Pilgrim

Near-Lossless MPI Tracing and Proxy Application Autogeneration

Chen Wang ¹, Yanfei Guo ², Pavan Balaji ³, Marc Snir ¹
¹ University of Illinois Urbana-Champaign
² Argonne National Laboratory
³ Meta, Inc
Why do we collect MPI information

- For MPI and application users:
  - How frequent calls are?
  - Am I providing the right hints to MPI for my usage?
  - Am I using MPI correctly?
- For MPI developers:
  - What features are used and in what way?
  - Message sizes, communicator sizes, buffer reuse?
  - Are send/recv sizes the same or different?
  - Are collective operation datatype on all processes the same or different?
Pilgrim vs. Existing tools

Pilgrim stores every parameter of every MPI call

Existing tools are either incomplete or have unacceptable overhead (time or space)
Pilgrim vs. Existing tools

Pilgrim stores every parameter of every MPI call

Existing tools are either incomplete or have unacceptable overhead (time or space)

<table>
<thead>
<tr>
<th>Functions Supported</th>
<th>Cypress</th>
<th>ScalaTrace</th>
<th>Pilgrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total: 446</td>
<td>56</td>
<td>125</td>
<td>446</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Popular Parameters</th>
<th>Cypress</th>
<th>ScalaTrace</th>
<th>Pilgrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Status</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MPI_Request</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MPI_Comm</td>
<td>intra</td>
<td>intra and inter</td>
<td>intra and inter</td>
</tr>
<tr>
<td>MPI_Datatype</td>
<td>only the size</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>src/dst/tag</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>memory pointer</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>
Challenges

1. **Scalability:**
   - the longer an application runs or the more nodes it runs on, the more function calls it will make

2. **Usefulness:** meaningful for post-processing
   - MPI_Comm, MPI_Request, Memory pointers?

3. **Correctness and completeness:**
   - Over 400 MPI functions; Many corner cases, e.g., non-blocking communicator creation.
1. **Scalability:**
   - the longer an application runs or the more nodes it runs on, the more function calls it will make

2. **Usefulness:** meaningful for post-processing
   - MPI_Comm, MPI_Request, Memory pointers?

3. **Correctness and completeness:**
   - Over 400 MPI functions; Many corner cases, e.g., non-blocking communication creation.
How?

- Primarily relies on "recurring pattern recognition"
- The key is to detect as many patterns as possible.
- Store the patterns in a context-free-grammar (CFG)

```
MPI_Barrier(comm1);

“a a b a b c d c d”

S → a A² B²
A → a b
B → c d

MPI_Comm_size(comm2, 3);
```
How?

- Primarily relies on “recurring pattern recognition”
  - The key is to detect as many patterns as possible.
  - Store the patterns in a context-free-grammar (CFG)

\[
\begin{array}{|c|c|c|}
\hline
1 & 2 & 3 \\
\hline
8 & 9 & 4 \\
\hline
7 & 6 & 5 \\
\hline
\end{array}
\]

e.g., 2D 5-points periodical stencil will exhibit up to 9 unique communication patterns
Result

- Pilgrim stores more information with less space and time overhead.
- Only unique communication patterns matter.
- Trace size will stay constant if all patterns are recognized.
- Overhead depends on communication-to-computation ratio.

FLASH StirTurb 3D Simulation

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScalaTrace</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Pilgrim</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MPI Calls (×10E6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Result

- Pilgrim stores more information with less space and time overhead.
- Only unique communication patterns matter.
  - **Trace size will stay constant if all patterns are recognized**
- Overhead depends on communication-to-computation ratio.
Proxy app generation
Issues of the current method

Designing a proxy app manually is a nontrivial task and exposes three major issues:

1. It requires the involvement of the experts of the original application when it tries to best mimic the original behavior. It takes significant efforts since the original application code is most likely huge.

2. It requires access to the source code of the original application, which may be infeasible for classified applications.

3. Removing part of the intrinsic logic (e.g., computation) from the original application makes debugging and correctness checking difficult.
Proxy app generation

Autogeneration algorithm

With the detailed information preserved by Pilgrim, however, we can design a proxy app generator that addresses these issues.

1. The proxy app is generated automatically from Pilgrim traces with little or no human intervention.

2. It relies only on the traces of the targeted application, not the source code; and the tracing process has already stripped away the computation information.

3. Correctness checking is easy because we can run the generated proxy app with Pilgrim again and compare its traces with the original application's traces.
Proxy app generation

Autogeneration algorithm

With the detailed information preserved by Pilgrim, however, we can design a proxy app generator that addresses these issues.

1. The proxy app is generated automatically from Pilgrim traces with little or no human intervention.

2. It relies only on the traces of the targeted application, not the source code; and the tracing process has already stripped away the computation information.

3. Correctness checking is easy because we can run the generated proxy app with Pilgrim again and compare its traces with the original application's traces.
Proxy app generation

- Restore the captured recurring patterns, which represent the original code structure.

- The proxy app will be small and clean if Pilgrim compresses well.

- The generated code size is proportional to the size of the compressed trace.
Other features and limitations

• Pilgrim tracks all memory management operations (CUDA and CPU).
  • It also knows which GPU device was the memory allocated

• Multi-threading support is almost done, e.g., MPI_THREAD_MULTIPLE

• For the proxy-app generator, user-defined MPI functions (e.g., MPI_Copy_function) are not supported
  • Pilgrim traces function calls only at runtime and does not perform compile-time analysis
Resources

- Pilgrim is publicly available at: https://github.com/pmodels/pilgrim

- Papers:
  - Chen Wang, Yanfei Guo, Pavan Balaji, and Marc Snir. “Near-Lossless MPI Tracing and Proxy Application Autogeneration”, under review, TPDS.

- Contact:
  - Chen Wang (chenw5@illinois.edu)
  - Yanfei Guo (yguo@anl.gov)