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

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

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

**Root process:** Proc 0

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A1</td>
<td></td>
</tr>
</tbody>
</table>

Proc 1

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>B1</td>
<td></td>
</tr>
</tbody>
</table>

---

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

---

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>Element count in one message</td>
</tr>
<tr>
<td>(s)</td>
<td>Size of the data type in bytes</td>
</tr>
<tr>
<td>(P)</td>
<td>Total number of processes</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Prob.</th>
<th>Collective</th>
<th>MPI library</th>
<th>Type</th>
<th>Effect</th>
<th>Scale ((P, M))</th>
<th>Root cause (inside MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MPI.Gather</td>
<td>OpenMPI 1.4.3</td>
<td>3</td>
<td>H</td>
<td>((64, 4KB))</td>
<td>Environment setting dependency</td>
</tr>
<tr>
<td>2</td>
<td>MPI.Alltoall</td>
<td>OpenMPI 1.4.3</td>
<td>3</td>
<td>H</td>
<td>((44, 4MB))</td>
<td>Environment setting dependency</td>
</tr>
<tr>
<td>3</td>
<td>MPI.Allgather</td>
<td>OpenMPI 1.4.3</td>
<td>3</td>
<td>H</td>
<td>((64, 4MB))</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>MPI.Alltoallv</td>
<td>OpenMPI 1.7</td>
<td>3</td>
<td>H</td>
<td>((96, 512KB))</td>
<td>Network connection failure</td>
</tr>
<tr>
<td>5</td>
<td>MPI.Allgather</td>
<td>MPICH 2</td>
<td>3</td>
<td>D</td>
<td>(P \cdot M &gt; \text{INT_MAX})</td>
<td>Integer overflow in MPI</td>
</tr>
<tr>
<td>6</td>
<td>MPI.Send + Recv</td>
<td>Intel MPI 5.1.2</td>
<td>2</td>
<td>H</td>
<td>((2, 64KB))</td>
<td>OS (ubuntu) dependency</td>
</tr>
<tr>
<td>7</td>
<td>MPI.Bcast</td>
<td>Intel MPI 5.1.2</td>
<td>2 or 3</td>
<td>H</td>
<td>((2, 64KB))</td>
<td>Unknown to developers</td>
</tr>
<tr>
<td>8</td>
<td>MPI.Bcast</td>
<td>Intel MPI 2017</td>
<td>2 or 3</td>
<td>H</td>
<td>((-1, 16KB))</td>
<td>Platform (KNL &amp; BDW) dependency</td>
</tr>
</tbody>
</table>
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 \textit{displs} (C int, commonly 32 bits) of \textbf{irregular collectives} can be easily corrupted by integer overflow.

Calculate address: \( \text{recvbuf} + \text{displs}[0] \times s \)

<table>
<thead>
<tr>
<th>Each process’ sendbuf</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root’s recvbuf</td>
<td></td>
</tr>
</tbody>
</table>

In MPI\_Gatherv, the root process calculate addresses for the incoming messages when \textit{displs} is not corrupted.
Outside MPI

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

\[ \text{Calculate address: } \text{recvbuf} + \text{displs}[0] \times s \]

Each process' \textit{sendbuf}

Root's \textit{recvbuf}: 0

In MPI\_Gatherv, the root process calculate addresses for the incoming messages when \textit{displs} is not corrupted.
Outside MPI

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

\textbf{Calculate address:} \hspace{1cm} \textit{recvbuf} + \textit{displs}[1] \times s

Each process’
\textit{sendbuf}

\begin{tabular}{c}
\hline
1 \\
\hline
\end{tabular}

Root’s \textit{recvbuf}

\begin{tabular}{c}
\hline
0 \\
\hline
\end{tabular}

In MPI\textunderscore Gatherv, the root process calculate addresses for the incoming messages when \textit{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`

<table>
<thead>
<tr>
<th>Root’s <code>recvbuf</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

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: \( \text{recvbuf} + \text{displs}[2] \times s \)

Each process’ `sendbuf`  

Root’s `recvbuf`  

<table>
<thead>
<tr>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

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 \textit{displs} (C \texttt{int}, commonly 32 bits) of \textit{irregular collectives} can be easily corrupted by integer overflow.

Calculate address: \[ \text{recvbuf} + \text{displs}[2] \times s \]

Each process’ \textit{sendbuf}

Root’s \textit{recvbuf}:

\begin{center}
\begin{tabular}{ccc}
0 & 1 & 2 \\
\end{tabular}
\end{center}

In MPI\_Gatherv, the root process calculate addresses for the incoming messages when \textit{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`

| 0 | 1 | 2 | i | P-1 |

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: \(recvbuf + displs[0] \times s\)

In MPI_Gatherv, the root process calculate addresses for the incoming messages when `displs` is corrupted.
Outside MPI

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

Calculate address: \[ \text{recvbuf} + \text{displs}[0] \times s \]

In MPI\_Gatherv, the root process calculate addresses for the incoming messages when \textit{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: \( \text{recvbuf} + \text{displs}[1] \times s \)

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: \[ \text{recvbuf} + \text{displs}[1] \times s \]

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:  
\[ \text{recvbuf} + \text{displs}[2] \times s \]

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

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

Calculate address: \[ \text{recvbuf} + \text{displs}[2] \times s \]

In MPI\_Gatherv, the root process calculate addresses for the incoming messages when \textit{displs} is corrupted.
In the user code, displacement array `displs` (C int, commonly 32 bits) of irregular collectives can be easily corrupted by integer overflow.

Calculate address: \( \text{recvbuf} + \text{displs}[i] \times s \)

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: \( \text{recvbuf} + \text{displs}[i] \times s \)

\[ \text{displs}[i] < 0 \]

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
  
  \[ N < displs[P - 1] + INT_MAX \]

  \[ \rightarrow N < 2 \times INT_MAX \]

- For **MPI_Gather** (a regular collective),
  
  \[ N \leq P \times INT_MAX \]
Outside MPI

- In the user code, displacement array $\text{displs}$ (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
  \[ N < \text{displs}[P - 1] + \text{INT\_MAX} \]
  \[ \rightarrow N < 2 \text{ INT\_MAX} \]
  **Huge gap:** \( \frac{2}{P} \)

- For MPI_Gather (a regular collective),
  \[ N \leq P \text{ INT\_MAX} \]
Outside MPI

- Irregular collectives’ limitation due to displacement array \textit{displs} of data type \textit{C int}

- Replace \textit{int} with \textit{long long int}?
  - Discussed yet never done --- backward compatibility
An immediate remedy is in need!
Outline

- Scaling Problem
- Avoidance Framework
- Evaluation
- Conclusion
Avoidance

Workaround strategy

Scaling problem’s trigger

```c
int MPI_Collective (...) {
    if (true == check_problem_trigger)
        MPI_Collective_Fix (...)  // invoke our fix
    else
        PMPI_Collective (...)     // invoke the default
}
```
Trigger (1) [Outside MPI]

Irregular collectives’ limitation’s trigger is

\[ \text{displs}[i] < 0 \]
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

(W1) Partition communication

(W1-A) Partition processes

(W1-B) Partition the message

(W2) Build big data type
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 $= s \times n$
  - Bigger data type (bigger $s$) $\rightarrow$ 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.

<table>
<thead>
<tr>
<th>Scale</th>
<th>Original</th>
<th>W1-A</th>
<th>W1-B</th>
<th>W2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_s</td>
<td>R_M</td>
<td>n_s</td>
<td>R_M</td>
<td>n_s</td>
</tr>
<tr>
<td>P</td>
<td>192</td>
<td>10.5</td>
<td>2.21</td>
<td>256</td>
</tr>
<tr>
<td>768</td>
<td>2.625</td>
<td>2.03</td>
<td>68</td>
<td>52.60</td>
</tr>
</tbody>
</table>

- \( n_s \): 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

\[ \text{MPI\_Gatherv} \ [P=768, s=1] \text{ B bug occurs when } n > 2.625 \text{ M}. \]
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
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!