Compile Time Correctness Checking of MPI Communication Schemes Using Coloured Petri Nets

PARS Workshop 2025

Michael Blesel, Michael Kuhn, Hans Fangohr
michael.blesel@ovgu.de
michael.kuhn@ovgu.de
hans.fangohr@mpsd.mpg.de

September 19, 2025

Faculty of Computer Science
Otto von Guericke University Magdeburg

Outline

- Parallel programming with MPI can get very complex
- **Process local errors**:
  - Data type mismatches, wrong handling of memory buffers, ...
  - Can be analyzed from the perspeective of a single process
- **Communication scheme based errors**:
  - Deadlocks, lost messages, ...
  - Analysis of these errors needs to reason about the interplay of multiple processes
- Code with these errors often compiles without a problem and needs to be analyzed manually
- Therefore correctness checking tools are desirable

- Can be broadly categorized into *static* and *dynamic* approaches
- **Dynamic tools**:
  - E.g. *MUST*
  - Can detect errors like deadlocks during runtime
- **Static analysis tools**
  - E.g. *PARCOACH*, *clang-tidy*/*MPI-Checker*
  - Mostly just work on the source code and don't need to execute the program
  - However, existing tools can not detect all the mentioned communication scheme based errors

Outline

- We introduce a *clang* compiler plugin called *SPMDClang*
  - *C/C++* support
- Does correctness checking for MPI communication schemes during compilation
  - Error detection for *deadlocks* and *lost messages*
- User gets familiar looking compiler warnings about errors in MPI communication
- We aim to keep setup/configuration effort as low as possible (just switch out the compiler in the program's build system)
- To achieve this a type of simulation of the program's communication scheme based on *Coloured Petri Nets* is executed during compilation

1. Creation of a global control flow graph (CFG)
2. Creation of specific CFG instances for each simulated MPI process
3. Creation of a Coloured Petri Net (CPN) that models the MPI communication scheme
4. Simulation of all orders of execution of the CPN to detect communication errors
5. Create compiler warnings for the detected errors

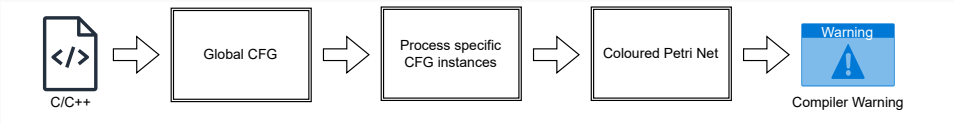


Figure 1: SPMDClang workflow

Outline

Example Program

- This example code produces a deadlock due to the blocking send operations
- The SPMDClang plugin will transform the code into a CPN and detect this error

```
1 if (rank == 0) {
2   MPI_Ssend(&v, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
3   MPI_Recv(&r, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD, ...);
4 } else if (rank == 1) {
5   MPI_Ssend(&v, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
6   MPI_Recv(&r, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, ...);
7 }
8 MPI_Barrier(MPI_COMM_WORLD);
```

Listing 1: Deadlock example code

Step 1: Creation of a Global CFG

- First a global CFG for the program is created from the function based CFGs that Clang creates
- `main` function of the program is scanned and the CFG of every function call to a user defined function is merged into it
- This step is recursively repeated until there is one CFG of the whole program
- This is required for the following analysis and transformation steps

Global CFG Example

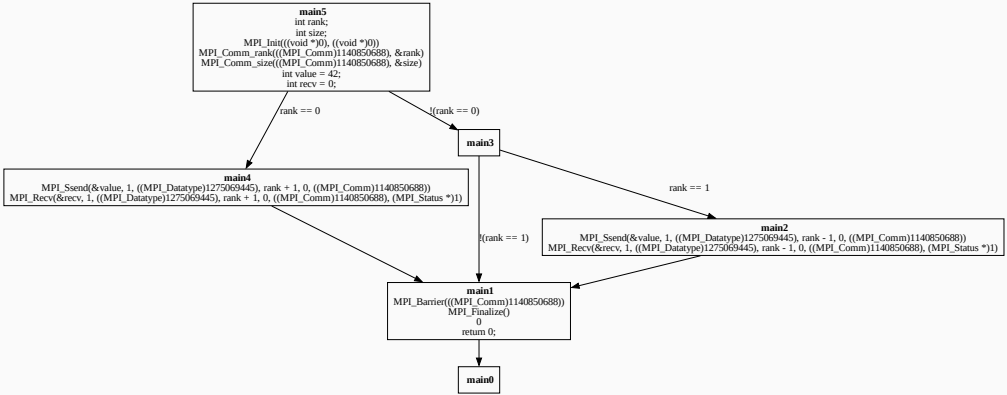


Figure 2: Global CFG of the example code

Interlude: Handling Runtime Dependent Values

- **Problem:** MPI programs will have runtime dependent variables, which makes a compile time analysis difficult
  - Obvious examples are MPI `rank` and `size` variables
- **Solution:** We simulate the communication scheme for specific numbers of processes
  - For example an even and an odd number of processes
- This means we can not check for correctness of all possible numbers of processes
  - However, this should suffice for most generalized communication schemes
- Future plans are to make the configurations that are simulated user definable (with sane defaults if not specified)

Step 2: Create MPI Process Specific CFGs

- SPMDClang now creates a specific CFG for each MPI process that is to be simulated
- Each CFG instance only contains parts of the control flow that can be reached by it
- This requires the plugin to be able to evaluate MPI `rank` and `size` based expressions that influence the control flow of the program
- To achieve this we apply a form of constant propagation

- Abstract Syntax Tree (AST) is scanned for `MPI_Comm_rank`/`MPI_Comm_size` calls
- Variables used in these calls are initialized with specific values
- AST is then traversed and each *integer*/*boolean* expression is assigned a lattice value
- This is also done for non `rank`/`size` variables

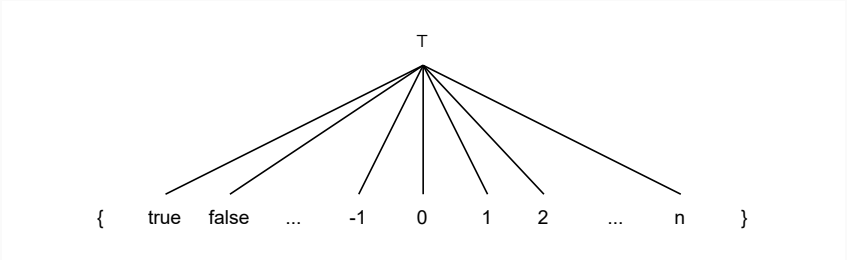


Figure 3: Lattice used by SPMDClang's constant propagation step

Results of Constant Propagation Step

- If the constant propagation was succcessful a CFG instance for each MPI process can be created which only contains code that can be reached by this process
- Additionally: the plugin is now also able to evaluate the specific values in a MPI function call (such as `source`, `destination`, `tag`)
- We now have fulfilled all requirements to create a CPN that models the MPI communication of the given program for the chosen number of processes

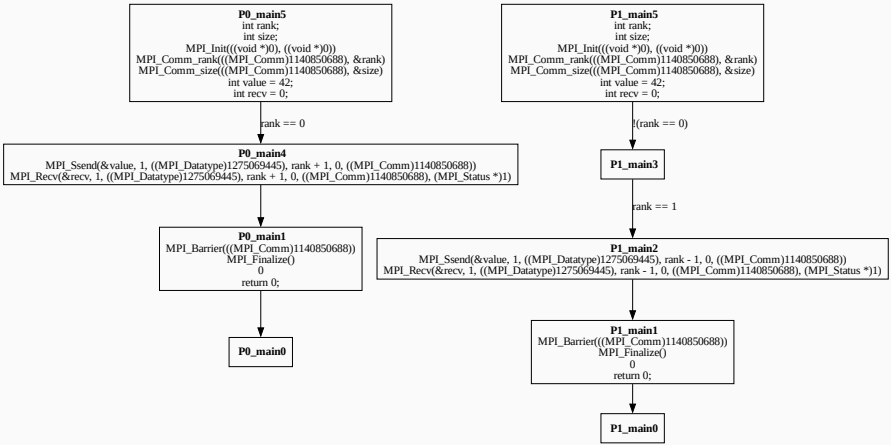
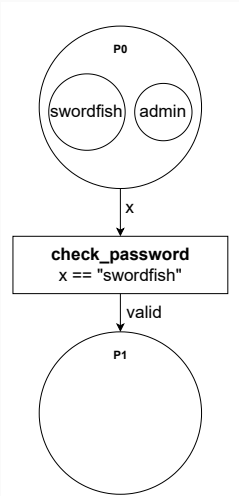


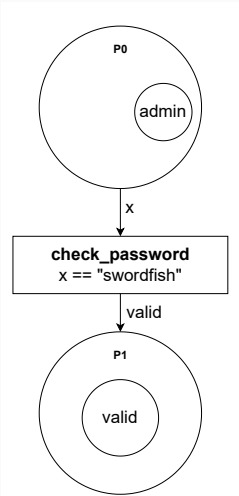
Figure 4: CFG instances for the example code with 2 MPI processes

Outline

- Petri nets have the following components:
- **Places**: Can contain tokens and are connected to transitions
- **Transitions**: Consume tokens when activated and can have conditions for accepted tokens (guards)
- **Tokens**: Can contain arbitrary data
- **Arcs**: Connect places and transitions. The labels are either a variable that binds to an incoming token or the the token type that is produced by a transition

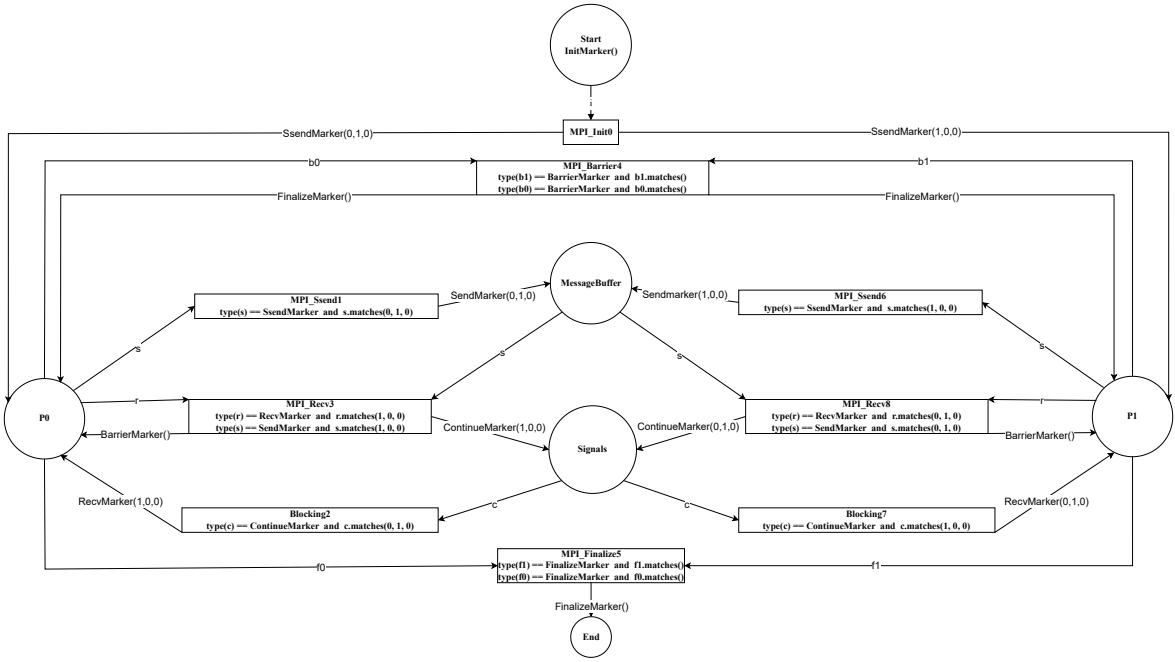


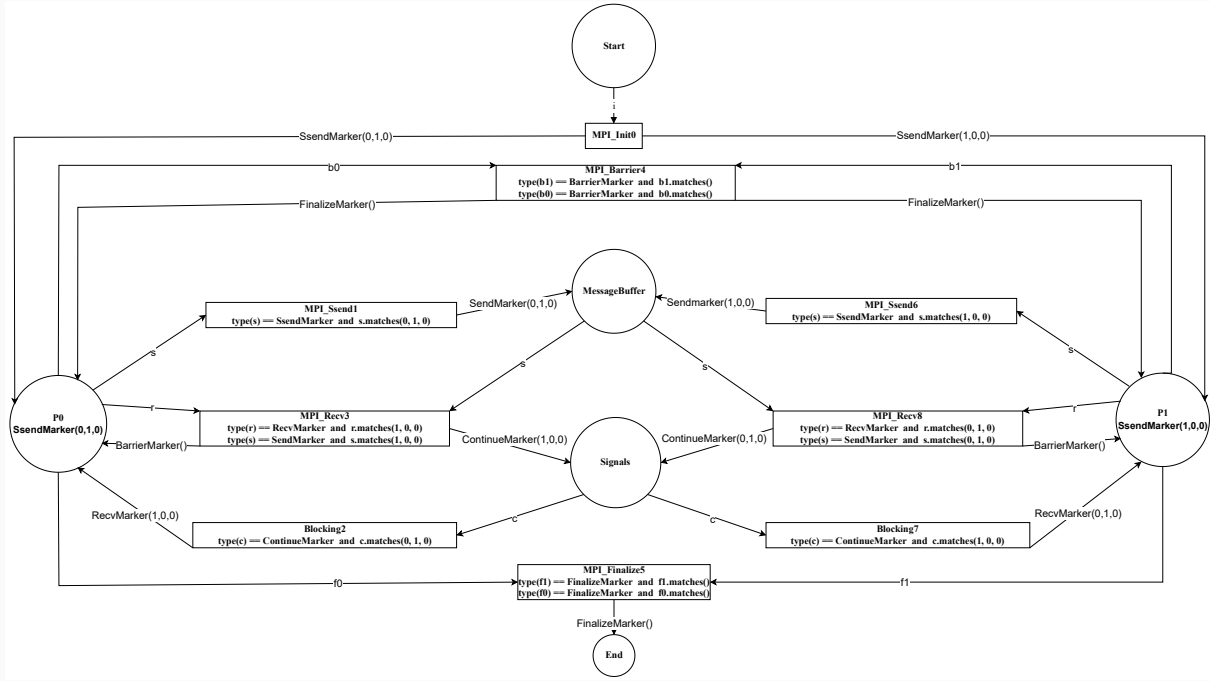
- Petri nets have the following components:
- **Places**: Can contain tokens and are connected to transitions
- **Transitions**: Consume tokens when activated and can have conditions for accepted tokens (guards)
- **Tokens**: Can contain arbitrary data
- **Arcs**: Connect places and transitions. The labels are either a variable that binds to an incoming token or the the token type that is produced by a transition

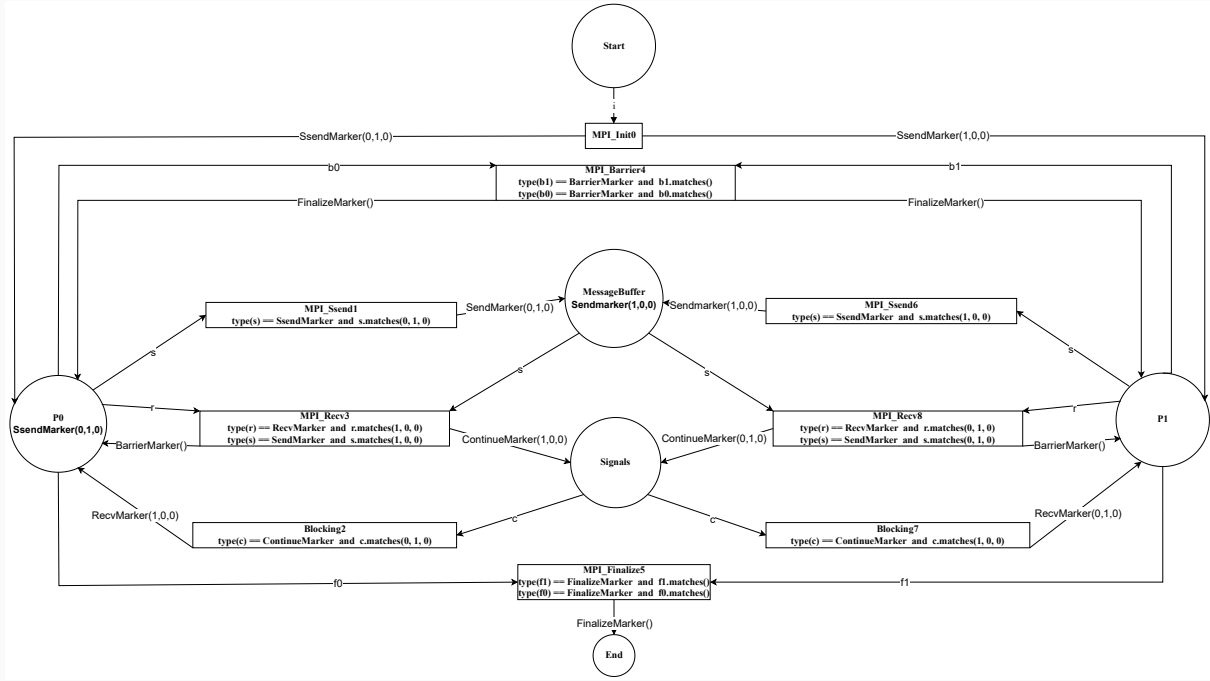


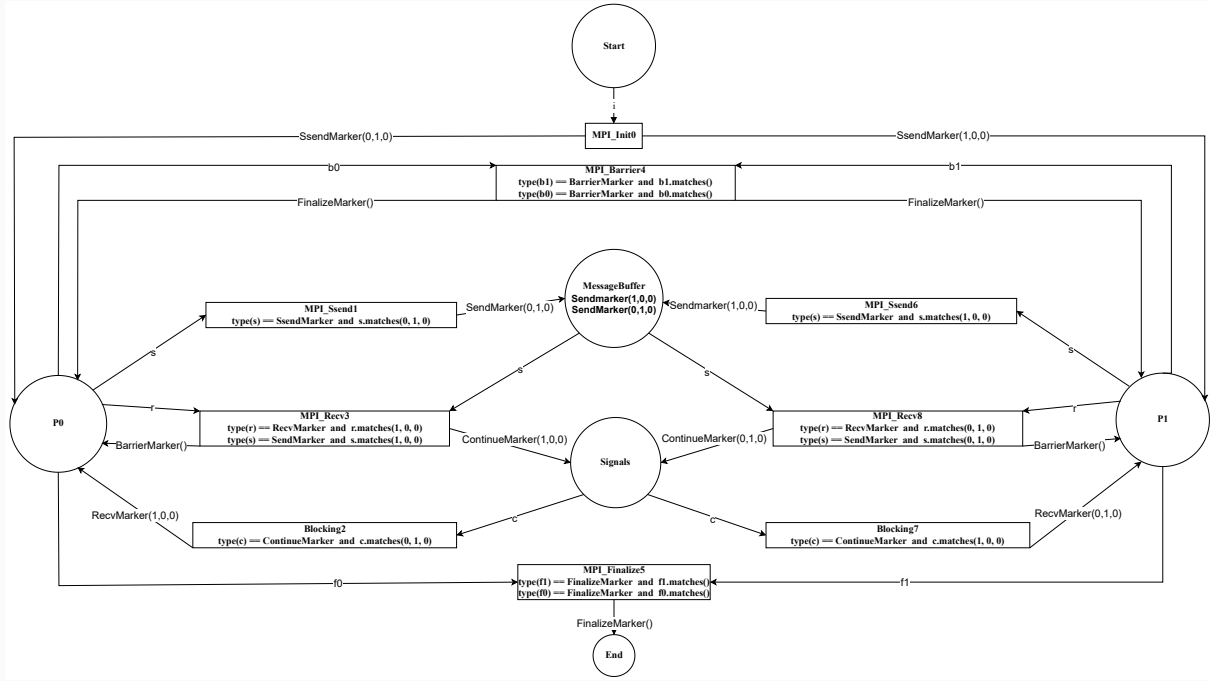
CPN Analysis

- The CFG instances from the previous step are used to create a CPN
- Each process is assigned a place in the net
- MPI operations are modeled as transitions
- The tokens model the MPI function calls in the program
- During the simulation each possible order of execution of communication operations is considered
- The final states of the CPN are then analyzed to detect errors in the modeled communication scheme


Start
InitMarker()
MPI_Init0
SsendMarker(0,1,0)
SsendMarker(1,0,0)
b0
b1
MPI_Barrier4
type(b1) == BarrierMarker and b1.matches()
type(b0) == BarrierMarker and b0.matches()
FinalizeMarker()
FinalizeMarker()
MessageBuffer
SendMarker(0,1,0)
Sendmarker(1,0,0)
MPI_Ssend1
type(s) == SsendMarker and s.matches(0, 1, 0)
MPI_Ssend6
type(s) == SsendMarker and s.matches(1, 0, 0)
s
s
s
s
P0
P1
r
MPI_Recv3
type(r) == RecvMarker and r.matches(1, 0, 0)
type(s) == SendMarker and s.matches(1, 0, 0)
MPI_Recv8
type(r) == RecvMarker and r.matches(0, 1, 0)
type(s) == SendMarker and s.matches(0, 1, 0)
r
BarrierMarker()
ContinueMarker(1,0,0)
ContinueMarker(0,1,0)
BarrierMarker()
Signals
RecvMarker(1,0,0)
c
c
RecvMarker(0,1,0)
Blocking2
type(c) == ContinueMarker and c.matches(0, 1, 0)
Blocking7
type(c) == ContinueMarker and c.matches(1, 0, 0)
f0
f1
MPI_Finalize5
type(f1) == FinalizeMarker and f1.matches()
type(f0) == FinalizeMarker and f0.matches()
FinalizeMarker()
End


Start
MPI_Init0
SsendMarker(0,1,0)
SsendMarker(1,0,0)
b0
b1
MPI_Barrier4
type(b1) == BarrierMarker and b1.matches()
type(b0) == BarrierMarker and b0.matches()
FinalizeMarker()
FinalizeMarker()
MessageBuffer
MPI_Ssend1
type(s) == SsendMarker and s.matches(0, 1, 0)
SendMarker(0,1,0)
Sendmarker(1,0,0)
MPI_Ssend6
type(s) == SsendMarker and s.matches(1, 0, 0)
s
s
s
s
P0
SsendMarker(0,1,0)
r
MPI_Recv3
type(r) == RecvMarker and r.matches(1, 0, 0)
type(s) == SendMarker and s.matches(1, 0, 0)
BarrierMarker()
ContinueMarker(1,0,0)
Signals
ContinueMarker(0,1,0)
MPI_Recv8
type(r) == RecvMarker and r.matches(0, 1, 0)
type(s) == SendMarker and s.matches(0, 1, 0)
r
BarrierMarker()
P1
SsendMarker(1,0,0)
RecvMarker(1,0,0)
c
c
RecvMarker(0,1,0)
Blocking2
type(c) == ContinueMarker and c.matches(0, 1, 0)
Blocking7
type(c) == ContinueMarker and c.matches(1, 0, 0)
f0
f1
MPI_Finalize5
type(f1) == FinalizeMarker and f1.matches()
type(f0) == FinalizeMarker and f0.matches()
FinalizeMarker()
End


Start
MPI_Init0
SsendMarker(0,1,0)
SsendMarker(1,0,0)
b0
b1
MPI_Barrier4
type(b1) == BarrierMarker and b1.matches()
type(b0) == BarrierMarker and b0.matches()
FinalizeMarker()
FinalizeMarker()
MessageBuffer
Sendmarker(1,0,0)
MPI_Ssend1
type(s) == SsendMarker and s.matches(0, 1, 0)
SendMarker(0,1,0)
Sendmarker(1,0,0)
MPI_Ssend6
type(s) == SsendMarker and s.matches(1, 0, 0)
s
s
s
s
P0
SsendMarker(0,1,0)
r
MPI_Recv3
type(r) == RecvMarker and r.matches(1, 0, 0)
type(s) == SendMarker and s.matches(1, 0, 0)
BarrierMarker()
ContinueMarker(1,0,0)
Signals
ContinueMarker(0,1,0)
MPI_Recv8
type(r) == RecvMarker and r.matches(0, 1, 0)
type(s) == SendMarker and s.matches(0, 1, 0)
r
BarrierMarker()
P1
RecvMarker(1,0,0)
c
c
RecvMarker(0,1,0)
Blocking2
type(c) == ContinueMarker and c.matches(0, 1, 0)
Blocking7
type(c) == ContinueMarker and c.matches(1, 0, 0)
f0
f1
MPI_Finalize5
type(f1) == FinalizeMarker and f1.matches()
type(f0) == FinalizeMarker and f0.matches()
FinalizeMarker()
End


Start
i
MPI_Init0
SsendMarker(0,1,0)
SsendMarker(1,0,0)
b0
b1
MPI_Barrier4
type(b1) == BarrierMarker and b1.matches()
type(b0) == BarrierMarker and b0.matches()
FinalizeMarker()
FinalizeMarker()
MessageBuffer
Sendmarker(1,0,0)
SendMarker(0,1,0)
MPI_Ssend1
type(s) == SsendMarker and s.matches(0, 1, 0)
SendMarker(0,1,0)
Sendmarker(1,0,0)
MPI_Ssend6
type(s) == SsendMarker and s.matches(1, 0, 0)
s
s
s
s
P0
P1
r
r
MPI_Recv3
type(r) == RecvMarker and r.matches(1, 0, 0)
type(s) == SendMarker and s.matches(1, 0, 0)
MPI_Recv8
type(r) == RecvMarker and r.matches(0, 1, 0)
type(s) == SendMarker and s.matches(0, 1, 0)
BarrierMarker()
ContinueMarker(1,0,0)
ContinueMarker(0,1,0)
BarrierMarker()
Signals
RecvMarker(1,0,0)
c
c
RecvMarker(0,1,0)
Blocking2
type(c) == ContinueMarker and c.matches(0, 1, 0)
Blocking7
type(c) == ContinueMarker and c.matches(1, 0, 0)
f0
f1
MPI_Finalize5
type(f1) == FinalizeMarker and f1.matches()
type(f0) == FinalizeMarker and f0.matches()
FinalizeMarker()
End

Error Detection and Categorization

- The terminal states of the CPN are analyzed for errors in the communication scheme
- No token in the *End* state means a deadlock somewhere
- Remaining tokens in any other place mean that some message was not received

```
pars-example.c:21:9: warning: Deadlock detected. Rank P0 stuck at:
   21 |  MPI_Ssend(&value, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
      |  ^
pars-example.c:25:9: warning: Deadlock detected. Rank P1 stuck at:
   25 |  MPI_Ssend(&value, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD);
```

Listing 2: Compiler warnings generated for the example code

Outline

Currently Supported Features

- SPMDClang can detect *deadlocks* and *lost messages* for the supported MPI operations
  - `MPI_Ssend`, `MPI_Bsend`, `MPI_Recv`, `MPI_Barrier` currently implemented
- General MPI semantics such as *envelope-matching* and *non-overtaking messages* are working correctly
- These cover a lot of the possible semantics of MPI operations
- *non-blocking* and *one-sided* operations are still to be investigated

Current Limitations

- SPMDClang is still in development and is currently not yet able to be used on 'real life' applications
- Not all language concepts of C/C++ are supported yet (most importantly *loops*)
- Runtime dependent variables which are not MPI `rank/size` that influence the control flow need to be handled
  - If the value can not be evaluated we have to simulate all possible ways it can influence the control flow
  - Most likely either via producing multiple CPNs or by adding this functionality to the CPN directly
- As a *Clang* plugin we are (mostly) limited to analyzing one translation unit

Compile Time Overhead

- The time requirement of exploring all possible orders of execution for a CPN grows very quickly with the number of processes being simulated
- This makes it not very feasible to use for larger process counts
- However, it should be sufficient to simulate for a limited number of processes to detect general errors in a communication scheme

Processes	Time (s)	States	Edges	Processes	Time (s)	States	Edges
Baseline	0.102s	-	-				
4	0.27s	58	114	10	23.38s	23,170	115,842
8	2.49s	3,106	12,418	11	63.45s	54,818	299,042
9	6.35s	7,346	32,722	12	235.40s	172,930	1,037,570

Table 1: Compile times for a ring communication program with different numbers of processes

Outline

Conclusion

- We have shown that the approach works for a limited subset of MPI operations and C/C++ language concepts
- Many of the missing features only require time to implement because they work in a similar way to the already implemented ones
- Most important next step is to add enough functionality to be able to run on 'real' programs
  - Support for *loops* is the most important feature in that regard