Non-Intrusively Avoiding Scaling Problems in and out of MPI Collectives

Hongbo Li, Zizhong Chen, Rajiv Gupta, and Min Xie

UC RIVERSIDE UNIVERSITY OF CALIFORNIA

NATIONAL UNIVERSITY OF DEFENSE TECHNOLOGY

May 21st, 2018

Outline

- Scaling Problem
- Avoidance Framework
- Evaluation
- Conclusion

Outline

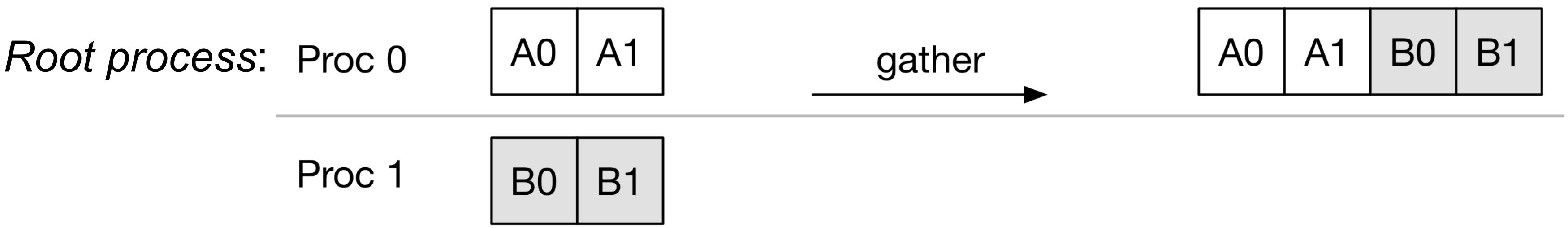
- Scaling Problem
- Avoidance Framework
- Evaluation
- Conclusion

Scaling Problem

- Scaling problem is a type of bug that occurs when the program runs at a large scale in terms of
  - the number of processes (P)
  - OR the input size
  - OR both

- They frequently arise with the use of MPI collectives as collective communication involves
  - a group of processes
  - and message size (input size)

An Example of MPI Collective



MPI_Gather using two processes ($P = 2$) with each transferring two elements $n = 2$.

Symbol	Meaning
n	Element count in one message
s	Size of the data type in bytes
P	Total number of processes

Scaling Problem

- The root cause of a scaling problem with the use of MPI collectives can be
  - inside MPI collectives
  - or outside MPI collectives

Inside MPI

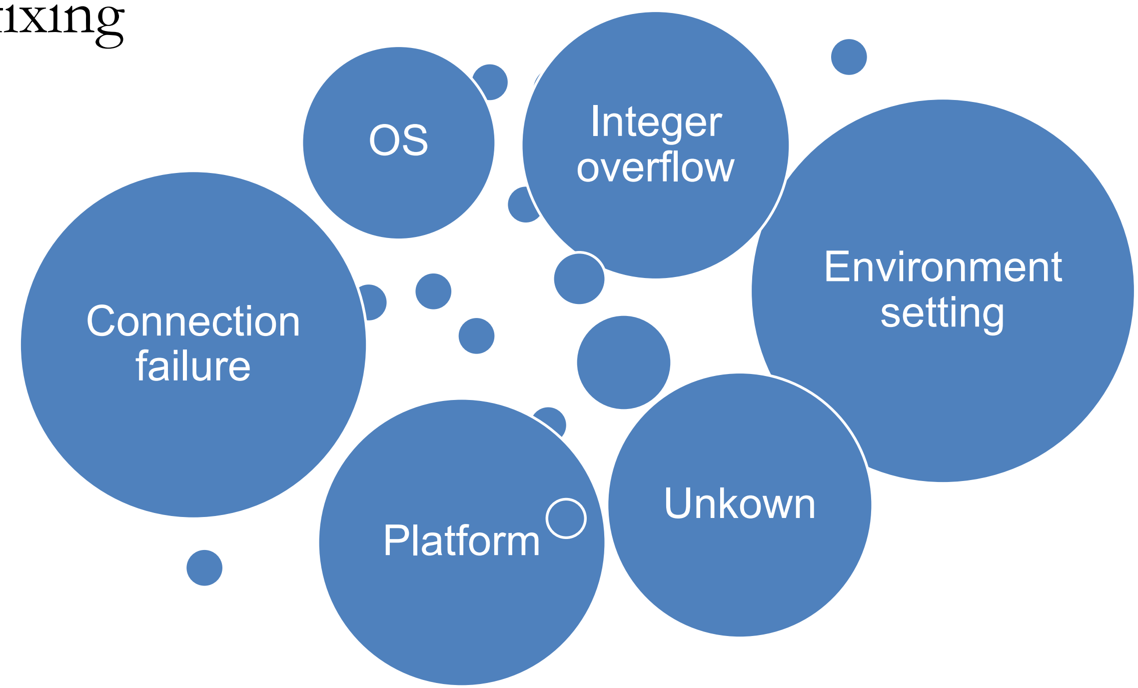
- Many scaling problems are challenging to deal with
  - They escape the testing in the development phase
- It takes days and months to wait for an official fix
  - Difficulty exists in bug reproduction, root-cause diagnosis, and fixing

Scaling problems reported online.

Prob.	Collective	MPI library	Type	Effect	Scale (P, M)	Root cause (inside MPI)
1	MPI_Gather	OpenMPI 1.4.3	3	H	(64, 4KB)	Environment setting dependency
2	MPI_Alltoall	OpenMPI 1.4.3	3	H	(44, 4MB)	Environment setting dependency
3	MPI_Allgather	OpenMPI 1.4.3	3	H	(64, 4MB)	–
4	MPI_Alltoallv	OpenMPI 1.7	3	H	(96, 512KB)	Network connection failure
5	MPI_Allgather	MPICH 2	3	D	$P \cdot M >$ INT_MAX	Integer overflow in MPI
6	MPI_Send + Recv	Intel MPI 5.1.2	2	H	(2, 64KB)	OS (ubuntu) dependency
7	MPI_Bcast	Intel MPI 5.1.2	2 or 3	H	(2, 64KB)	Unknown to developers
8	MPI_Bcast	Intel MPI 2017	2 or 3	H	(–, 16KB)	Platform (KNL & BDW) dependency

Inside MPI

- Many scaling problems are challenging to deal with
  - They escape the testing in the development phase
- It takes days and months to wait for an official fix
  - Difficulty exists in bug reproduction, root-cause diagnosis, and fixing



Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of **irregular collectives** can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[0] * s$

Each process' *sendbuf*

0

Root's *recvbuf*

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is not corrupted.

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of **irregular collectives** can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[0] * s$

Each process' *sendbuf*

Root's *recvbuf*

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is not corrupted.

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of **irregular collectives** can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[1] * s$

Each process' *sendbuf*

Root's *recvbuf*

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is not corrupted.

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of **irregular collectives** can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[1] * s$

Each process' *sendbuf*

Root's *recvbuf*

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is not corrupted.

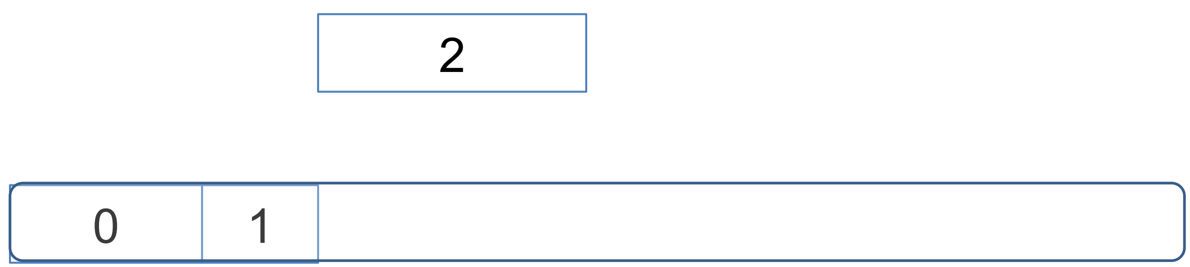
Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of **irregular collectives** can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[2] * s$

Each process' *sendbuf*

Root's *recvbuf*



In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is not corrupted.

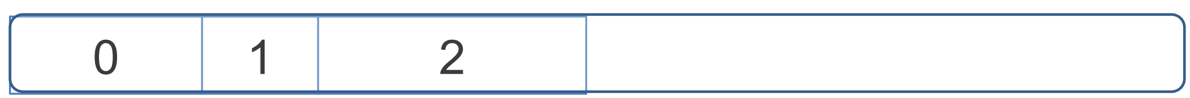
Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of **irregular collectives** can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[2] * s$

Each process' *sendbuf*

Root's *recvbuf*



In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is not corrupted.

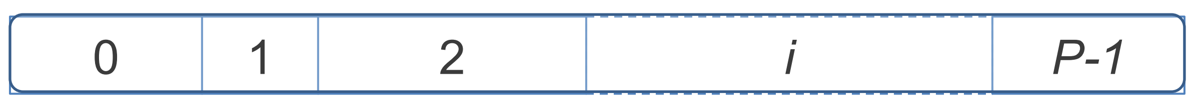
Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of **irregular collectives** can be easily corrupted by integer overflow

Calculate address:

Each process' *sendbuf*

Root's *recvbuf*



In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is not corrupted.

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[0] * s$

0

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[0] * s$

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[1] * s$

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

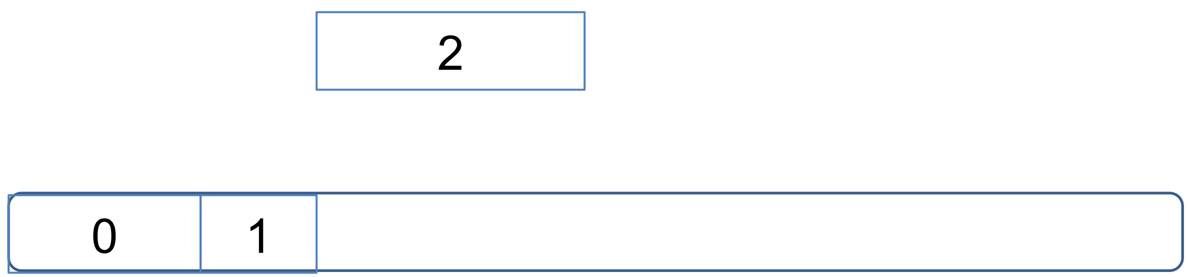
Calculate address: $recvbuf + displs[1] * s$

In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

Calculate address: $recvbuf + \color{orange}{displs[2]} * s$

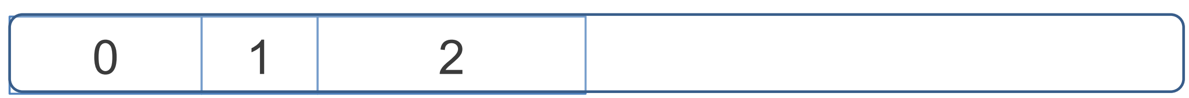


In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[2] * s$



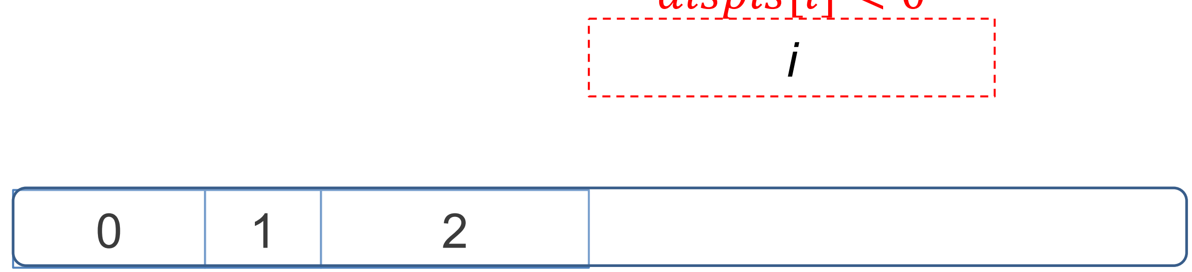
In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[i] * s$

$displs[i] < 0$



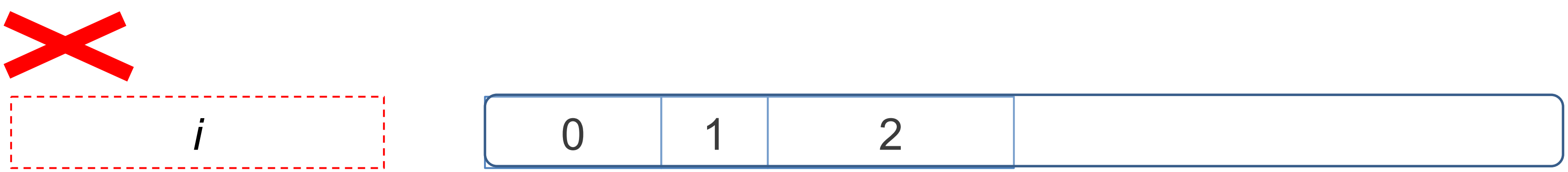
In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

Calculate address: $recvbuf + displs[i] * s$

$displs[i] < 0$



In MPI_Gatherv, the root process calculate addresses for the incoming messages when *displs* is corrupted

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

- For MPI_Gatherv, the number of elements (N) received by the root process satisfies

$$N < displs[P-1] + INT\_MAX$$

$$\rightarrow \boldsymbol{N < 2\ INT\_MAX}$$

- For MPI_Gather (a regular collective),

$$\boldsymbol{N \leq P\ INT\_MAX}$$

Outside MPI

- In the user code, displacement array ***displs*** (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow

- For MPI_Gatherv, the number of elements (N) received by the root process satisfies

$$N < displs[P-1] + INT\_MAX$$
$$\rightarrow \boldsymbol{N < 2\ INT\_MAX}$$

Huge gap: $\boldsymbol{\frac{2}{P}}$

- For MPI_Gather (a regular collective),

$$\boldsymbol{N \le P\ INT\_MAX}$$

Outside MPI

- Irregular collectives' limitation due to displacement array *displs* of data type *C int*

- Replace *int* with *long long int* ?
  - Discussed yet never done --- backward compatibility

An immediate remedy is in need!

Outline

- Scaling Problem
- Avoidance Framework
- Evaluation
- Conclusion

Avoidance

```
int MPI_Collective (…) {
        if (true == check_problem_trigger)
                MPI_Collective_Fix (…)        // invoke our fix
        else
                PMPI_Collective (…)           // invoke the default
}
```

Scaling problem's trigger

Workaround strategy

Trigger (1) [Outside MPI]

- Irregular collectives' limitation's trigger is

$$displs[i] < 0$$

Trigger (2) [Inside MPI]

- **Users perform testing**
  - It tells users if there is a scaling problem
  - It also tells at what scale the problem occurs

- Do users really need a fancy supercomputer to perform testing?

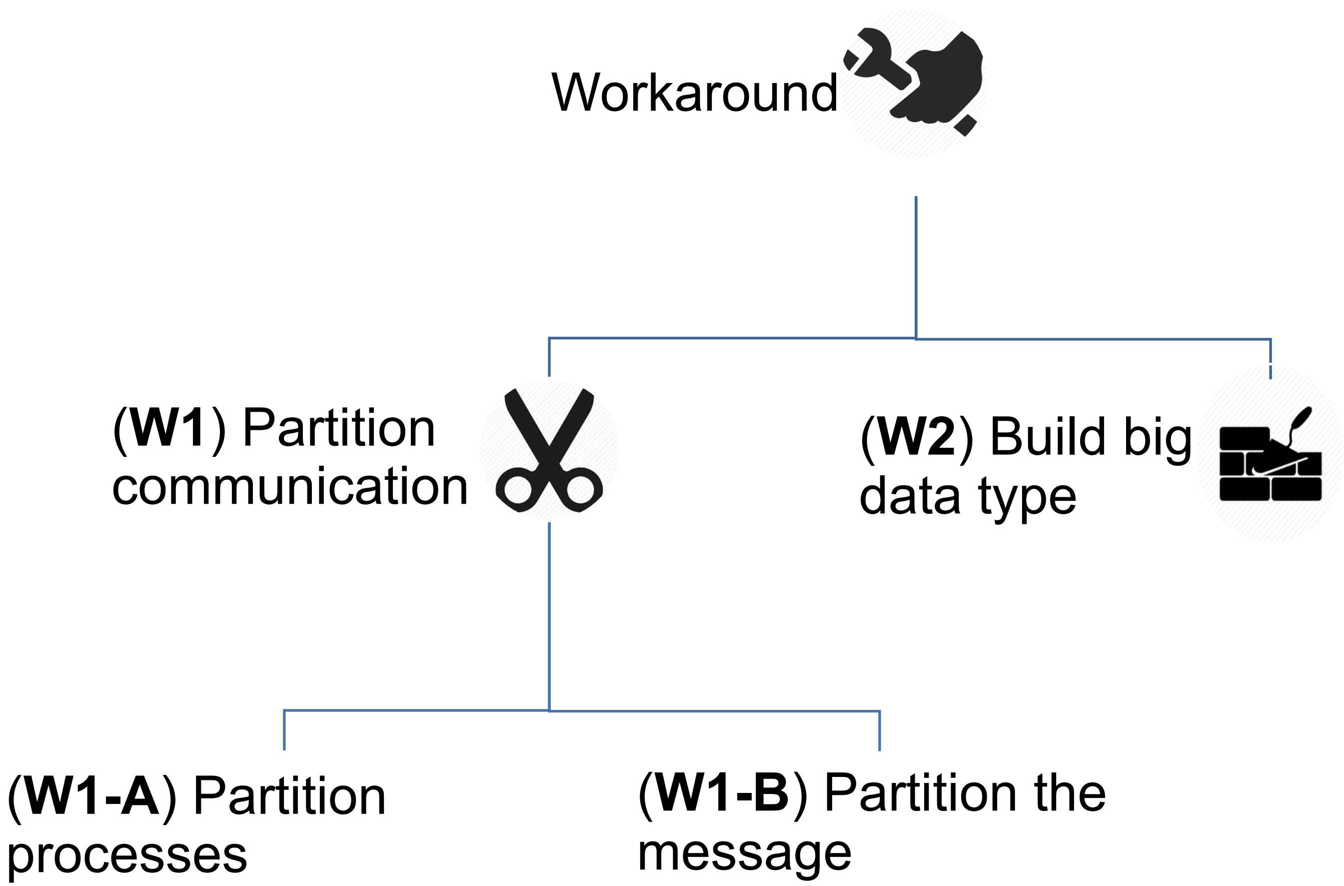
Not Necessary!

Trigger (2) [Inside MPI]

- **User side testing:** users manifest potential scaling problems of MPI routines of their interest
  - It tells users if there is a scaling problem
  - It also tells at what scale the problem occurs

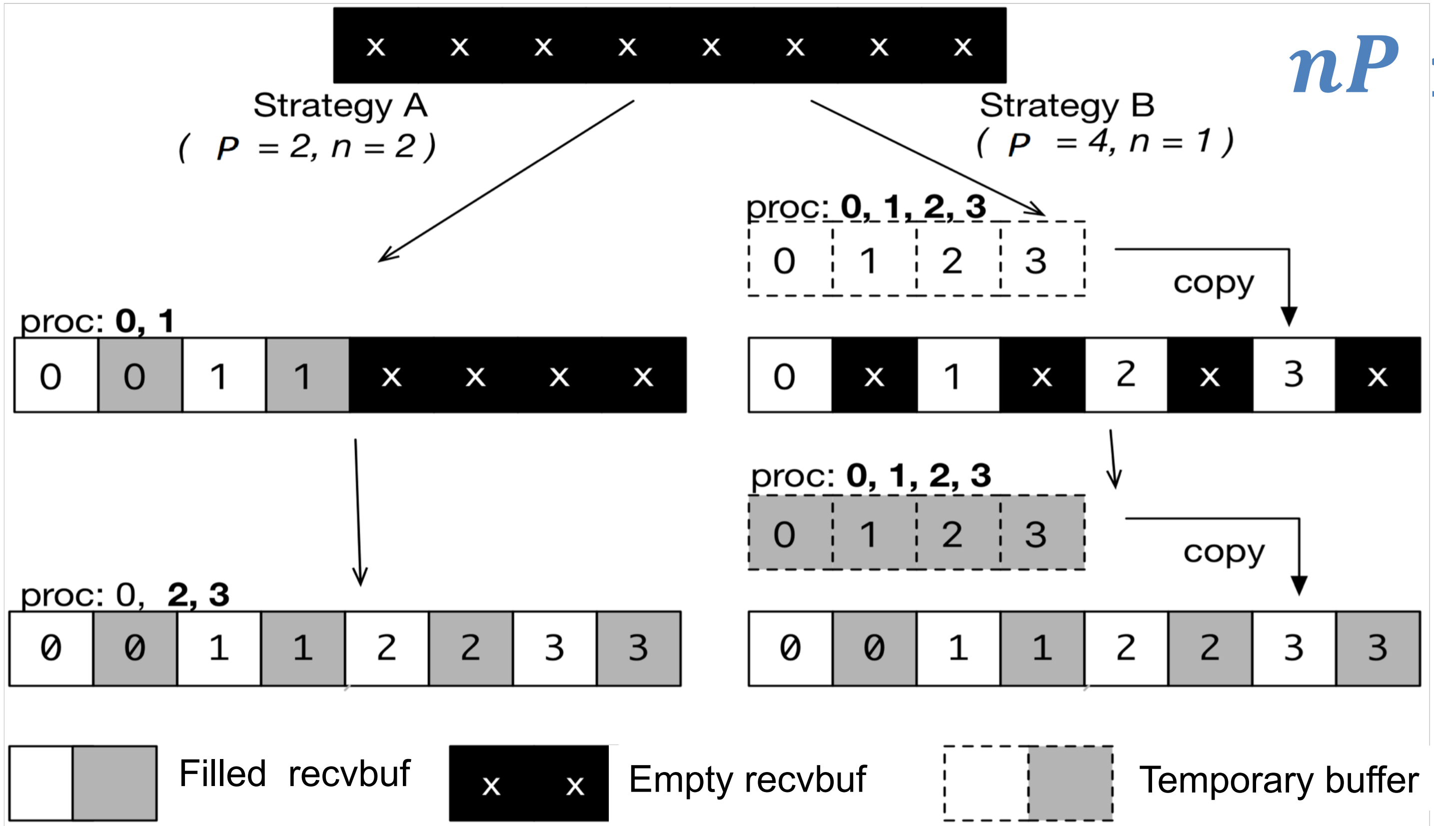
- Most scaling problems with the use of MPI collectives relate to both parallelism scale and message size
  - With ONLY 2 nodes with each having 24 cores and 64 GB memory, we easily find 4 scaling problems inside released MPI libraries.
  - Scaling problems related only to the number of processes are not found yet

Workarounds



Workaround (1) ✂

$nP \leq 4$



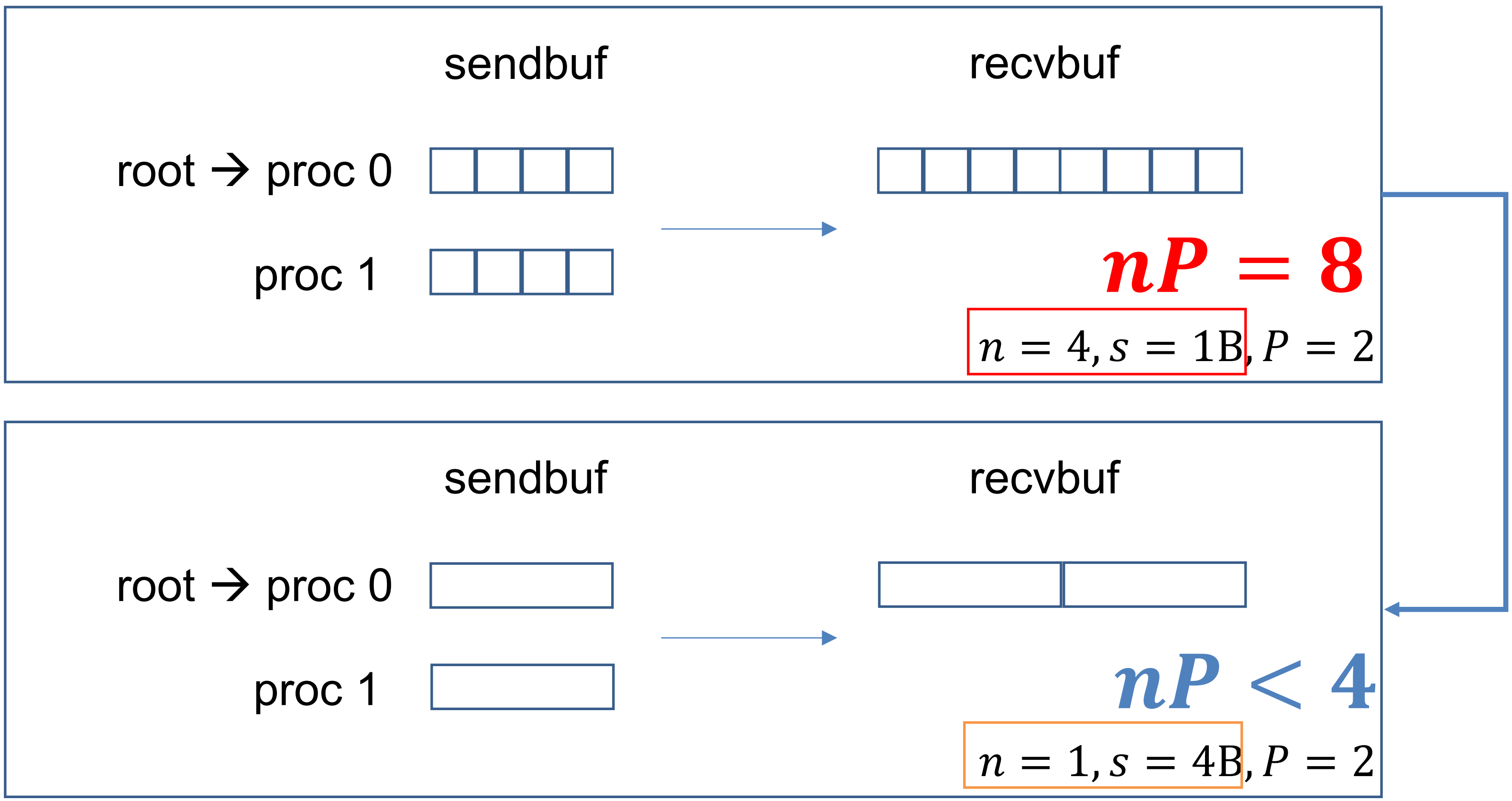
Partitioning one MPI_Gatherv communication using two strategies supposing the bug is triggered when $nP > 4$. Four processes ($P = 4$) are involved with each sending two elements ($n = 2$) and process 0 is the root process.

Workaround (2)

- Build big data type
  - Message size = s*n
  - Bigger data type (bigger s) → smaller n

- Only effective when the scaling problem is unrelated to s
  - Effective case: $nP > 4$
  - Ineffective case: $snP > 4$

Workaround (2)



Build big data type for MPI_Gather to avoid a bug triggered when $nP > 4$.

Outline

- Scaling Problem
- Avoidance Framework
- Evaluation
- Conclusion

Evaluation – Setting

- Tianhe-2:
  - Each node has 24 cores and 64GB DRAM
  - One process per core
- MPI_Gatherv
  - Effectiveness of avoiding scaling problem
  - Performance

Evaluation – Effectiveness

- Our workarounds are effective till the memory limit is hit

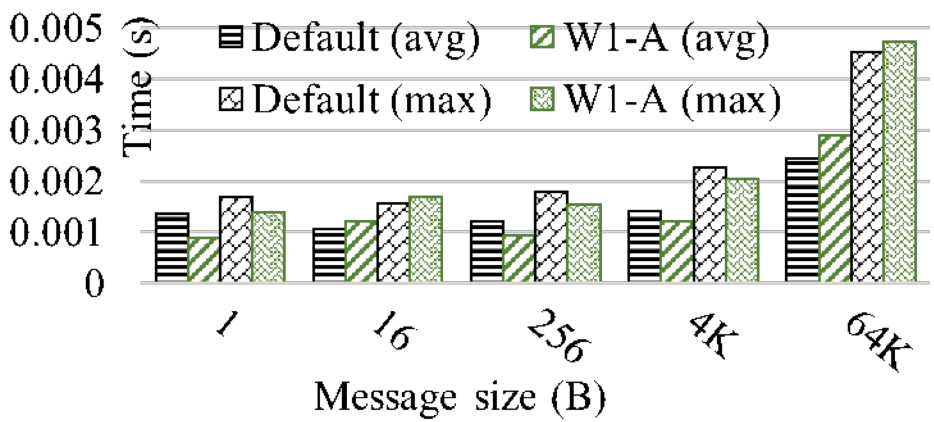
Workarounds for MPI_Gatherv that avoids the irregular collective limitation problem.

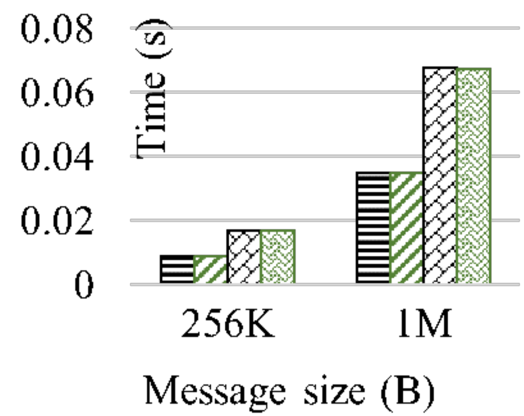
Scale ↓		Original		W1-A		W1-B		W2	
		n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M
P	192	10.5	2.21	256	54.00	256	54.0	272	57.38
	768	2.625	2.03	68	52.60	72	55.69	72	55.69

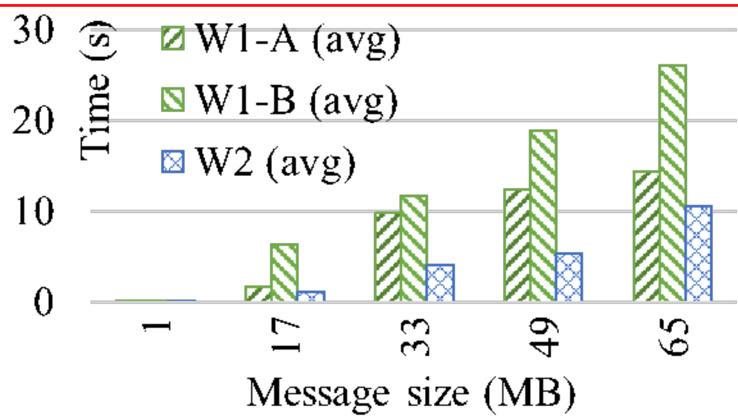
- n_s: the maximal workable n (unit: 1 M, i.e., 2^20)
- R_M : the maximal memory consumption on one node calculated according to MPI standard

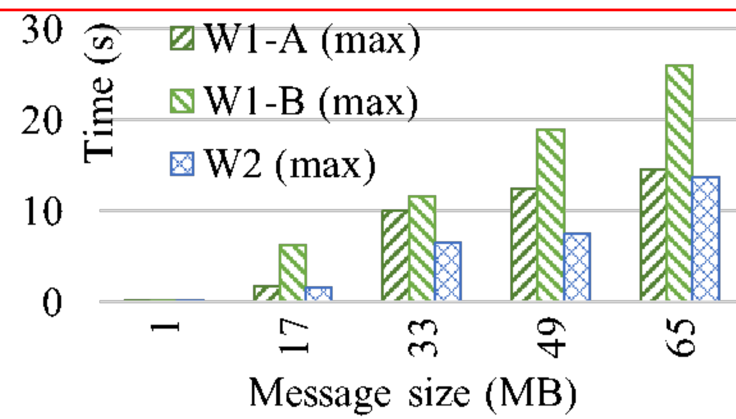
23X increase!

Evaluation – Performance









MPI_Gatherv [*P=768*, *s=1* B bug occurs when *n>2.625* M].

Evaluation -- Summary

- Effectiveness:
  - W1-B is the best
- Performance:
  - W2 is the best
  - The time cost of a collective based on either W1-A or W1-B increases linearly as n increases

Outline

- Scaling Problem
- Avoidance Framework
- Evaluation
- Conclusion

Conclusion

- Scaling problems are hard to be fixed and thus uses often need to spend days and months to wait for an official fix

- We provide a non-intrusive framework for application users as an immediate remedy
  - Easier than debugging
  - Faster than official fix

Thank you!