MUST: Compiler-aided MPI correctness checking with TypeART

TECHNISCHE UNIVERSITÄT DARMSTADT

RWTH AACHEN UNIVERSITY

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

IT Center | RWTH AACHEN UNIVERSITY | SCIENTIFIC COMPUTING

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

`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?

```
#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?

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_W
  MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);

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

1	No MPI_Init before first MPI call
2	Fortran type in C
3	Recv-Recv deadlock
4	Rank 0: src=size (out of range)
5	Type not committed
6	Type not free'd before end
7	Send 4 int, Recv 2 int: trunc.
8	No MPI_Finalize before end

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_W
  MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);

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

1	No MPI_Init before first MPI call
2	Fortran type in C
3	Recv-Recv deadlock
4	Rank 0: src=size (out of range)
5	Type not committed
6	Type not free'd before end
7	Send 4 int, Recv 2 int: trunc.
8	No MPI_Finalize before end

MUST Report of Deadlock

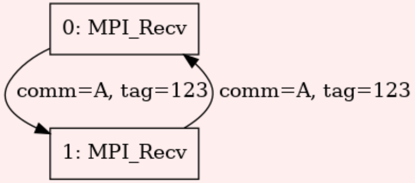
Message

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

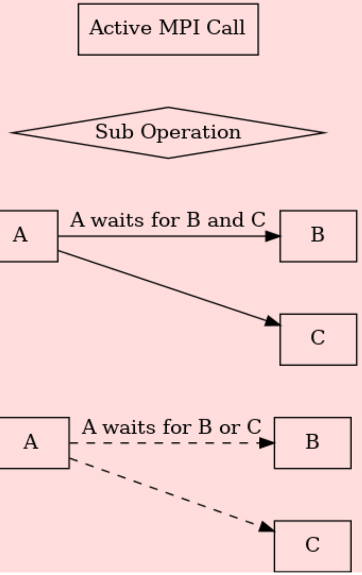
Comm:	A
MPI_COMM_WORLD	

Wait-for Graph



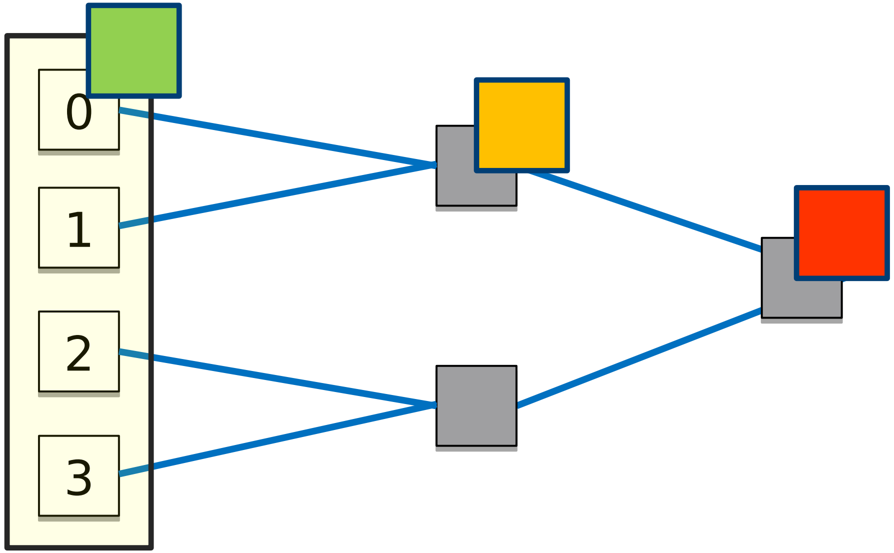
Call stack omitted for this example

Legend



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:

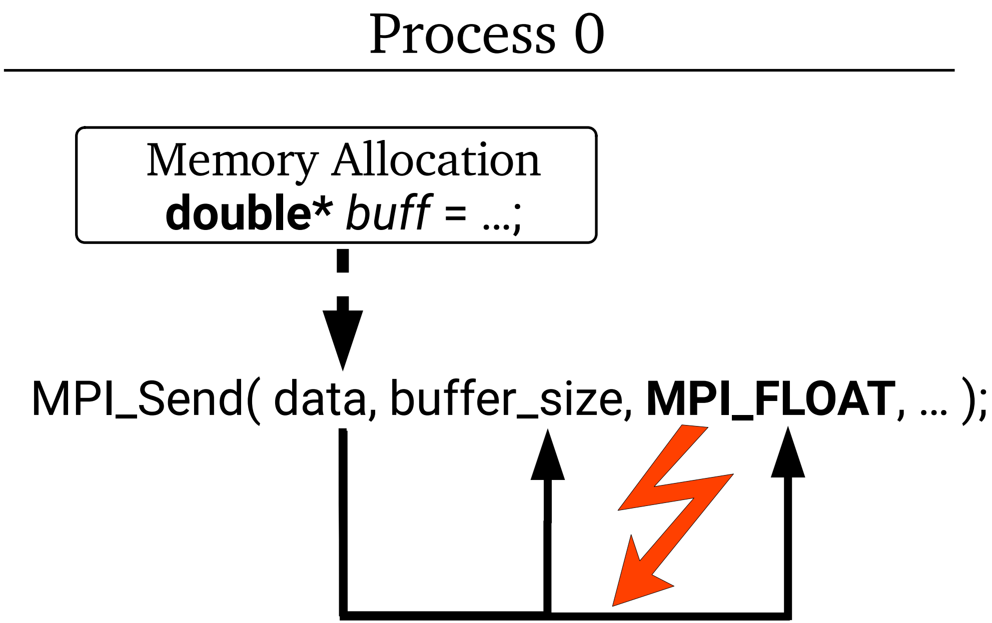
Process 0	Process 1
	Distributed analysis with MUST
MPI_Send(data, buffer_size, **MPI_FLOAT**, …);	MPI_Recv(data, buffer_size, **MPI_DOUBLE**, …);

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



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]:

```
MPI_Bcast( &global_data[3], global_size-3, MPI_INT, … );
```

HYPRE_BigInt $\xrightarrow{\text{alias}}$ long long

104.milc, a SPEC MPI benchmark [SpecMPI'07]

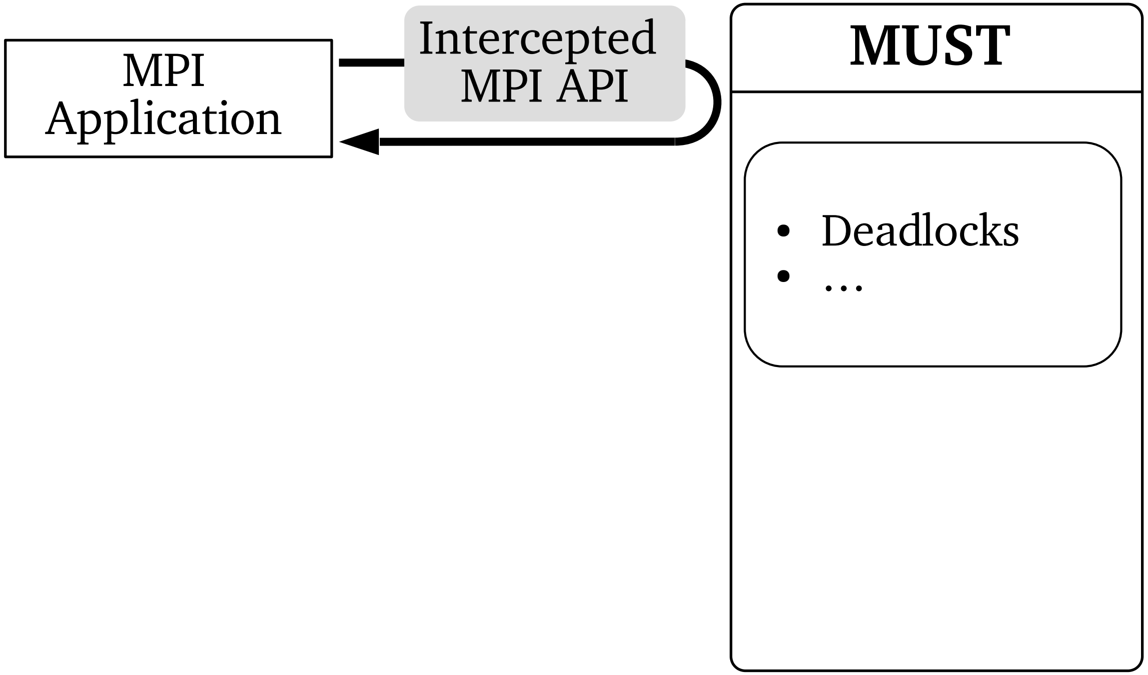
- In com_mpi.c:480:

```
MPI_Allreduce( cpt, &work, 2, MPI_FLOAT, … );
```

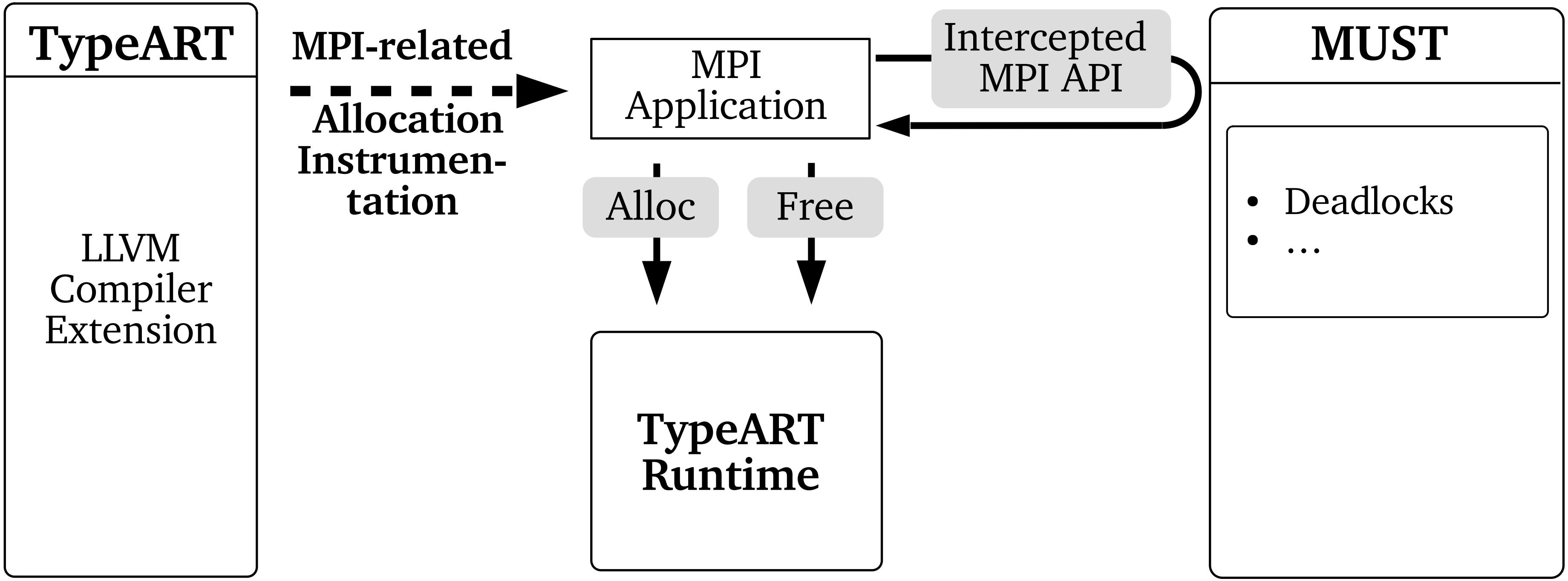
```
struct { float a; float b; };
```

Interpreted as array of 2 floats
→ *Benign today, but tomorrow?*

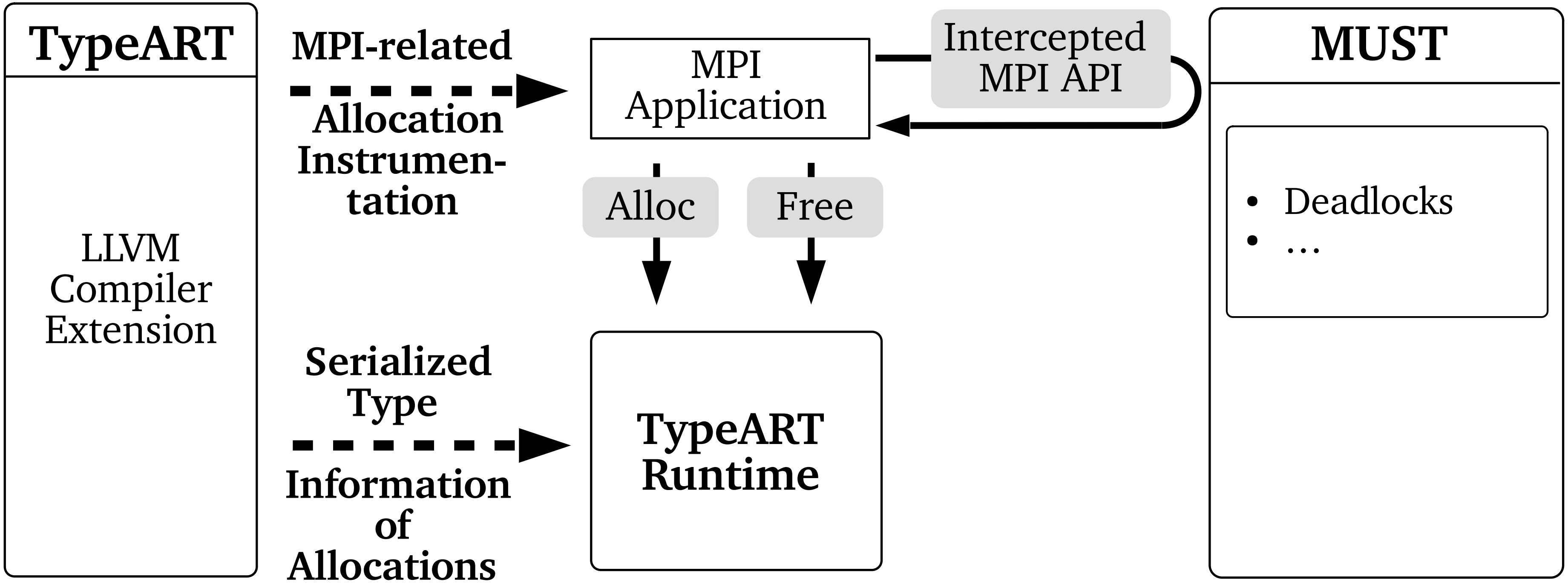
MUST and TypeART: Dynamic MPI Checks



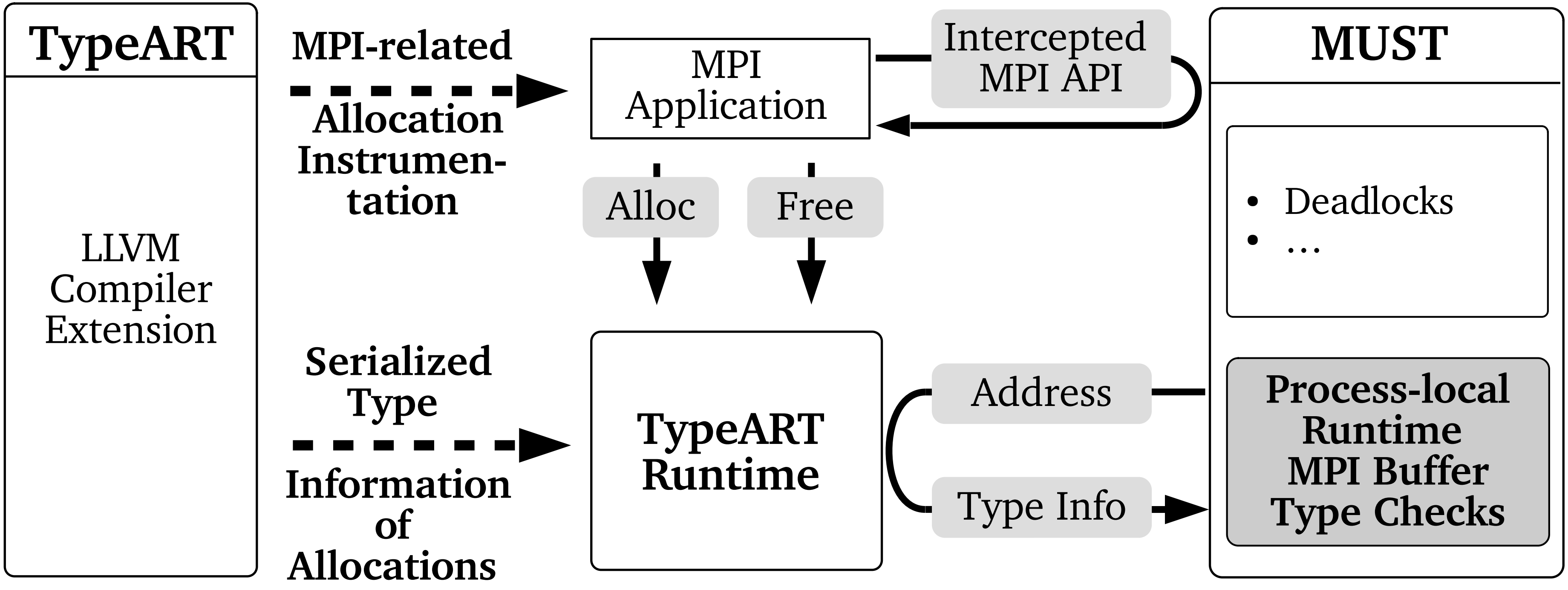
MUST and TypeART: Instrumenting Allocations



MUST and TypeART: Type Information for Each Alloc



MUST and TypeART: MPI Buffer Type Checking



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:

```
float* data = (float*) malloc(n * sizeof(float));
__typeart_alloc ( (void*) data, TYPEART_FLOAT, n )
```

Added instrumentation

Pointer, Type ID, Array length

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

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

Can filter with whole-program view

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;
}
```

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
Type ID = 256

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

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
- …

IN: Pointer, OUT: ID, Array length

- **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

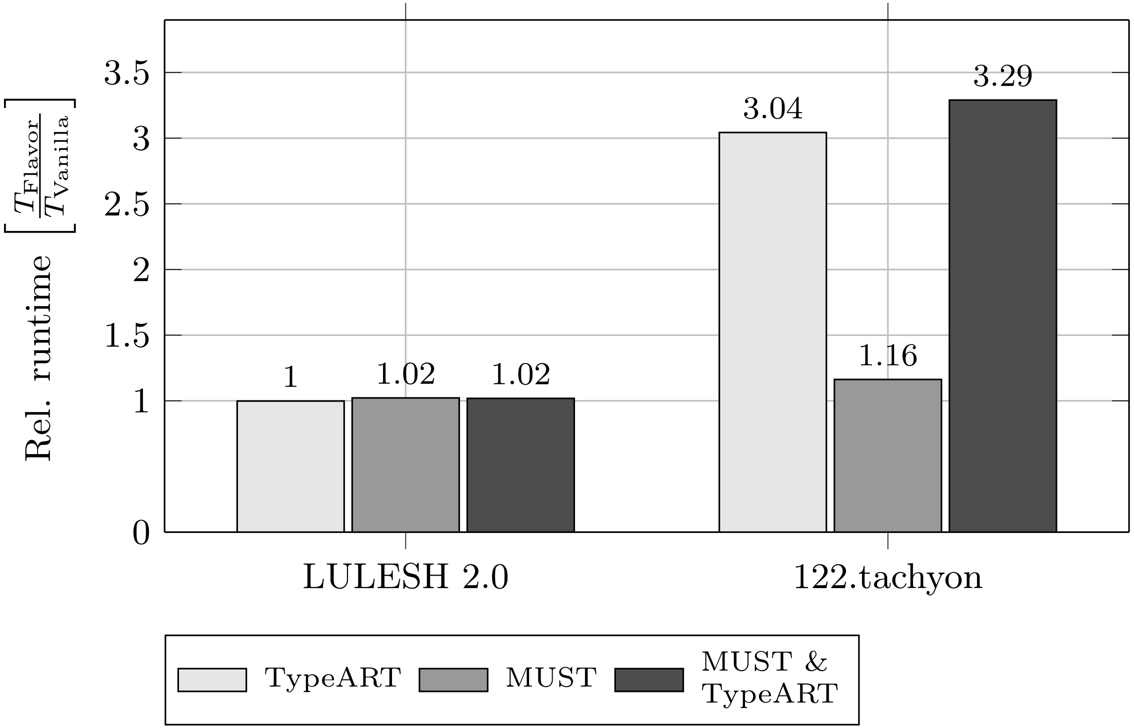
Note: The mustrun script will use an extra process for non-local checks (invisible to application)

- “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 %*

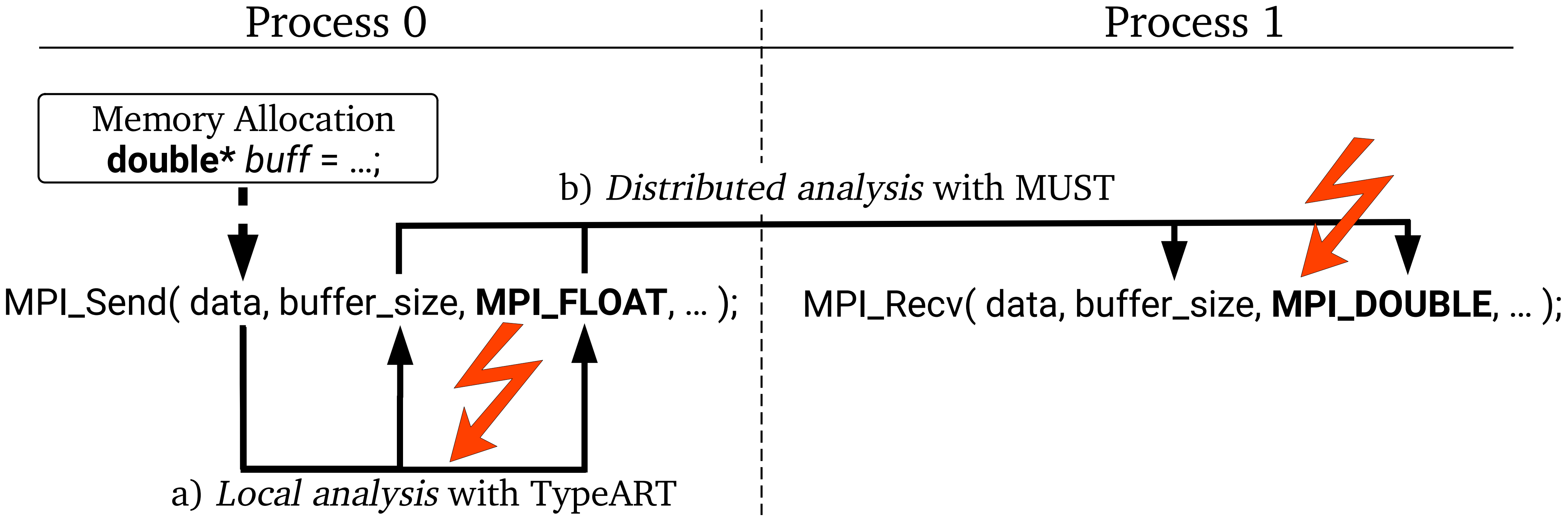
	Memory Alloc and Free Operations			
	Heap	Free	Stack (%)	Global (%)
LULESH 2.0	14	6	54 (21.0)	80 (100)
122.tachyon	80	51	579 (2.0)	372 (97.3)

Dynamic traced memory operation counts tracked by the TypeART runtime

	Total Global	Total Heap	Total Stack
LULESH 2.0	0	525,060	34,149
122.tachyon	10	13,759	78,307,707

Conclusion

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



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

[1] A. Hück, J.-P. Lehr, S. Kreutzer, J. Protze, C. Terboven, C. Bischof, M. S. Müller. **”Compiler-aided type tracking for correctness checking of MPI applications”** In 2nd International Workshop on Software Correctness for HPC Applications (Correctness), pp. 51–58. IEEE, 2018. DOI: **10.1109/Correctness.2018.00011**

[2] A. Hück, J. Protze, J.-P. Lehr, C. Terboven, C. Bischof, M. S. Müller. **“Towards compiler-aided correctness checking of adjoint MPI applications”** In 4th International Workshop on Software Correctness for HPC Applications (Correctness). IEEE/ACM, 2020, pp. 40–48. DOI: **10.1109/Correctness51934.2020.00010**

[3] A. Hück, S. Kreutzer, J. Protze, J.P. Lehr, C. Bischof, C. Terboven, M. S. Müller. **“Compiler-Aided Type Correctness of Hybrid MPI-OpenMP Applications”** In IT Professional, vol. 24, no. 2, pp. 45–51. IEEE, 2022. DOI: **10.1109/MITP.2021.3093949**

[4] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, M. S. Müller. **“MPI Runtime Error Detection with MUST: Advances in Deadlock Detection”** In Scientific Programming, vol. 21, no. 3-4, pp. 109–121, 2013. DOI: **10.3233/SPR-130368**

References

[DKL LLVM’15] A. Droste, M. Kuhn, and T. Ludwig, **“MPI-Checker: Static Analysis for MPI”**, In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM ’15. New York, NY, USA: ACM, 2015, pp. 3:1–3:10.

[Coral’20] **“CORAL benchmark codes”**, Last accessed Feb 2020. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[SpecMPI’07] M. S. Müller, et al, **“SPEC MPI2007 — an application benchmark suite for parallel systems using MPI”**, Concurrency and Computation: Practice and Experience, vol. 22, no. 2, pp. 191–205, 2009.