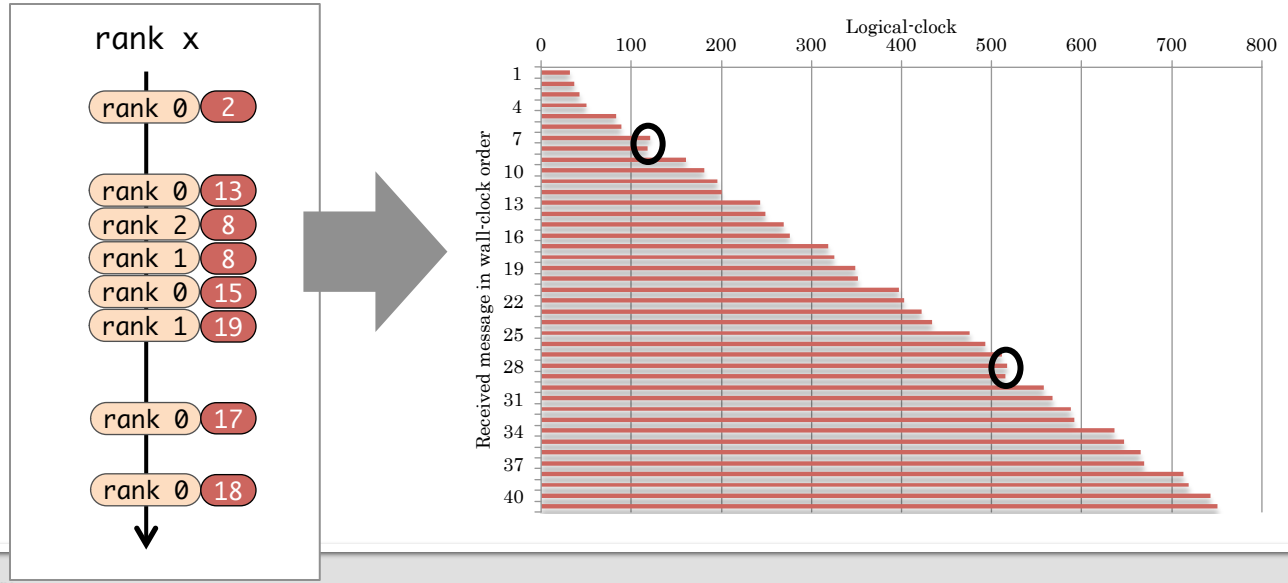
- Record-and-replay produces large amount of recording data
  - Over "**10 GB/node**" for 24 hours in MCB
- For scalable record-replay with low overhead, the record data must fit into local memory, but capacity is limited
  - Storing in shared/parallel file system is not scalable approach



Challenges Record size reduction for scalable record-replay

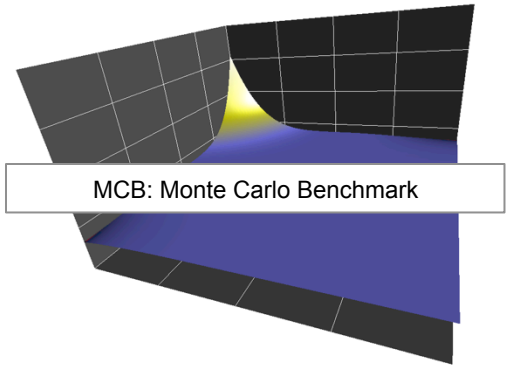
Proposal: Clock Delta Compression (CDC)

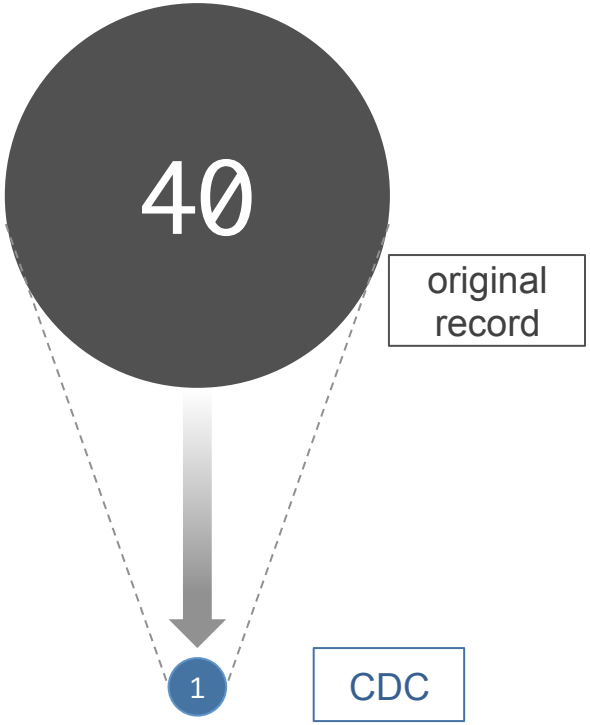
- Putting logical-clock (Lamport clock) into each MPI message
- Actual message receive orders (i.e. wall-clock orders) are very similar to logical clock orders in each MPI rank
  - MPI messages are received in almost monotonically increasing logical-clock order
- CDC records only the order differences 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





Outline

- Background
- General record-and-replay
- CDC: Clock delta compression
- Implementation
- Evaluation
- Conclusion

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

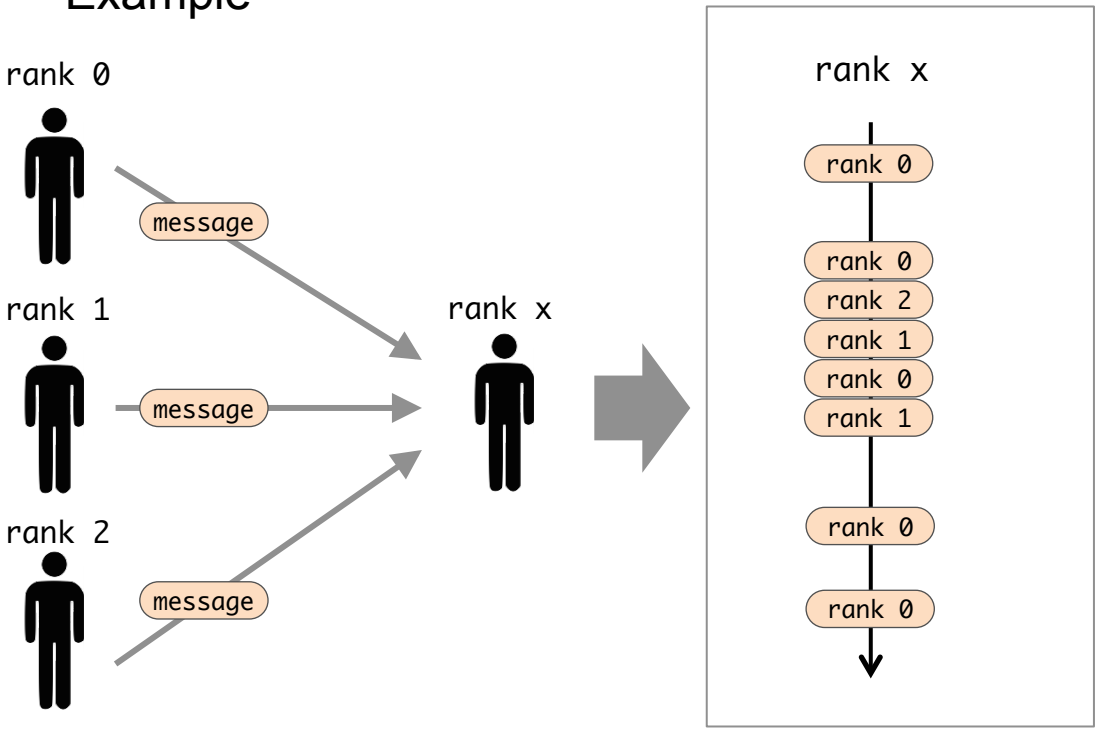
	Wait family	Test family
single	MPI_Wait	MPI_Test
any	MPI_Waitany	MPI_Testany
some	MPI_Waitsome	MPI_Testsome
all	MPI_Waitall	MPI_Testall

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

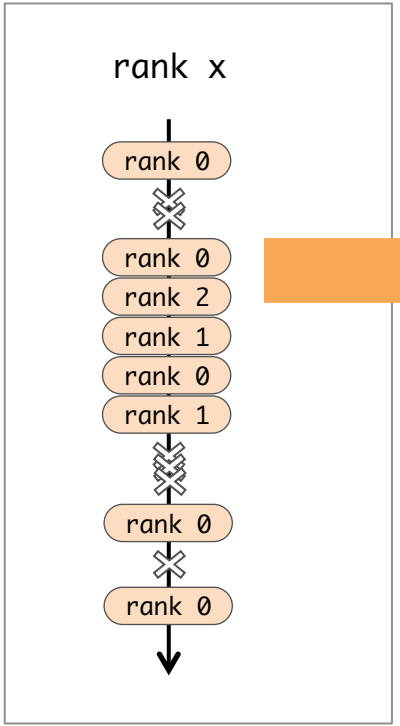


Necessary values for correct replay

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



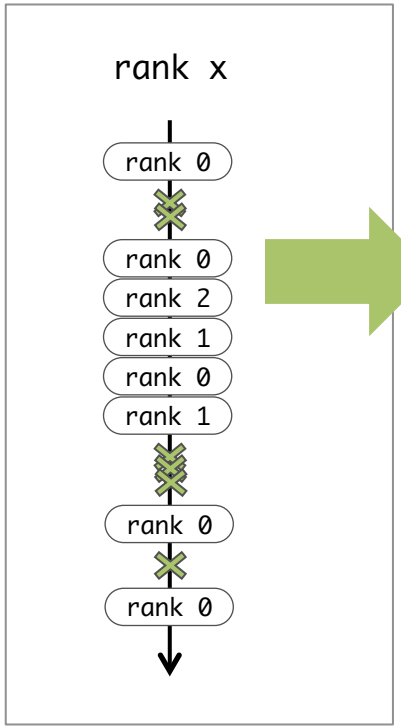
count	flag	rank
--	1	0
2	0	--
		0
		2
--	1	1
--	1	0
--	1	1
3	0	--
--	1	0
--	0	--
1	1	0

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

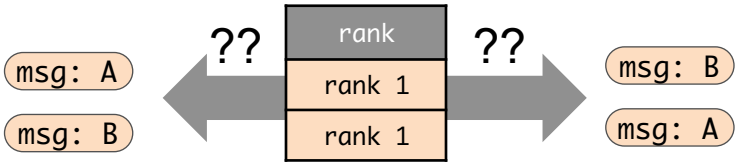
- rank
  - Who send the messages?
- count & flag
  - For MPI_Test family
    - flag: Matched or unmatched ?
    - count: How many time unmatched ?
- id
  - For application-level out-of-order
- with_next
  - For matching some/all functions

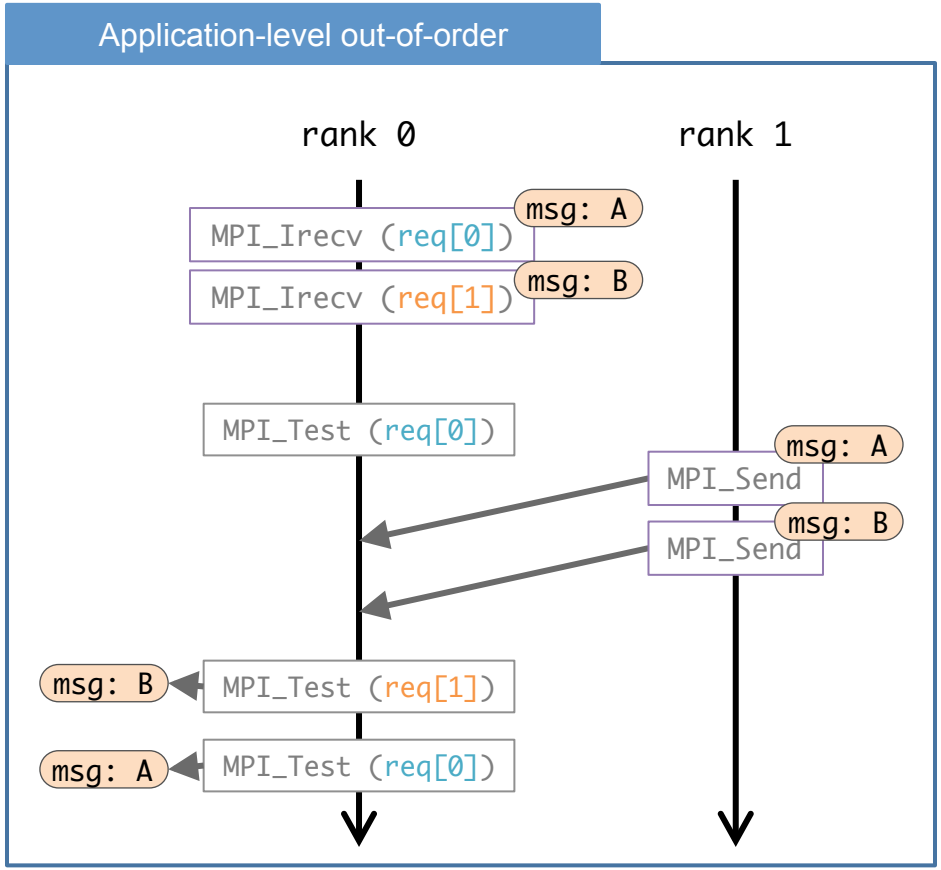


count	flag	rank
--	1	0
2	0	--
--	1	0
--	1	2
--	1	1
--	1	0
--	1	1
3	0	--
--	1	0
--	0	--
1	1	0

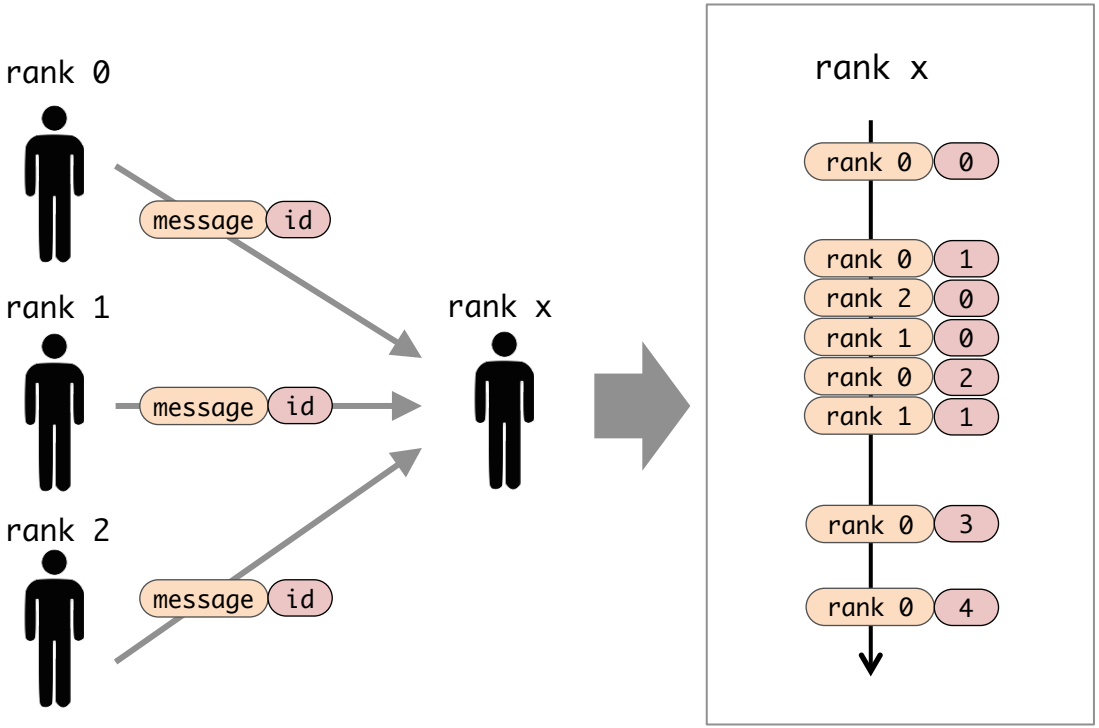
Application-level out-of-order

- MPI guarantees that any two communications executed by a process are ordered
  - Send: A ➔ B
  - Recv: A ➔ 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 A ➔ B and B ➔ A





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

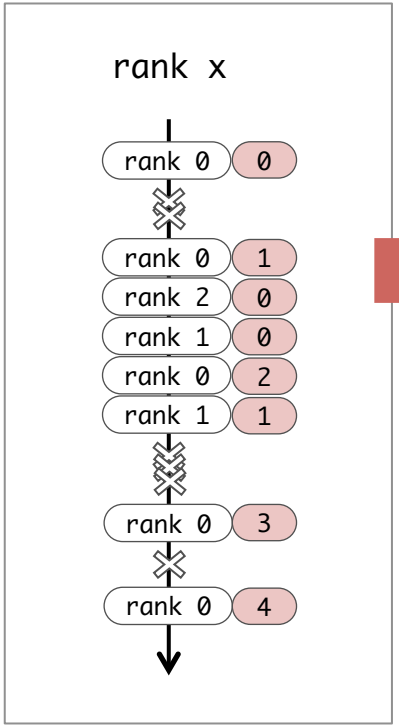


Necessary values for correct replay

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



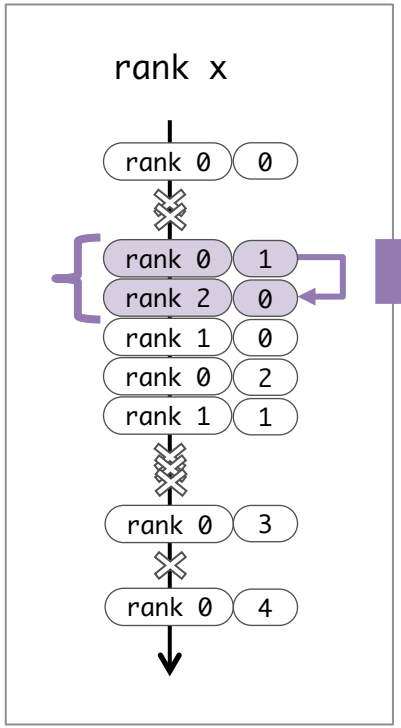
count	flag	rank	id
--	1	0	0
2	0	--	--
			1
			0
--	1	1	0
--	1	0	2
--	1	1	1
3	0	--	--
--	1	0	3
--	0	--	--
1	1	0	4

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

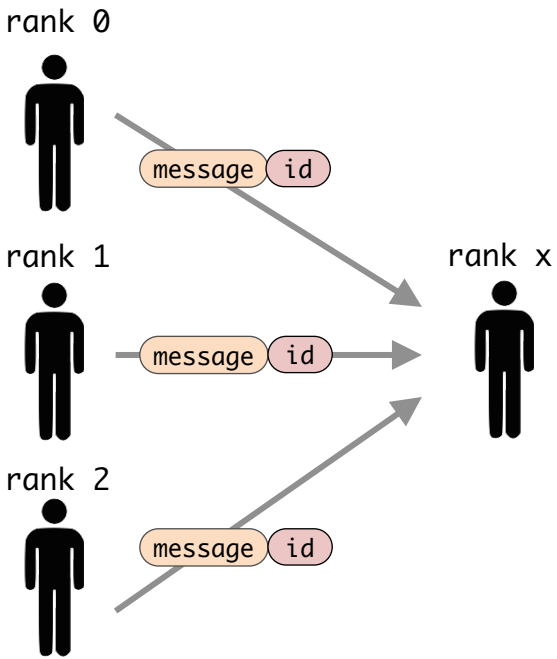
- rank
  - Who send the messages?
- count & flag
  - For MPI_Test family
    - flag: Matched or unmatched ?
    - count: How many time unmatched ?
- id
  - For application-level out-of-order
- with_next
  - For matching some/all functions

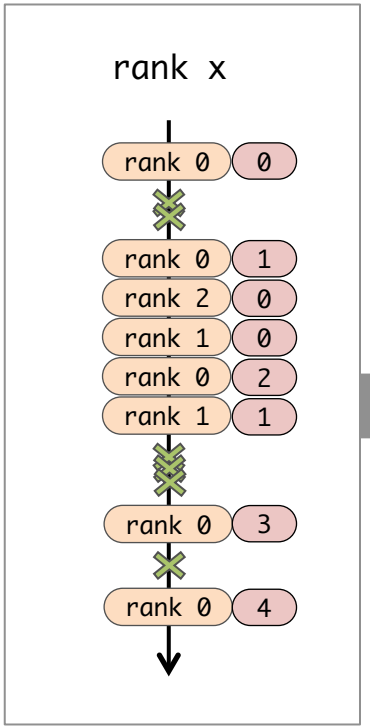


count	flag	rank	id	with_next
--	1	0	0	0
2	0	--	--	--
				1
				0
--	1	1	0	0
--	1	0	2	0
--	1	1	1	0
3	0	--	--	--
--	1	0	3	0
--	0	--	--	--
1	1	0	4	0

Necessary values for correct replay

- Example





event = 5 values

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
--	0	--	--	--
1	1	0	4	0

11 events

55 values

CDC: Clock delta compression

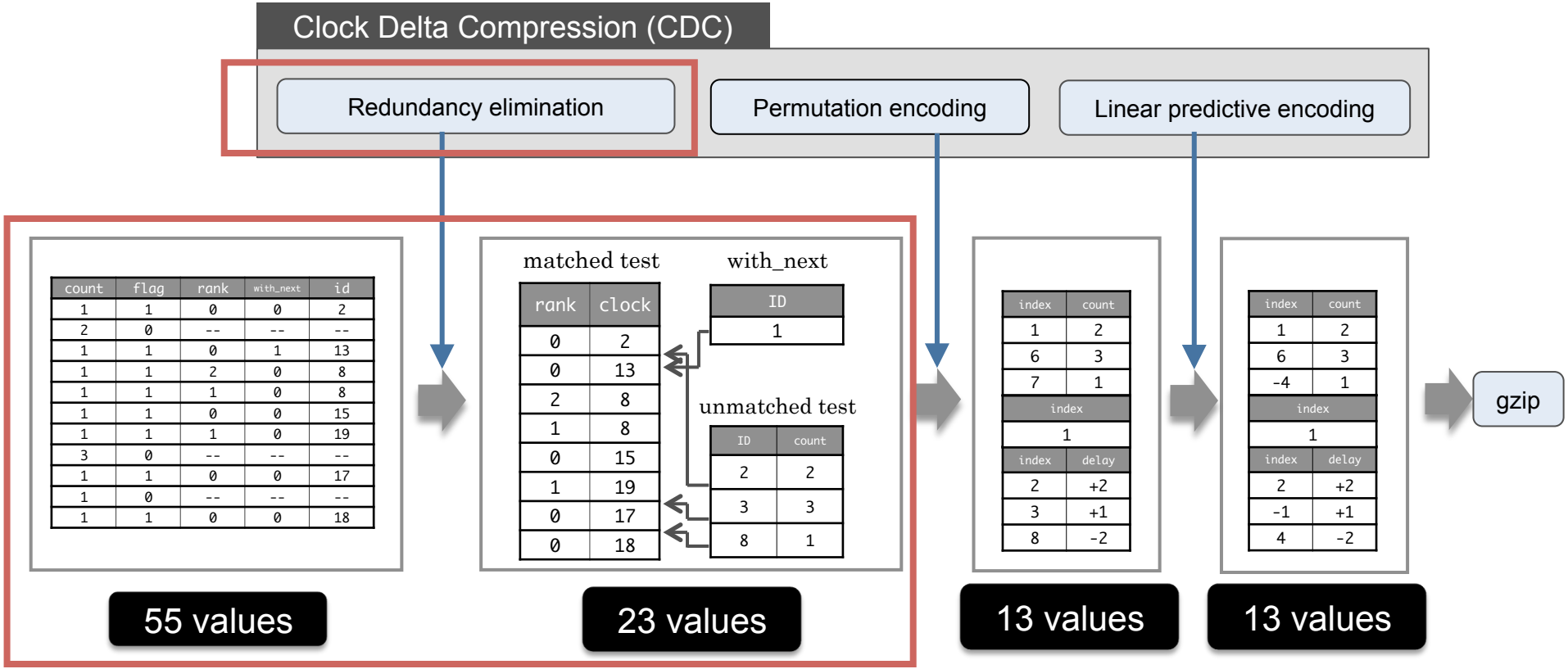
Clock Delta Compression (CDC)

- 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



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

matched table

index	rank	id
1:	0	0
2:	0	1
3:	2	0
4:	1	0
5:	0	2
6:	1	1
7:	0	3
8:	0	4

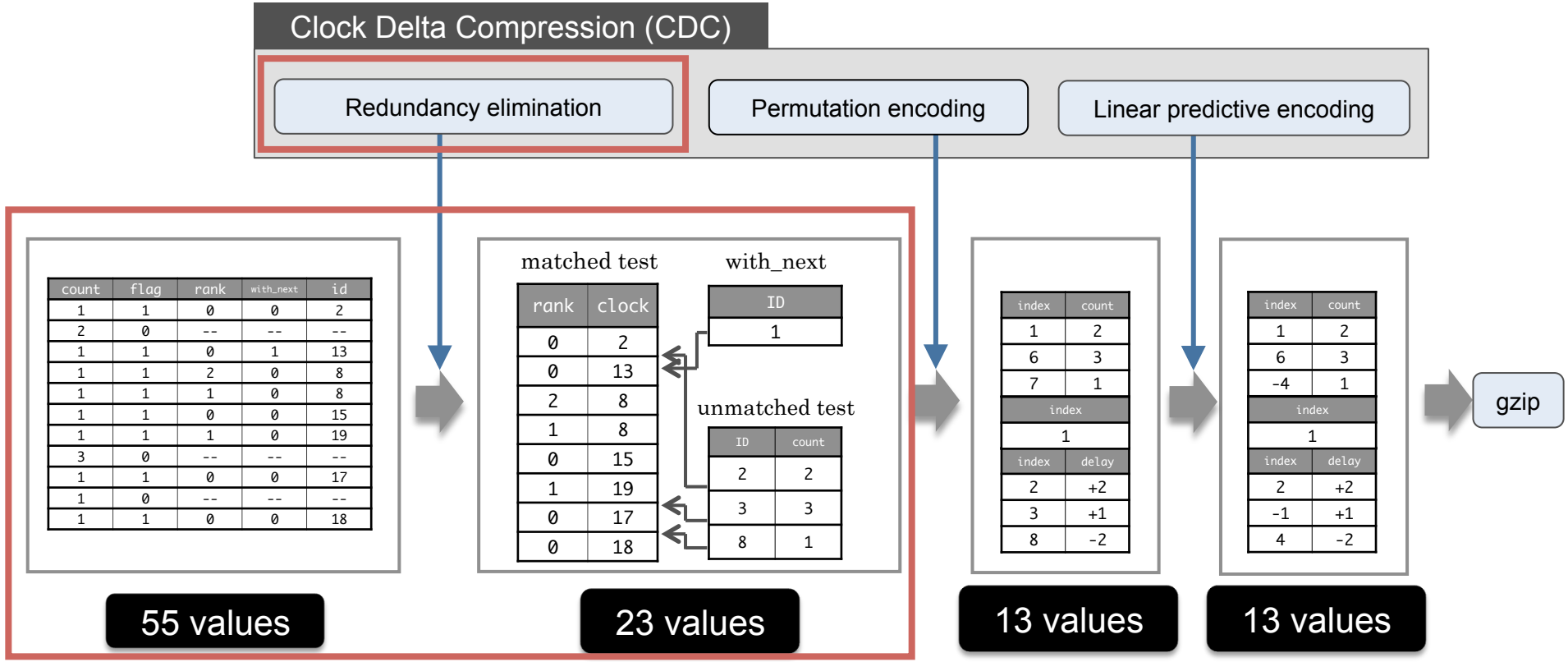
with_next table

index
2

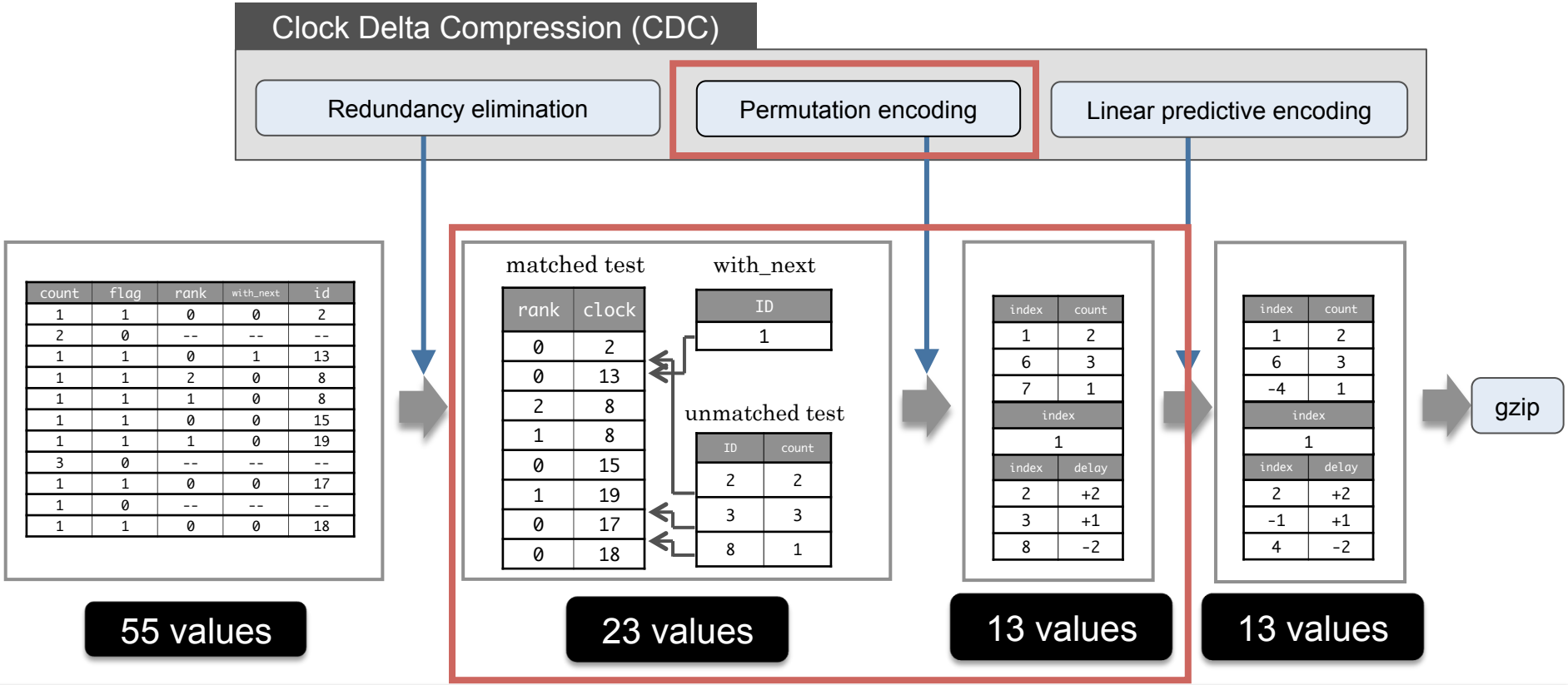
unmatched table

index	count
2	2
7	3
8	1

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

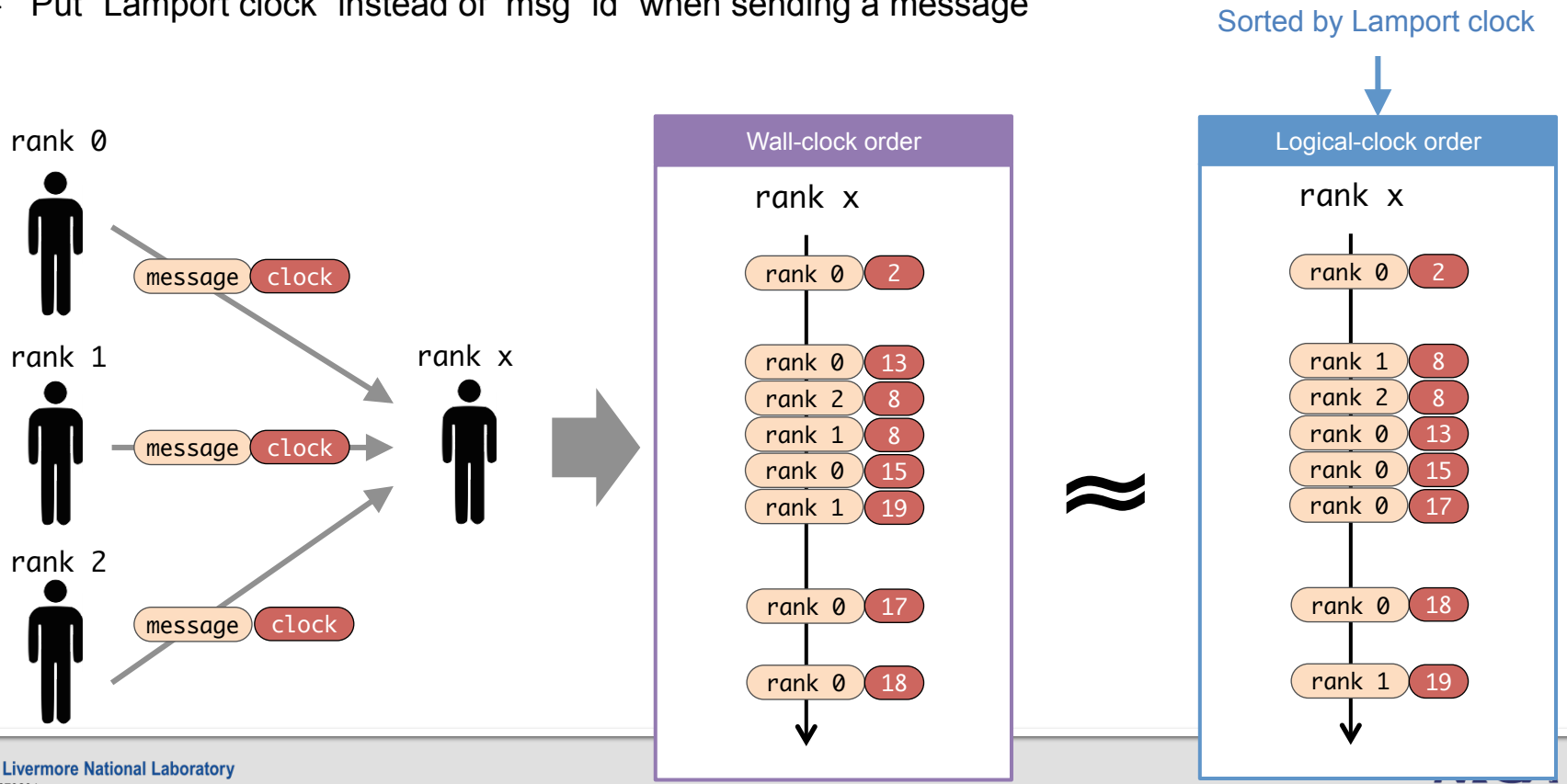
index	count
1	2
6	3
-4	1

index
1

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

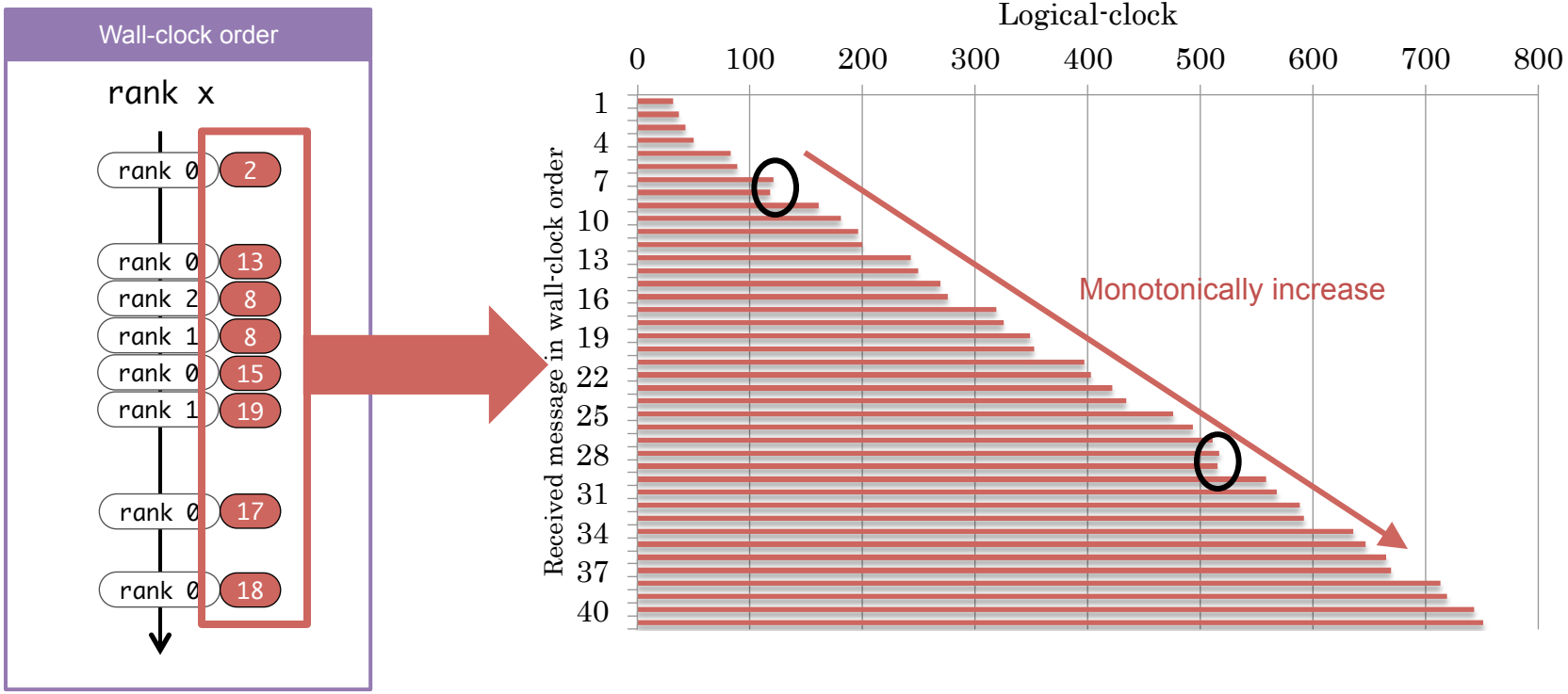
Key observation in communications

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



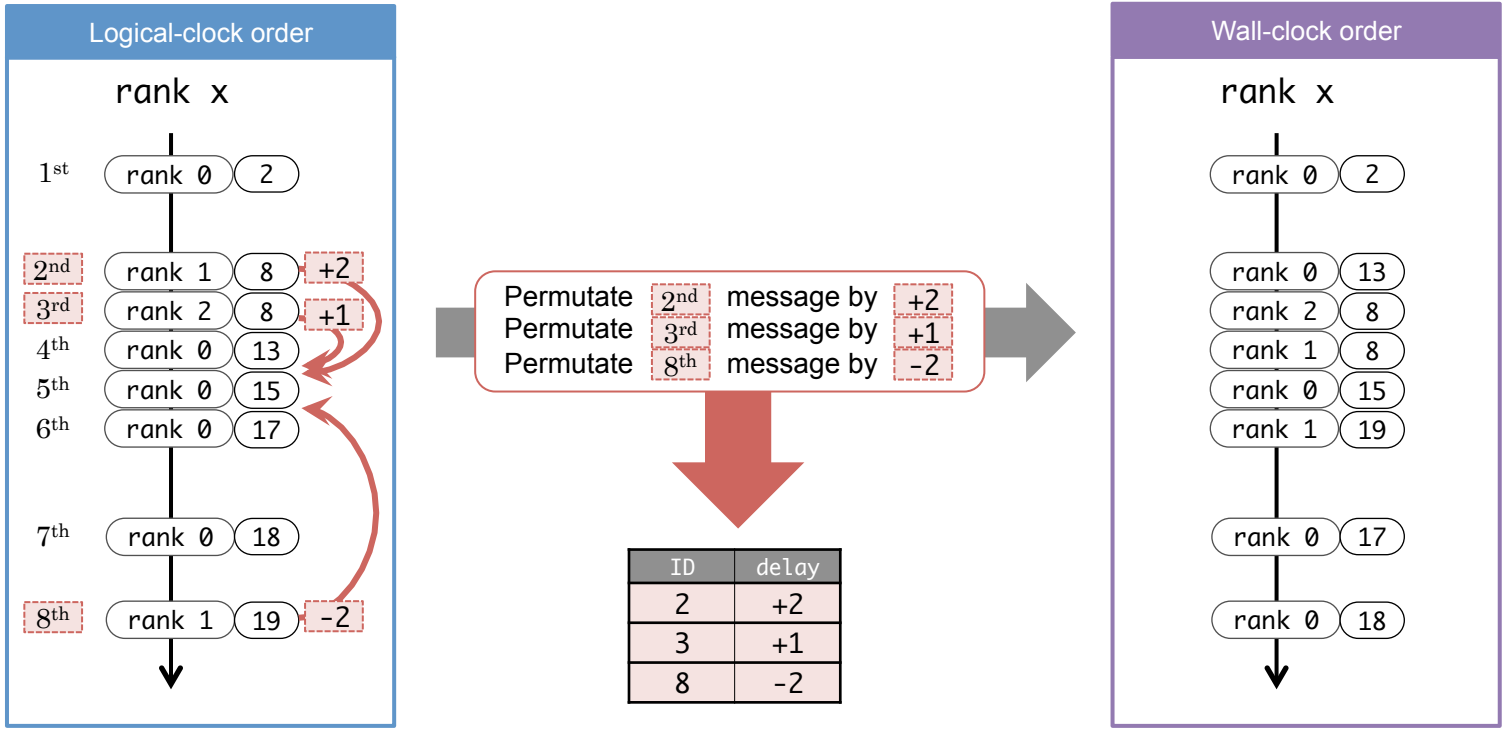
Case study: Received logical-clock values in MCB

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



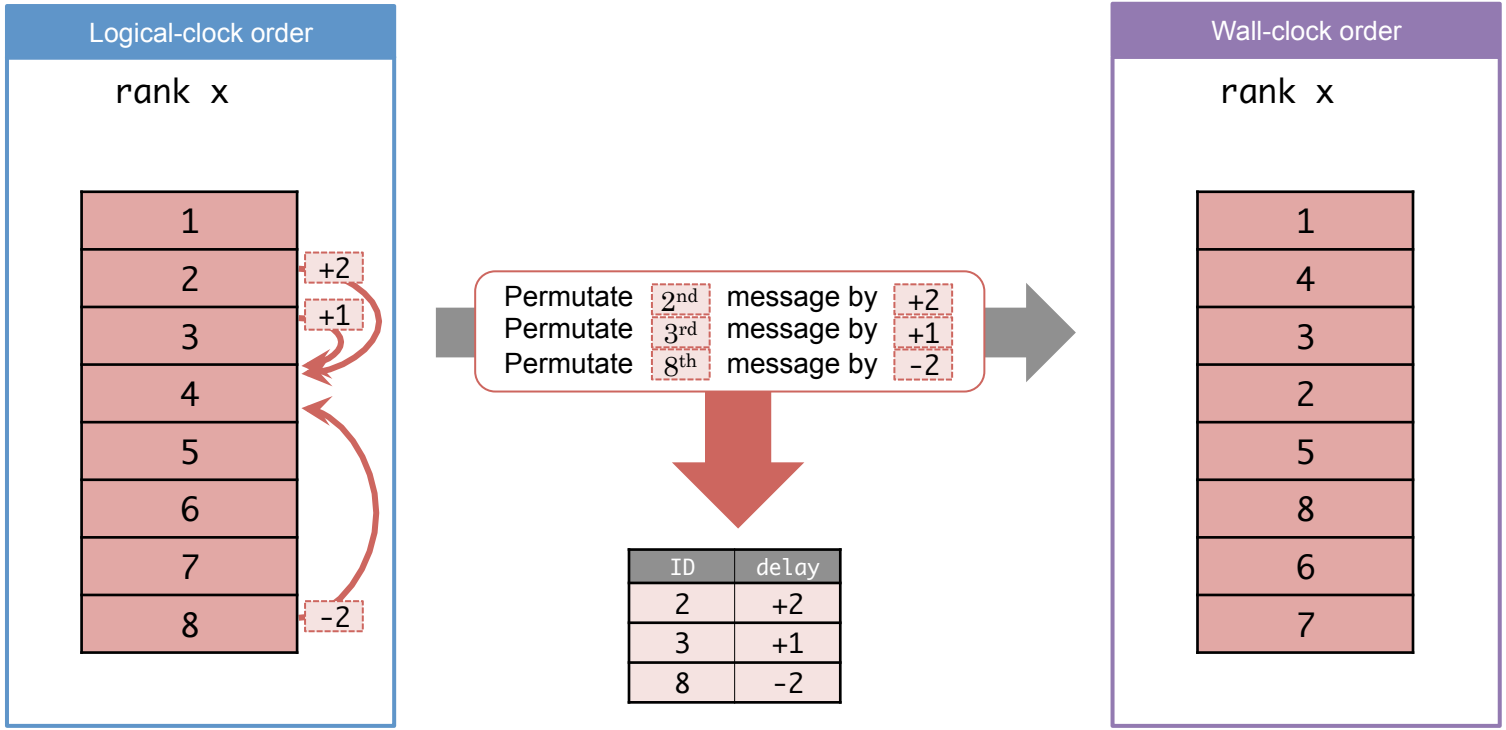
Permutation encoding

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

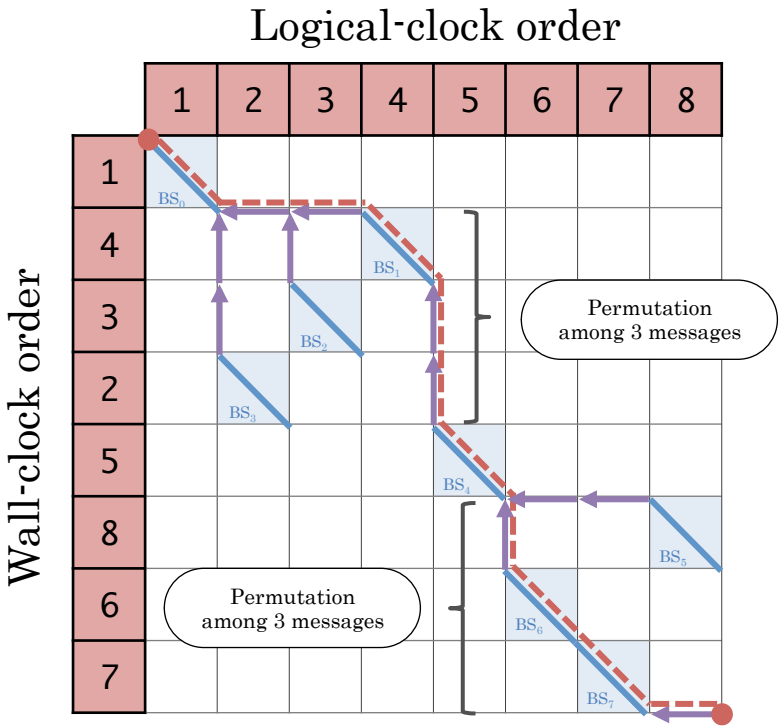


Permutation encoding

- 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

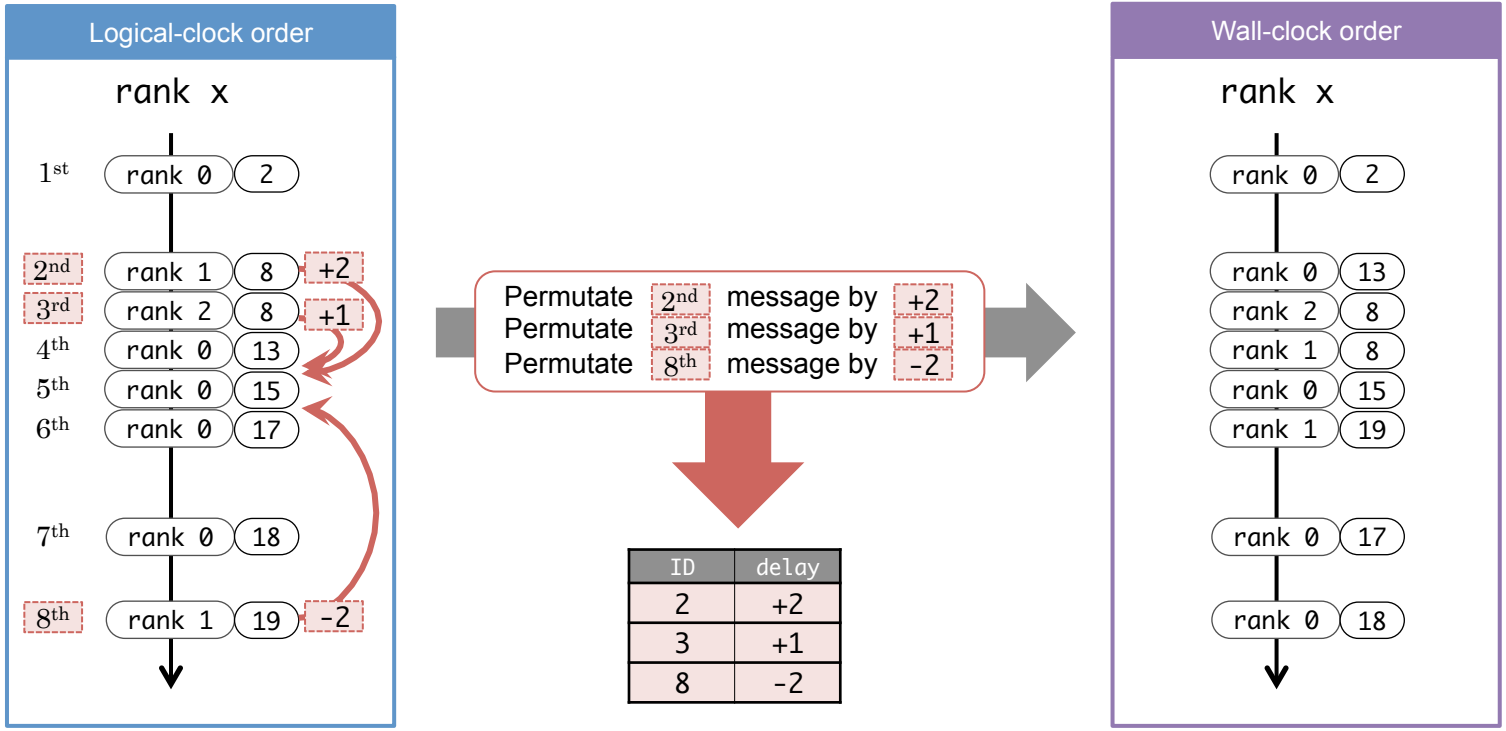


- Edit distance algorithm
  - Compute similarity between two strings
    - Wall-clock order
    - Logical-clock order
  - Time complexity: $O(N^2)$
    - N: length of the strings

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

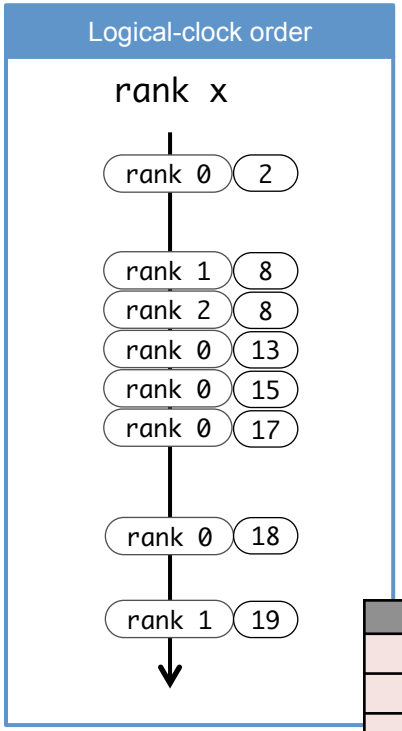
  ➔ Time complexity: O(N+D)
  - N: Length of the strings
  - D: Edit distance

Why Logical-clock order is not recorded ?

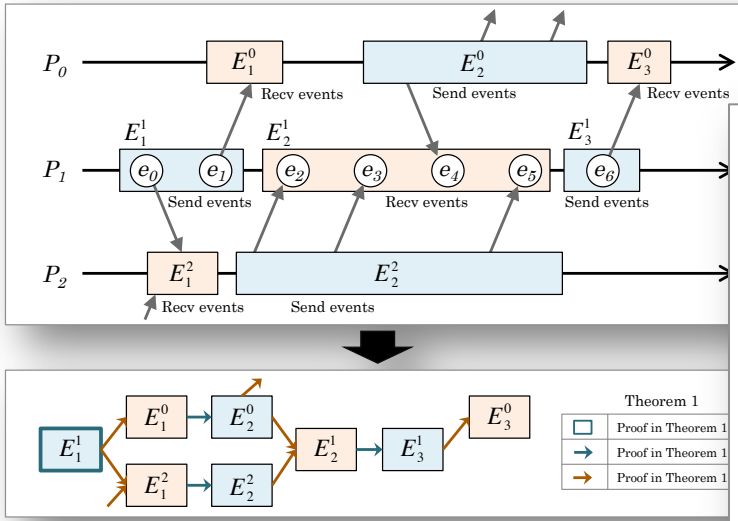


Logical-clock order is reproducible

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



ID	delay
2	+2
3	+1
8	-2



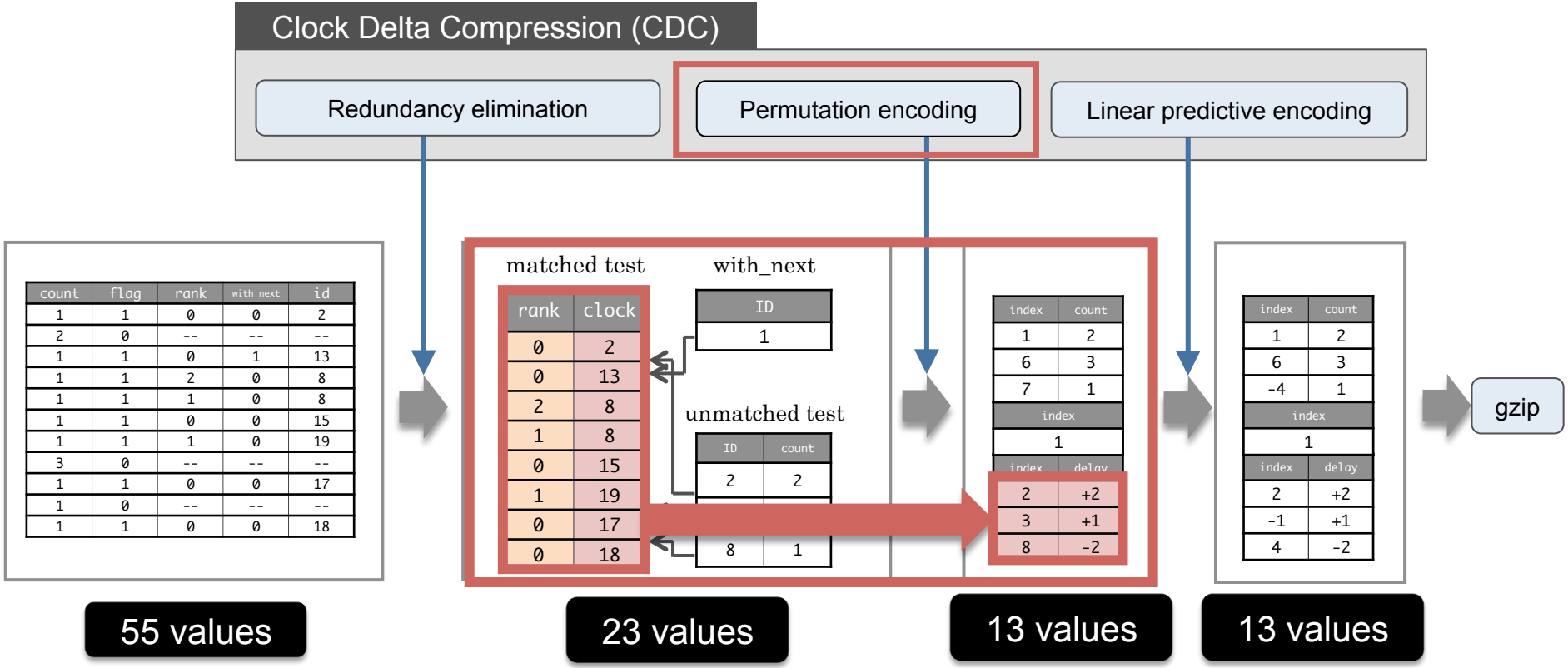
Theorem 1. CDC can correctly replay message events, that is, $\boldsymbol{E} = \hat{\boldsymbol{E}}$ where $\boldsymbol{E}$ and $\hat{\boldsymbol{E}}$ are ordered sets of events for a record and a replay mode.

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

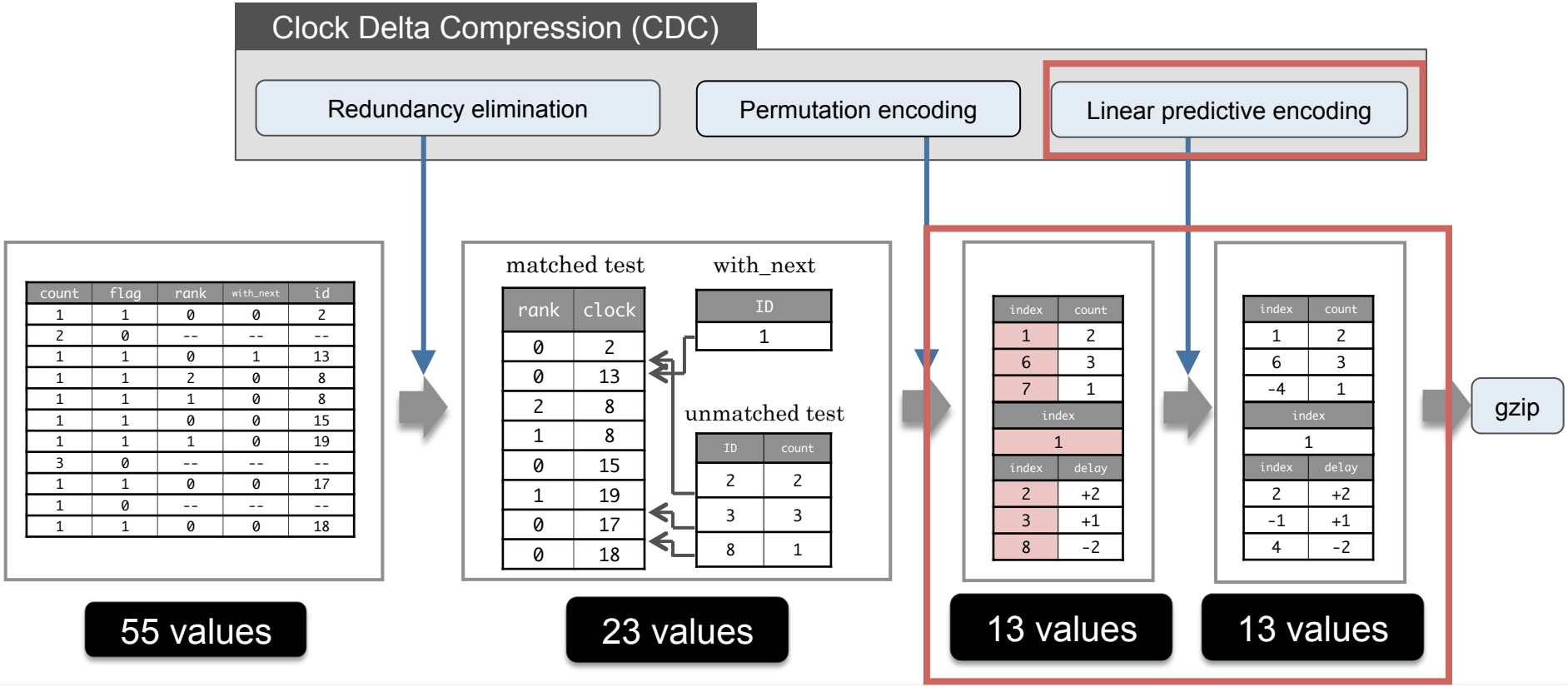
index	delay
2	+2
3	+1
8	-2

index	count
1	2
6	3
-4	1

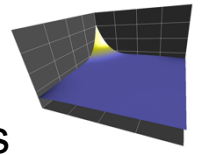
index
1

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

CDC: Clock delta compression



Case study: index values in MCB

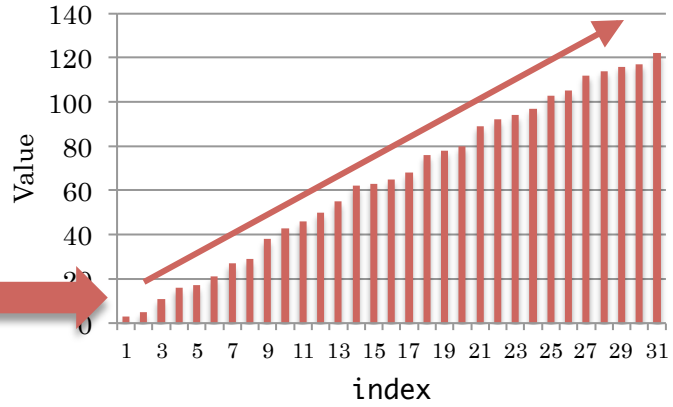


- Problem in the format: `index` values linearly increase as CDC records events
- Compression rate by gzip becomes worse as the table size increases
  - gzip encodes frequent sequence of bits into shorter bits
  - If we can encode these values into close to zero, gzip can give a high compression rate

index	count
1	2
6	3
7	1

index
1

index	delay
2	+2
3	
8	-2
⋮	



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, $\{e_1, e_2, \ldots, e_N\}$

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

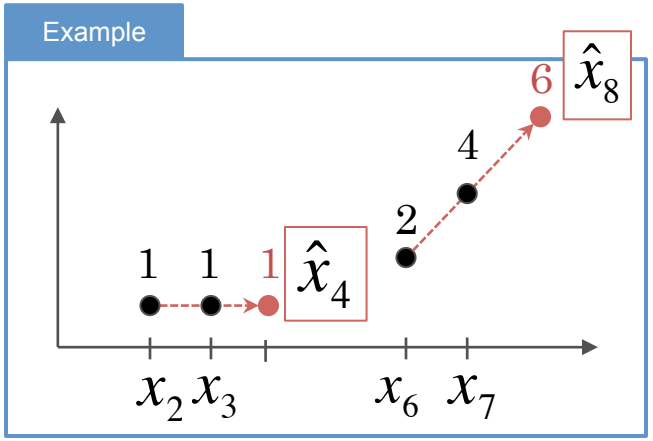
$$e_n = x_n - \hat{x}_n$$

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

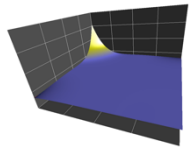
- Choice of p, and co-efficients, $\{a_1, a_2, \ldots, a_p\}$, affects accuracy of prediction
- In CDC, we predict x_n is on an extension of a line created by x_{n-1}, x_{n-2}

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

Example



Case study: Linear predictive encoding in MCB

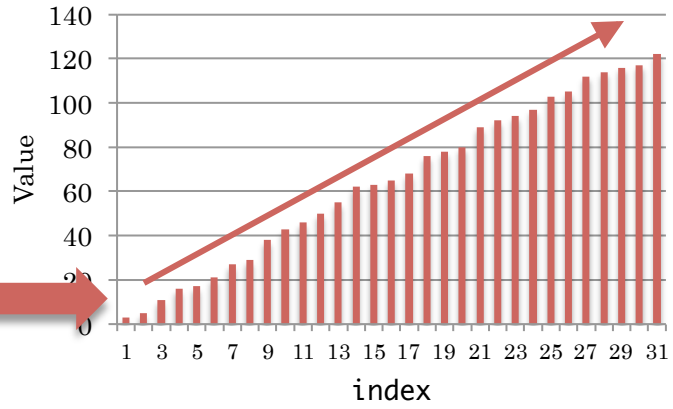


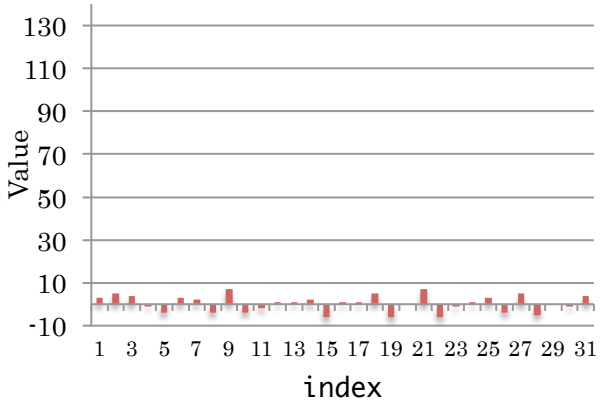
Linear predictive encoding

index	count
1	2
6	3
7	1

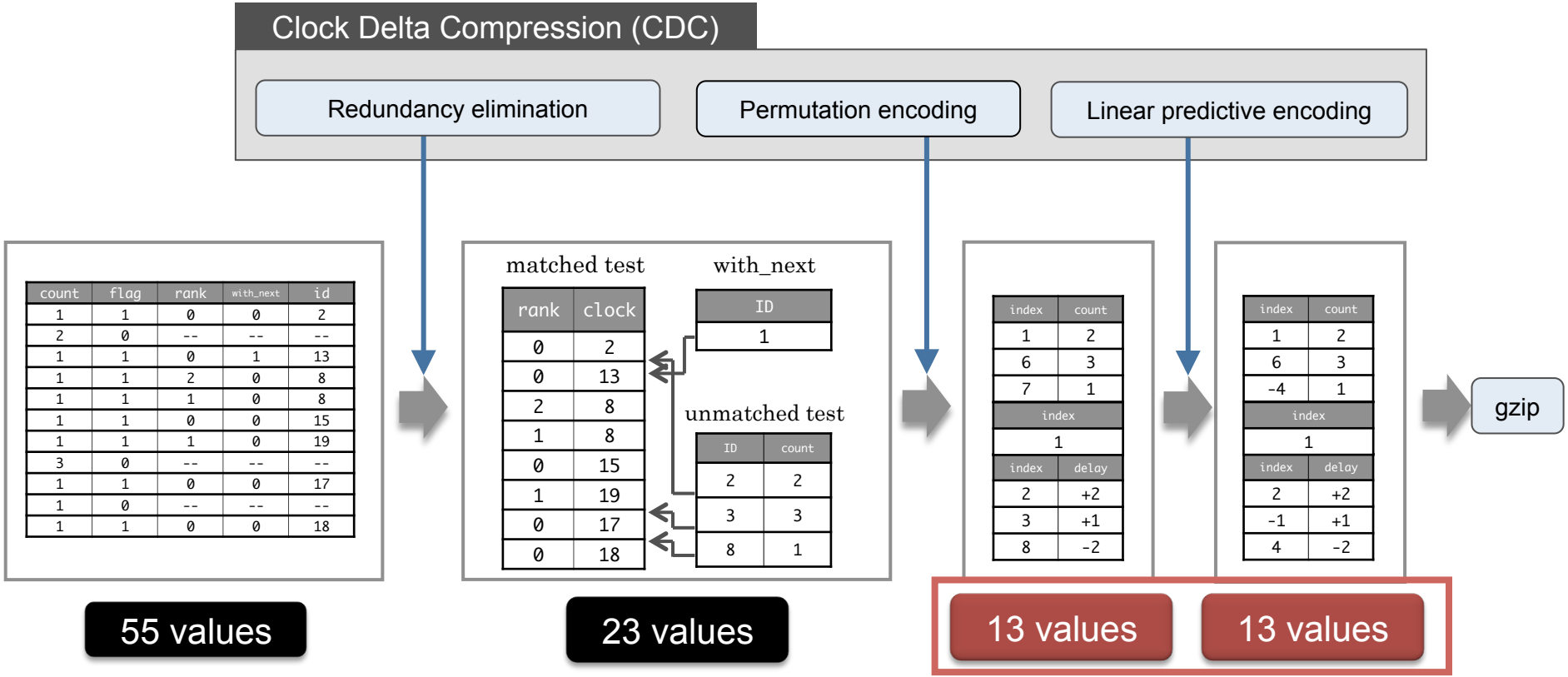
index
1

index	delay
2	+2
3	
8	-2





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

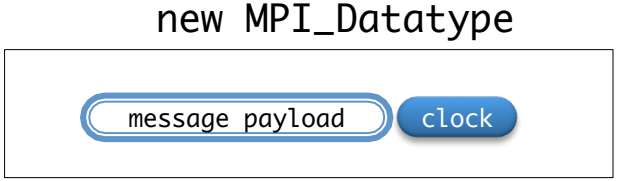
index	delay
2	+2
-1	+1
4	-2

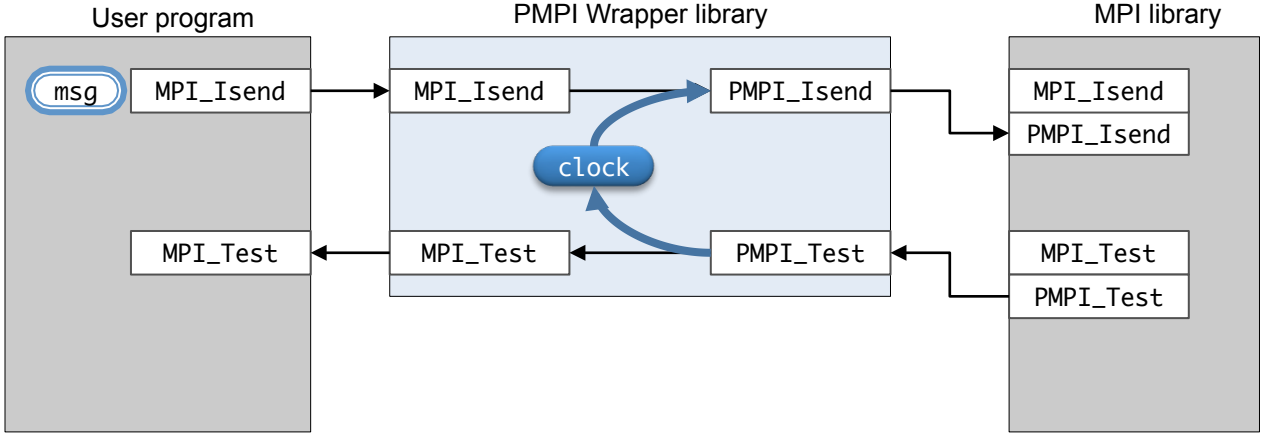
Outline

- Background
- General record-and-replay
- CDC: Clock delta compression
- Implementation
- Evaluation
- Conclusion

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

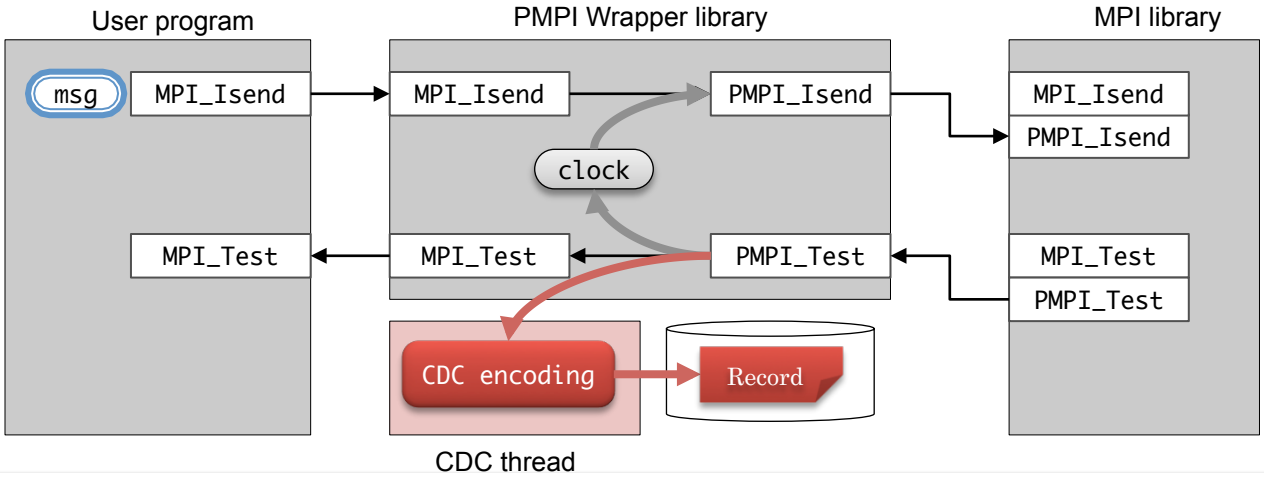




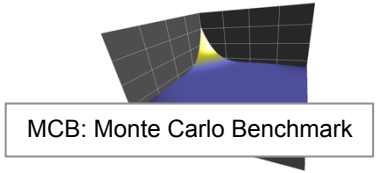
[1] M. Schulz, G. Bronevetsky, and B. R. Supinski. On the Performance of Transparent MPI Piggyback Messages. In Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 194–201, Berlin, Heidelberg, 2008. Springer-Verlag.

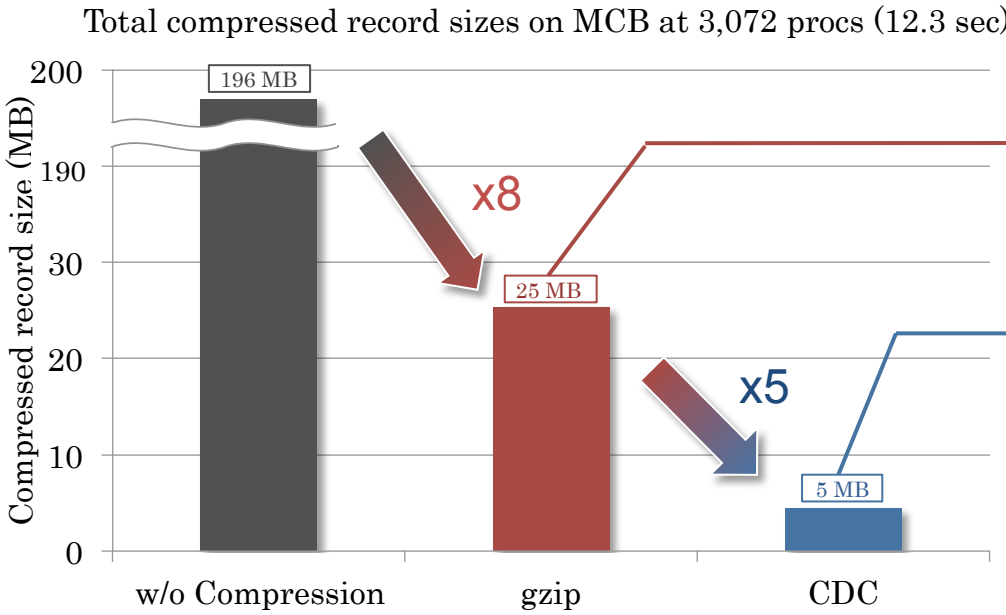
Asynchronous encoding

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



Compression improvement in MCB





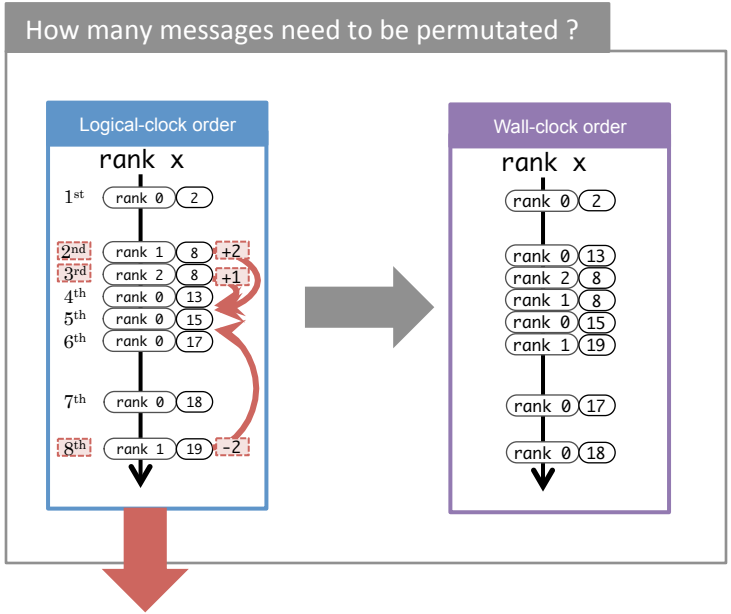
gzip itself can reduce the original format by 8x

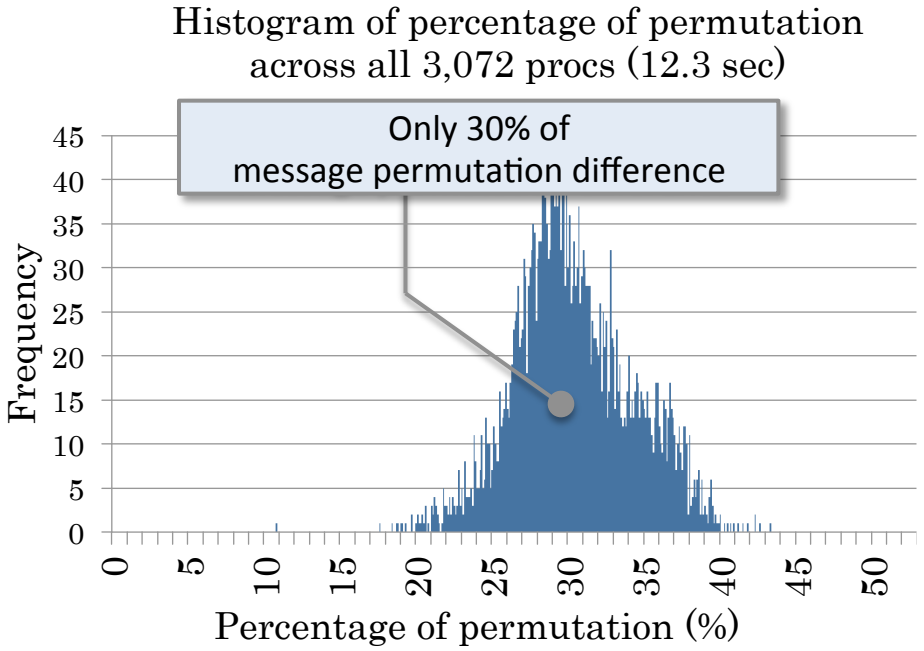
5x more reduction

- 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

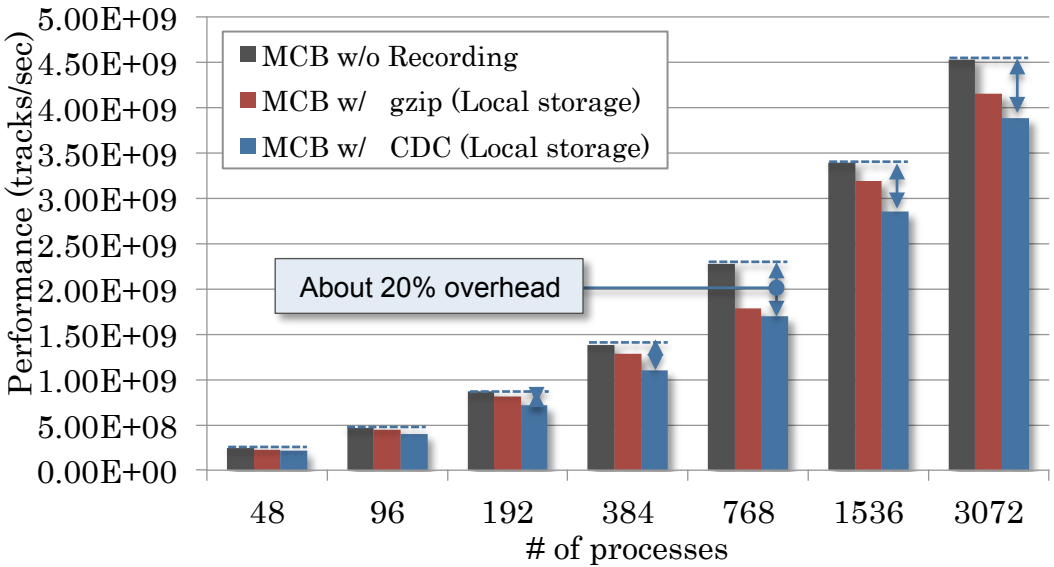




3 messages out of 8 = 37% of similarity

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

Conclusion

- MPI non-determinism is problematic for debugging
- Record-and-replay solve the problem
  - However, it produces large amount of data
  - This hampers scalability of the tool

- CDC: Clock Delta Compression
  - Only record difference between wall-clock order and logical-clock order
    - Logical-clock order is always reproducible

- With CDC, the applications can be scale even if recording
  - All record data can be fit into local memory for longer time

- Future work
  - Reduce record size more by using more accurate Logical-clock and accurate prediction for LP encoding

Thanks !

Speaker:

Kento Sato (佐藤 賢斗)
Lawrence Livermore National Laboratory

https://kento.github.io
(The slides will be uploaded here)

Acknowledgement

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

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-679294).


Lawrence Livermore
National Laboratory