MUST: Compiler-aided MPI correctness checking with TypeART

FOSDEM'23 HPC, Big Data, and Data Science Devroom
Sunday, 5th February 2023, Brussels, Belgium

Alexander Hück
Scientific Computing, TU Darmstadt
alexander.hueck@sc.tu-darmstadt.de

Joachim Jenke
IT Center, RWTH Aachen
jenke@itc.rwth-aachen.de

MUST
https://itc.rwth-aachen.de/must

TypeART
https://github.com/tudasc/typeart
Message Passing Interface (MPI)

MPI is the de-facto standard for parallel, distributed application in HPC
- Defines a large set of (communication) routines
- Designed for heterogeneous systems: handles conversions for, e.g., endianness
- Only minimal error checking can be expected from the MPI library itself

```c
MPI_Send(buffer, n, MPI_DOUBLE, destination, tag, comm)
```

1. Data is transferred as a type-less `void*` buffer
2. Data length and type is user-specified
3. Message envelope must have proper destination/receiver
How Many Errors in this Example?

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv) {
    int rank, size, buf[8];

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);
    return 0;
}
```
How Many Errors in this Example?

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv) {
    int rank, size, buf[8];

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);
    return 0;
}
```

No MPI_Init before first MPI call
#include <mpi.h>
#include <stdio.h>

int main (int argc, char ** argv) {
  int rank, size, buf[8];

  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  MPI_Datatype type;
  MPI_Type_contiguous (2, MPI_INTEGER, &type);

  MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);

  printf ("Hello, I am rank %d of %d.\n", rank, size);
  return 0;
}
How Many Errors in this Example?

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv) {
    int rank, size, buf[8];

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);
    return 0;
}
```

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No MPI_Init before first MPI call</td>
</tr>
<tr>
<td>2</td>
<td>Fortran type in C</td>
</tr>
<tr>
<td>3</td>
<td>Recv-Recv deadlock</td>
</tr>
<tr>
<td>4</td>
<td>Rank 0: src=size (out of range)</td>
</tr>
<tr>
<td>5</td>
<td>Type not committed</td>
</tr>
<tr>
<td>6</td>
<td>Type not free’d before end</td>
</tr>
<tr>
<td>7</td>
<td>Send 4 int, Recv 2 int: trunc.</td>
</tr>
<tr>
<td>8</td>
<td>No MPI_Finalize before end</td>
</tr>
</tbody>
</table>
MUST Report of Deadlock

Message
The application issued a set of MPI calls that can cause a deadlock! The graphs below show details on this situation. This includes a wait-for graph that shows active wait-for dependencies between the processes that cause the deadlock. Note that this process set only includes processes that cause the deadlock and no further processes. A legend details the wait-for graph components in addition. Below these graphs, a message queue graph shows active and unmatched point-to-point communications. This graph only includes operations that could have been intended to match a point-to-point operation that is relevant to the deadlock situation. The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary).

Active Communicators

Wait-for Graph

Legend

Active MPI Call

Sub Operation

Call stack omitted for this example
MUST: Agent-based Distributed Analysis

MPI application with 4 ranks

- Local analysis (e.g., TypeART MPI buffer type correctness)
- Distributed analysis for scalability (e.g., matching MPI send-recv types)
- Centralized analysis for global knowledge (e.g., global deadlocks)
- MUST specific communication (hidden from application)
MPI Type Correctness

MPI_Send(buffer, n, MPI_DOUBLE, ...)

1. Data is transferred as a type-less void* buffer
2. Data length and type is user-specified
Limitations of Dynamic Checking of Type-Related Errors

MUST can detect type mismatches of, e.g., send-recv communication pairs:

\[
\text{MPI\_Send( data, buffer\_size, MPI\_FLOAT, \ldots );} \\
\text{MPI\_Recv( data, buffer\_size, MPI\_DOUBLE, \ldots );}
\]
Limitations of Dynamic Checking for Type-Related Errors

MUST cannot check type-less `void*` buffer `data` for, e.g., `MPI_Send`

- *Is it of type* `MPI_FLOAT`?
- *Is it of length* `buffer_size`?

→ Can not be answered by MUST without further tooling

**Process 0**

```
Memory Allocation
double* buff = ...;
```

```
MPI_Send( data, buffer_size, MPI_FLOAT, ... );
```

*Local analysis* only possible with TypeART
Examples

AMG2013, a CORAL performance and parallel scaling benchmark [Coral’20]
- In parcsr_mv/par_csr_matrix.c:1236, reported by [DKL LLVM’15]:

\[
\text{MPI\_Bcast( \&global\_data[3], global\_size-3, MPI\_INT, ... );}
\]

104.milc, a SPEC MPI benchmark [SpecMPI’07]
- In com_mpi.c:480:

\[
\text{MPI\_Allreduce( cpt, \&work, 2, MPI\_FLOAT, ... );}
\]

Interpreted as array of 2 floats → Benign today, but tomorrow?
MUST and TypeART: Dynamic MPI Checks

- Deadlocks
- ...

Intercepted MPI API

MPI Application

MUST
MUST and TypeART: Instrumenting Allocations

TypeART

MPI-related Allocation Instrumentation

LLVM Compiler Extension

MPI Application

Intercepted MPI API

Alloc Free

TypeART Runtime

MUST

- Deadlocks
- ...

- ...

- ...

- ...
MUST and TypeART: Type Information for Each Alloc

**TypeART**
- LLVM Compiler Extension
- MPI-related Allocation Instrumentation
- Serialized Type Information of Allocations

**MPI Application**
- Intercepted MPI API

**TypeART Runtime**
- Alloc
- Free

**MUST**
- Deadlocks
- …
MUST and TypeART: MPI Buffer Type Checking

TypeART

- LLVM Compiler Extension

MPI-related Allocation Instrumentation

Serialized Type Information of Allocations

MPI Application

Allocates

Interceptored MPI API

MUST

- Deadlocks
- ...

TypeART Runtime

Intercepts MPI API

Process-local Runtime MPI Buffer Type Checks

Address

Type Info
Instrumentation of Allocations

Memory allocation instrumentation is done on LLVM IR
- For each allocation, collect (1) memory address, (2) type id (to map to type layout), and (3) dynamic array length
- C-like example of heap allocation:

\[
\text{float* data} = (\text{float*}) \text{malloc}(n \times \text{sizeof(float)}); \\
\_\text{typeart\_alloc} ((\text{void*}) \text{data}, \text{TYPEART\_FLOAT}, n)
\]

Not shown
- Stack allocation handling more complex but overall similar interface
- Lifetime (and the tracking) is scope dependent
- Globals registered once at startup for whole program duration

Added instrumentation

Filtered during compilation if never passed to MPI
Filtering of allocations

TypeART statically filters stack and global allocations that are never part of an MPI call.

```c
Translation Unit A

void foo_bar(double* d);
void bar(double* d) {
  *d = 2;
}
void foo() {
  double a = 1.0, b = 2.0;
  bar(&a);
  foo_bar(&b);
}

Translation Unit B

void foo_bar(double* d) {
  *d = 2.0 + *d;
}

Can filter with whole-program view
```
Type ID

Type ID is a unique number for each (used) type in the target code

1. **Built-in** types have pre-determined type ids, e.g.,
   - float: TYPEART_FLOAT
   - etc...

2. **User-defined** types are serialized during compilation by our pass (.yaml)

```
struct Point {
    int i;
    double d1;
    double d2;
};
```

Static Info

- id: 256
- name: struct.Point
- extent: 24
- member_count: 3
- offsets: [0, 8, 16]
- types: {id: .. }

Type ID = 256
Querying Types

C API to query type information behind a memory address, e.g.,

- **typeart_status** typeart_get_type(const void* addr, int* type_id, size_t* count);

  Return status:
  - TYPEART_OK
  - TYPEART_UNKNOWN_ADDRESS
  - ...

- **typeart_status** typeart_resolve_type_id(int type_id, typeart_struct_layout* struct_layout);

  ```
  struct typeart_struct_layout {
    int type_id;
    const char* name;
    ...
  };
  ```

  OUT
MUST and TypeART usage

1) Compile and link with TypeART compiler wrapper (optional, for buffer type checks)
   - mpicc → typeart-mpicc

2) Replace “mpiexec” with command “mustrun”
   - e.g., mustrun -np 4 --must:typeart my-app.bin

3) Inspect “MUST_Output.html” in run directory for issues

Note: The mustrun script will use an extra process for non-local checks (invisible to application)
   - “mustrun -np 4 ...” will issue a “mpirun -np 5 ...”
   - Make sure to allocate the extra task in batch jobs
Brief Evaluation (1/2)

Runtime/memory impact depends on number of tracked allocations

- Lulesh → comparably few stack and few heap allocations in total
- 122.tachyon → High number of stack allocations
Brief Evaluation (2/2)

**Static** instrumentation of memory operations with *allocation filtering in %*

<table>
<thead>
<tr>
<th></th>
<th>Memory Alloc and Free Operations</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Heap</td>
<td>Free</td>
<td>Stack (%)</td>
<td>Global (%)</td>
</tr>
<tr>
<td>LULESH 2.0</td>
<td>14</td>
<td>6</td>
<td>54</td>
<td>21.0</td>
<td>80</td>
</tr>
<tr>
<td>122.tachyon</td>
<td>80</td>
<td>51</td>
<td>579</td>
<td>2.0</td>
<td>372</td>
</tr>
</tbody>
</table>

**Dynamic** traced memory operation counts tracked by the TypeART runtime

<table>
<thead>
<tr>
<th></th>
<th>Total Global</th>
<th>Total Heap</th>
<th>Total Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>LULESH 2.0</td>
<td>0</td>
<td>525,060</td>
<td>34,149</td>
</tr>
<tr>
<td>122.tachyon</td>
<td>10</td>
<td>13,759</td>
<td>78,307,707</td>
</tr>
</tbody>
</table>
Conclusion

MUST & TypeART combine dynamic analysis with compiler tooling to enable complete type checking of MPI applications.

Process 0

Memory Allocation

\[
\text{double* \texttt{buff} = \ldots;}
\]

MPI_Send( data, buffer_size, MPI_FLOAT, \ldots);

a) \textit{Local analysis} with TypeART

b) \textit{Distributed analysis} with MUST

Process 1

MPI_Recv( data, buffer_size, MPI_DOUBLE, \ldots);

\[
\text{MPI\_Send( data, buffer\_size, MPI\_FLOAT, \ldots);}
\]

\[
\text{MPI\_Recv( data, buffer\_size, MPI\_DOUBLE, \ldots);}
\]
Conclusion

MUST & TypeART combine dynamic analysis with compiler tooling to enable complete type checking of MPI applications.

**MUST**
https://itc.rwth-aachen.de/must
License BSD 3-Clause

“Dynamic MPI correctness checking”

**TypeART**
https://github.com/tudasc/typeart
License BSD 3-Clause

“C/C++ type and memory allocation tracking”
References


References

