Modelling MPI Communication using Colored Petri Nets

PARS-Workshop 2023

Michael Blesel, Tronje Krabbe, Michael Kuhn

michael.blesel@ovgu.de, tronje.krabbe@studium.uni-hamburg.de, michael.kuhn@ovgu.de

September 13, 2023

Outline

Introduction

- Work based on Master's thesis by Tronje Krabbe
- Speaker: Michael Blesel
  - Research associate at University Magdeburg
  - Parallel Computing and I/O working group
- The presented work is foundational research for a larger static analysis project about correctness checking of MPI codes

Motivation

- Developing MPI applications is difficult and introduces many new error types
- MPI is required for for many large scientific applications
- In HPC, implementations are often written by domain scientists with limited experience in computer science
- Therefore, tools for error detection are highly desirable

Deadlock

```
if (rank == 0) {
        MPI_Recv(&buf, 1, MPI_INT, 1, 1, /* ... */);
        MPI_Send(&buf, 1, MPI_INT, 1, 0, /* ... */);
} else if (rank == 1) {
        MPI_Recv(&buf, 1, MPI_INT, 0, 0, /* ... */);
        MPI_Send(&buf, 1, MPI_INT, 0, 1, /* ... */);
}
```

Unmatched Communication Operation

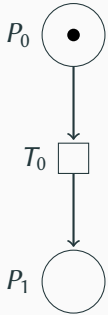
```c
if (rank == 0) {
        MPI_Send(&buf, 1, MPI_INT, 1, 0, /* ... */);
} else if (rank == 1) {
        /* do nothing */
}
```

Rank, Tag, or Type Mismatch

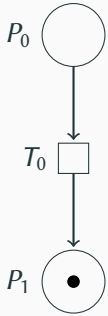
```
if (rank == 0) {
        MPI_Send(&buf, 1, MPI_DOUBLE, 1, 0, /* ... */);
} else if (rank == 1) {
        MPI_Recv(&buf, 1, MPI_INT, 0, 0, /* ... */);
}
```

Outline

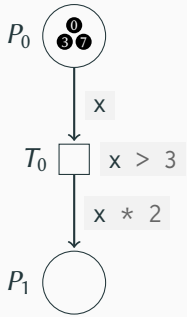
Petri Nets [Petri, 1962]



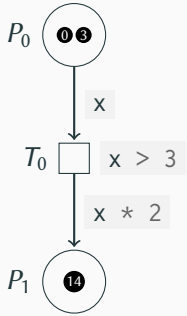
Petri Nets [Petri, 1962]



Transition Guards



Transition Guards

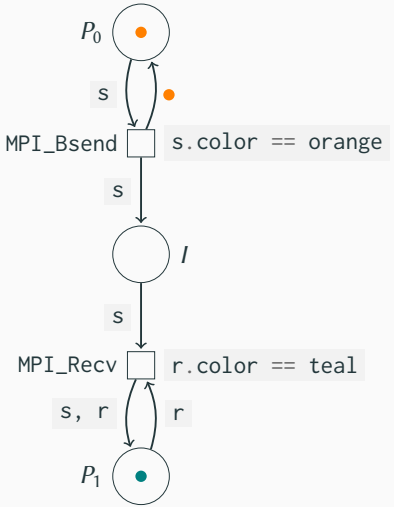


Outline

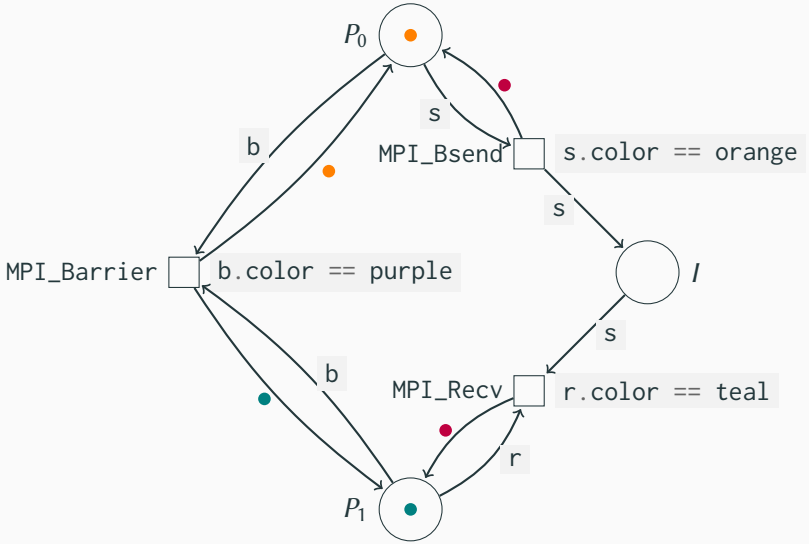
Communication Modes

- MPI provides multiple communication modes, such as:
  - standard (e.g. `MPI_Send`)
  - buffered (e.g. `MPI_Bsend`)
  - synchronous (e.g. `MPI_Ssend`)
  - ready (e.g. `MPI_Rsend`)
- These need to be modelled differently to reproduce the correct communication semantics

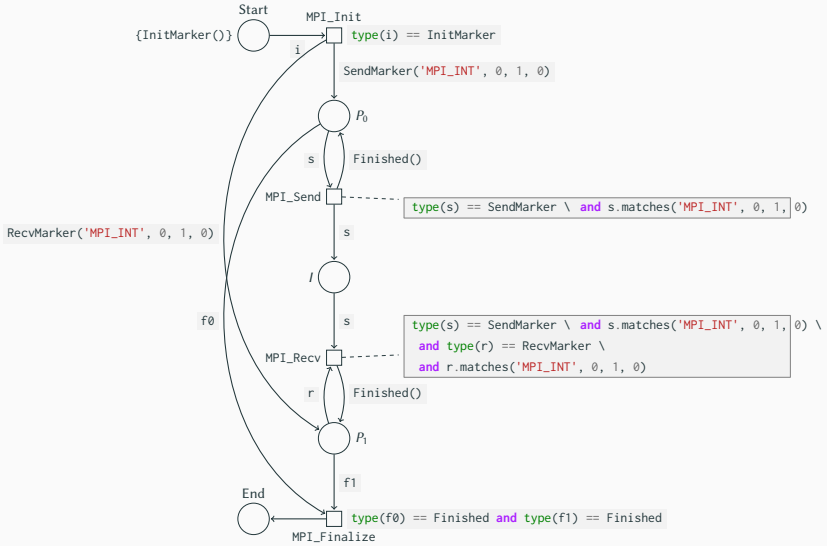
Buffered Send



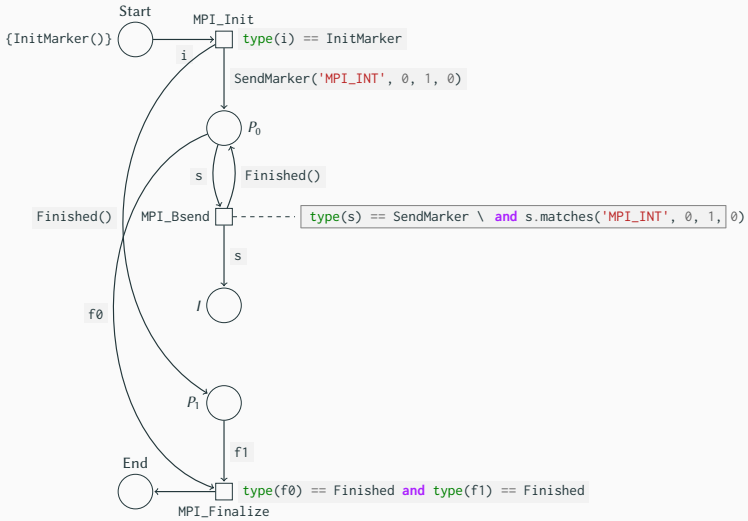
Barrier


P_0
s
b
MPI_Bsend
s.color == orange
s
MPI_Barrier
b.color == purple
I
s
b
MPI_Recv
r.color == teal
r
P_1

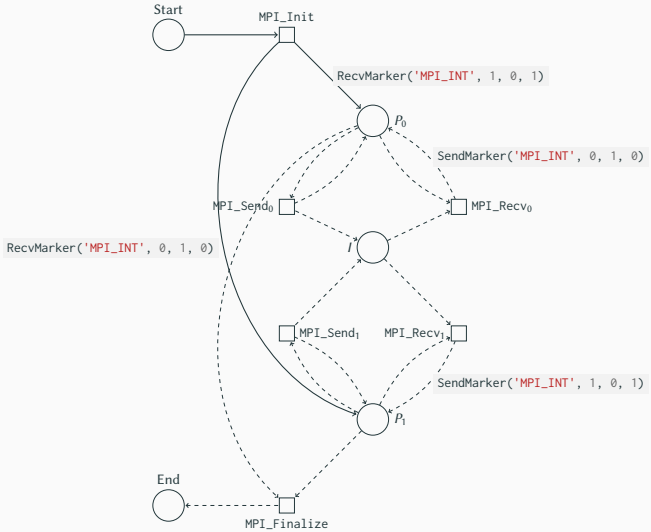
Simple Send


Start
{InitMarker()}
MPI_Init
type(i) == InitMarker
i
SendMarker('MPI_INT', 0, 1, 0)
P_0
s
Finished()
MPI_Send
type(s) == SendMarker \ and s.matches('MPI_INT', 0, 1, 0)
RecvMarker('MPI_INT', 0, 1, 0)
s
I
f0
s
type(s) == SendMarker \ and s.matches('MPI_INT', 0, 1, 0) \
and type(r) == RecvMarker \
and r.matches('MPI_INT', 0, 1, 0)
MPI_Recv
r
Finished()
P_1
f1
End
type(f0) == Finished and type(f1) == Finished
MPI_Finalize

Unmatched Bsend



Deadlock


Start
MPI_Init
RecvMarker('MPI_INT', 1, 0, 1)
P_0
SendMarker('MPI_INT', 0, 1, 0)
MPI_Send$_0$
MPI_Recv$_0$
RecvMarker('MPI_INT', 0, 1, 0)
I
MPI_Send$_1$
MPI_Recv$_1$
SendMarker('MPI_INT', 1, 0, 1)
P_1
End
MPI_Finalize

Modelling Control Flow

- How can the control flow of an application be modelled?
- Construct a CPN model for each possible path through the program (“symbolic execution”)
- Drawback: state-space explosion and thus performance implications
- Possible solution: simulate only parts of a program?

Outline

Original program

```
MPI_Init();

/* snip */

if (rank == 0) {
        MPI_Send(&send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
        MPI_Recv(&recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                                                MPI_STATUS_IGNORE);
}

MPI_Finalize();
```

Intermediate Representation

- CPN creation takes an Intermediate representation as input

```
0x0000 MPI_Init()
0x0001 MPI_Send(process=0, to=1, type='MPI_INT', tag=0, next=0x0003)
0x0002 MPI_Recv(process=1, from=0, type='MPI_INT', tag=0, next=0x0003)
0x0003 MPI_Finalize()
```

CPN Libraries

We developed two CPN libraries written in Rust

1. General-purpose CPN Library
   - Can be used to construct and evaluate any kind of CPN
   - Python as modelling language ⇒ performance is not great
2. Special-purpose CPN Library
   - Specifically tailored towards modelling MPI using the presented approach
   - Removed modelling language ⇒ about 20 times faster

[1] https://gitlab.com/tronje/cpn-rs
[2] https://gitlab.com/tronje/mpi-cpnets

IR Parser & Evaluator

- Parses IR and constructs a CPN
- Explores the state space of the CPN to detect logic errors in the MPI communication
- Is able to identify the errors presented earlier

[1] `https://gitlab.com/tronje/mpi-cpn-ir-parser`

Performance Overview

processes	state graph size	time	memory
2	6	0.05 s	18.0 KiB
4	72	0.05 s	168.5 KiB
6	990	0.07 s	4.4 MiB
8	13 776	0.88 s	107.5 MiB
10	191 862	19.80 s	2.3 GiB

- The evaluated model was of an implementation of the Jacobi method.
- Measurements taken on a system with an AMD Ryzen 5 3600X CPU, running Linux and Rust 1.70.0.

Outline

Caveats

- Only supported operations so far: `MPI_Init`, `MPI_Finalize`, `MPI_Bsend`, `MPI_Ssend`, `MPI_Recv`, `MPI_Allreduce`
  - preliminary work on non-blocking operations (e.g. `MPI_Isend`)
- Only tested on simple error examples and one “real world” program (a Jacobi method implementation)
- Translation from C to IR is missing
- Parallelization could be added for performance gains

Future Work

- Automatic translation of source code into IR format
- Support for the remaining MPI operations
- Performance optimizations
- **Ultimate goal**: Incorporate this work into a Clang based tool for compile time correctness checking of MPI applications

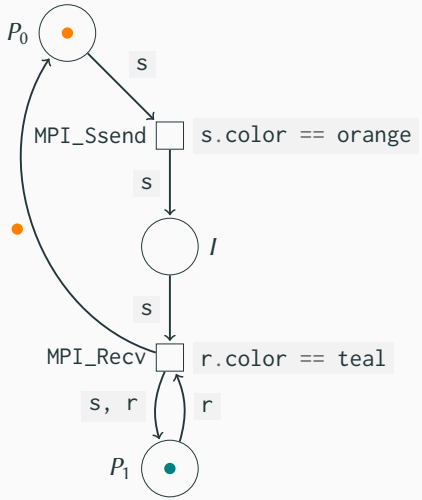
Conclusion

- Proof-of-concept that CPNs can be used to model MPI applications
- Communication between processes can be simulated and analysed
- Logical errors in communication schemes can be detected
- Automatic translation from source code into CPN model still needs to be worked on

References

[Petri, 1962] Petri, C. A. (1962). **Kommunikation mit Automaten.**

Synchronous Send



Unmatched Ssend