MASTERTHESIS

Modelling MPI Communication using Colored Petri Nets

vorgelegt von

Tronje Krabbe

tronje.krabbe@studium.uni-hamburg.de

MIN-Fakultät

Fachbereich Informatik

Scientific Computing

Studiengang: Informatik

Matrikelnummer: 6435002

Abgabedatum: 24.07.2023

Erstgutachter: Prof. Dr. Michael Kuhn

Zweitgutachter: Prof. Dr. Thomas Ludwig

Betreuer: Michael Blesel

Abstract

The Message Passing Interface (MPI) is a widely adopted standard for use in high-performance, distributed memory, parallel computing. Due to its high complexity, caused by complex communication patterns and often unknown numbers of participating processes, both manual and automated error detection for MPI software is difficult.

Colored Petri Nets (CPNs) offer a powerful, high-level modelling framework to model complex systems. Arbitrary annotations in a modelling language allow a CPN to model almost anything. They are suited particularly well for modelling distributed systems, making them a good match for MPI software.

This thesis presents a novel approach for modelling the communication patterns of MPI programs as Colored Petri Nets. It includes a fully-featured CPN implementation, a proposed specification of an intermediate representation for compiling MPI software into CPNs, as well as a proof-of-concept evaluator tool which can analyze the proposed IR and detect a variety of programming errors.

The evaluator is as yet incomplete, supporting only a very small subset of the MPI standard, but the modelling approach should transfer to most MPI operations, and the performance of the software is promising. Other approaches to modelling and checking MPI software are either slower or defunct.

Contents

1 Introduction **1**
1.1 General Context 1
1.2 Rationale 2
1.3 Leading Questions 2
1.4 Overview 3

2 Background **5**
2.1 Message Passing Interface 5
2.1.1 MPI Programming Errors 7
2.1.2 MPI Functionality Usage Statistics 10
2.2 Petri Nets 13
2.3 Colored Petri Nets 15
2.3.1 Color 15
2.3.2 Modelling Language 15
2.3.3 Transition Guards 16
2.4 Python 17

3 Modelling MPI Communication **19**
3.1 Communication Modes 19
3.2 Modelling Individual MPI Operations 20
3.2.1 Point-to-Point Operations 20
3.2.2 Collective Operations 25
3.3 Leveraging Higher-Level CPN Functionality 28
3.4 Modelling MPI Programs 31
3.4.1 Modelling Conditional Code 31
3.4.2 Modelling Error Examples 33
3.5 One-sided MPI Communication 39

4 Intermediate Representation **41**
4.1 Design Considerations 42
4.1.1 Places 42
4.1.2 Transitions 42
4.1.3 Arcs 43
4.2 Approach 44

5 Implementation **47**
5.1 Accessing the Source Code . . . 47
5.2 The cpnets Crate . . . 48
5.2.1 Net . . . 48
5.2.2 Marking . . . 50
5.2.3 State Graph . . . 50
5.2.4 Python Integration . . . 53
5.3 Proof-of-concept IR Evaluator . . . 54
5.3.1 Inferring Error Locations in Code . . . 54

6 Evaluation **57**
6.1 Error Examples . . . 57
6.1.1 Deadlocks . . . 57
6.1.2 Unmatched Send . . . 57
6.1.3 Unmatched Receive . . . 59
6.1.4 Rank, Tag or Type Mismatch . . . 61
6.1.5 Conclusions . . . 61
6.2 Jacobi Method . . . 62
6.2.1 State Graph Size . . . 63
6.2.2 Conclusions . . . 64
6.3 Performance . . . 66
6.3.1 Conclusions . . . 66
6.4 Revised Implementation . . . 68
6.4.1 Removing Python as the Modelling Language . . . 68
6.4.2 Improved Performance . . . 71
6.4.3 Accessing the Source Code . . . 71

7 Related Work **73**
7.1 CIVL . . . 73
7.1.1 Evaluation . . . 74
7.1.2 Performance . . . 75
7.1.3 Conclusion . . . 76
7.2 MPI-SV . . . 76
7.2.1 Evaluation . . . 77
7.3 MPI-Checker . . . 78
7.4 In-Situ Partial Order . . . 80
7.5 Kaira . . . 81
7.6 SNAKES . . . 81

8 Outlook **83**
8.1 Modelling . . . 83
8.1.1 Missing MPI Operations . . . 83
8.1.2 Symbolic Execution . . . 83

8.2 Implementation . . . 84
8.2.1 Missing MPI Operations . . . 84
8.2.2 Compilation . . . 84
8.2.3 Performance Optimizations . . . 84
8.2.4 Large-scale Evaluation . . . 85
8.2.5 Integration into other Tools . . . 85

9 Conclusion **87**

Bibliography **89**

List of Figures **93**

List of Listings **95**

List of Tables **97**

1 Introduction

This document details an M. Sc. thesis project that attempts to simulate Message Passing Interface (MPI) communication using Colored Petri Nets (CPNs). This chapter provides a general overview of both the project in general and this document specifically. Section 1.1 will introduce the context of the work, briefly describing MPI and CPNs and their uses. Section 1.2 on the next page will explain why pursuing this project was desirable. Section 1.3 on the following page poses four leading questions that will be answered throughout this document. Section 1.4 on page 3 concludes the chapter by detailing the structure of this document and giving an overview of the following chapters.

1.1 General Context

The thesis project is done in the context of High-Performance Computing (HPC). The Message Passing Interface is a standard for parallel computing in distributed-memory systems, often employed by supercomputers.

Implementing MPI communication correctly is difficult for various reasons (cf. e.g. Gorlatch, 2004; Gropp, 2001). The MPI compiler lacks compile-time logic checks. The OpenMPI manual for the compiler *mpicc* states: "mpicc is a convenience wrappers [*sic*] for the underlying C compiler" (The Open MPI Project, 2021). It does not check, for example, whether all send operations have a corresponding receive operation elsewhere in the program code. Or whether there are circular communication patterns that can cause deadlocks. The soundness of an MPI program may depend on the number of processes involved, which can vary between invocations. Perhaps an MPI program is only correct when an even number of processes are involved.

Colored Petri Nets provide a powerful modelling tool. According to Jensen and Kristensen, 2009, p. 3, "typical application domains of CP-nets are communication protocols, data networks, distributed algorithms, and embedded systems". They state that CPNs are "also applicable more generally for modelling systems where concurrency and communication are key characteristics". These are obviously characteristics of MPI programs. However, CPNs are relatively simple. A CPN is just a directed graph with two types of nodes and some additional features. This makes a CPN quite easy to display visually, which can help with the modelling process. The inclusion of annotations in a modelling language such as CPN ML makes CPNs essentially equivalent to a high-level programming language.

Part of the thesis project is an implementation of Colored Petri Nets. This implementation

is then used to model a variety of MPI programs, and to evaluate these models.

1.2 Rationale

The thesis aims to investigate the feasibility of translating MPI communication patterns into CPNs and using the resulting models to analyze the communication. The ultimate goal of this effort is to construct nets that accurately model the MPI communication they represent and can thus be used to detect logic errors in an MPI program.

It is explicitly not within the scope of the thesis to implement a fully automated translation of MPI programs into CPNs and subsequent analysis. However, this goal is kept in mind throughout the project, and it will attempt to provide the foundation for an automated process.

1.3 Leading Questions

This thesis document will answer the following leading questions.

Which individual MPI operations can be modelled, and how?

Answering this question entails creating CPN models for individual MPI operations. These models will not represent real-world MPI programs but will lay the groundwork for modelling more complex MPI software.

To determine whether an MPI operation can be modelled, a candidate CPN must be analyzed and shown to represent the operation accurately. This can likely be done by exploring its possible state space.

Because the MPI standard defines well over one hundred operations (cf. Message Passing Interface Forum, 2021), this thesis will cover only the most-used operations. It is possible that not all of these can be modelled accurately using CPNs.

Can more complex MPI programs with multiple MPI operations be modelled, and if so, how?

This question is the logical continuation of the previous one. If at least some individual MPI operations can be modelled, then an entire MPI program can likely be modelled, too. This will entail more complexity as the model is no longer concerned with just the pure MPI communication but also the control flow of the program, and the interaction between different MPI operations.

What information from an MPI program is relevant for creating a CPN representation?

This question ties in with the previous one. To accomplish a fully automated translation from MPI program to CPN at some point in the future, as mentioned in Section 1.2 on the facing page, an exact specification of the information that needs to be extracted from the program to facilitate the construction of a model is required.

What kinds of errors can be detected using the CPN-based modelling approach, and how?

Since the project aims to model MPI communication for error detection, the different kinds of programmer errors need to be collected, and each must be evaluated. The error types that CPNs can detect and how exactly this detection is possible shall be documented.

1.4 Overview

This section will give a short overview of the structure of the remaining chapters in this document. Chapter 2 on page 5 will continue with more in-depth information on MPI, CPNs and other concepts and technologies used for the thesis project. Chapter 3 on page 19 presents the modelling approach central to the thesis, and shows how individual MPI operations and entire MPI programs can be represented as CPNs. Chapter 4 on page 41 will propose an intermediate representation that can be used in future work in a translation pipeline from C program to CPN. Chapter 5 on page 47 provides insights into the technical aspects of the implementation. Chapter 6 on page 57 evaluates the merits of the thesis project and will show both the strengths and shortcomings of the modelling approach used. Chapter 7 on page 73 lists and compares related work and similar approaches to this thesis. Chapter 8 on page 83 and Chapter 9 on page 87 will conclude the document by giving and overview of possible future work on the thesis topic and summarizing the work done.

2 Background

This chapter will provide the theoretical and technical background for the thesis. First, Section 2.1 will describe the Message Passing Interface. The other defining concept of the thesis is the Colored Petri Net. Section 2.2 on page 13 will cover the the basic Petri Nets, before Section 2.3 on page 15 will describe Colored Petri Nets. Finally, as the Colored Petri Nets used for this thesis use Python as their modelling language, a short introduction to the Python programming language is given in Section 2.4 on page 17.

2.1 Message Passing Interface

The Message Passing Interface (MPI) (cf. Message Passing Interface Forum, 2021) is a standard for parallel computing in distributed-memory systems. MPI allows parallelization of software not just on one machine with multiple processor cores but across many machines, such as the nodes of a supercomputer. MPI standard implementations exist for various languages, but the standard provides library routines for C and Fortran. This thesis will focus on MPI programs written in C and using Version 4.0 of the MPI standard. There are competing implementations of the MPI standard, like OpenMPI[1] and MPICH[2]. Code written and tested for this thesis was built using OpenMPI, version 4.1.5.

MPI is used to specify the communication between processes. To this end, the standard includes, according to New Mexico State University, 2021, the following:

- Point-to-point communication, i.e., the sending of messages from one specific process to another.
- Collective communication, i.e., operations that involve a specific group of processes, or all processes
- Process groups
- Communicators
- Process topologies
- Language bindings for C and Fortran, as previously mentioned
- A profiling interface

[1]https://www.open-mpi.org/, accessed July 11, 2023
[2]https://www.mpich.org/, accessed July 11, 2023

```c
#include <mpi.h>

int main(void)
{
        int size, rank;
        int send = 42;
        int recv;

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

        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();
        return 0;
}
```

Listing 2.1: A simple MPI program that demonstrates `MPI_Send` and `MPI_Recv`.

- Miscellaneous environmental and inquiry functions

This thesis will focus on the first two points from the list above, namely point-to-point and collective communication. The remaining elements either specify process groupings that then use one of the mentioned communication methods or things that are unrelated to runtime communication, such as the exact language bindings or the profiling interface.

Listing 2.1 shows a simple MPI program that makes use of point-to-point communication. `MPI_Send` is used by a process to send a message to another process. To receive the message, this other process must invoke `MPI_Recv`. Note that most MPI functions can return an error code and that the program assumes that the number of processes in the `MPI_COMM_WORLD` communicator is equal to two. Both potential errors and the number of processes go unchecked in this example program for brevity. Other code examples in this thesis document will follow this convention.

An MPI communicator is a set of processes. The `MPI_COMM_WORLD` communicator contains all processes participating in the MPI program.

Listing 2.2 on the facing page demonstrates a collective operation, `MPI_Allreduce`. All

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

int main(void) {
        int rank, size, max_rank;

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

        MPI_Allreduce(&rank, &max_rank, 1,
                MPI_INT, MPI_MAX,
                MPI_COMM_WORLD);

        printf("process %d: the highest rank is: %d\n",
                        rank, max_rank);

        MPI_Finalize();
        return 0;
}
```

Listing 2.2: A simple MPI program that demonstrates `MPI_Allreduce`.

processes in the program invoke this operation, with their rank as the argument. The operation returns the maximum (as indicated by the `MPI_MAX` argument) of all its inputs to each process.

2.1.1 MPI Programming Errors

Nguyen Ba and Arora, 2018 list five types of logical errors in MPI programs:

- Incorrect initialization of variables
- Lack of code handling different numbers of processes
- Incorrect distribution of non-contiguous data
- Incorrect choice of collective operations for data distribution
- Incorrect handling of `break` statements in loops

This thesis will largely ignore these errors because they are indirectly related to MPI communication. A model of just the communication, such as the one this thesis proposes, will never be able to detect an incorrectly initialized variable. Instead, the project aims to identify the following types of errors reliably. Section 6.1 on page 57 will show that the modelling approach developed for the thesis project can detect all of these.

```
#include <mpi.h>

int main(void) {
        int size, rank, recv, send = 42;

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

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

        MPI_Finalize();
        return 0;
}
```

Listing 2.3: Two processes deadlock because they each call `MPI_Recv` and wait for one another indefinitely.

Deadlocks

A deadlock occurs when one process begins a communication operation that requires participation by another process, but this other process is waiting in a different operation for the first process. Neither process can continue, and the program stops its computation indefinitely.

Listing 2.3 shows a simple MPI program, which assumes two participating processes. Both call `MPI_Recv` before `MPI_Send`, which results in both processes waiting on the other indefinitely. The `MPI_Send` operations will never be executed.

Unmatched Send

An unmatched, synchronous send, e.g., `MPI_Send` in some cases, or `MPI_Ssend`, will cause a process to wait indefinitely. The send operation can only return once some other process has started a matching receive. But if there is no matching receive in the program, the operation will not return. This is a bug in the MPI program.

```c
#include <mpi.h>

int main(void) {
        int size, rank, out = 42;

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

        if (rank == 0) {
                MPI_Ssend(&out, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        } else {
                /* logic error: missing receive */
        }

        MPI_Finalize();
        return 0;
}
```

Listing 2.4: An example of an unmatched send operation in an MPI program.

An unmatched, buffered send, e.g., `MPI_Bsend`, does not have to wait for a matching receive operation. The sending process can continue almost immediately. However, it is still most likely a bug in the MPI program when such a send with no receiver exists. It was surely the intention of the programmer for the sent value to reach some other process.

Non-blocking MPI send operations, such as `MPI_Ibsend`, will always return immediately, but any subsequent `MPI_Wait` on the created request object will then never complete.

Listing 2.4 shows a short example program with an unmatched send operation.

Unmatched Receive

When a process initiates a receive operation with no matching send operation, the receive operation will never complete. This is the inverse of the unmatched send described in Section 2.1.1 on the facing page.

Rank or Tag Mismatch

Any send or receive operation must specify which rank to send to or receive from and a tag for the message. If the programmer accidentally specifies an incorrect rank or tag, they can create a combination of an unmatched send and unmatched receive.

```
keyword: mpi
language: c,c++
stars: >2
pushed: >2013-01-01
```

Listing 2.5: AGSearch repository query file.

2.1.2 MPI Functionality Usage Statistics

To better understand MPI usage, and to focus on the most-used MPI functions, the tool AGSearch[3] was used with the repository query shown in Listing 2.5. The query searches the popular git[4] hosting website GitHub[5] for repositories tagged with the keyword `mpi` that contain code in either of the two languages, C or C++. Repositories must have more than two stars on GitHub to filter out repositories with almost no popularity. Additionally, the repositories must have had commits (i.e., updates) pushed to them some time since January 1, 2013. AGSearch clones each repository that matches the query and performs a search of its source code. For this analysis, AGSearch was configured to search for each C-Code MPI function, as defined by the OpenMPI documentation[6]. Two additional code search terms were added, the wildcard receive parameters `MPI_ANY_SOURCE` and `MPI_ANY_TAG`. The results of the search are outlined in the following subsections.

Overview

AGSearch found 721 repositories. Of these, though, only 657 contained any of the code search terms. That is, 657 repositories make use of the MPI API. The remainder were likely false positives, such as the project "fuzzball", where the acronym "MPI" is used to refer to "Message Parsing Interpreter"[7], not the Message Passing Interface. After a cursory manual analysis, two repositories, https://github.com/open-mpi/ompi and https://github.com/pmodels/mpich, were identified as implementations of the MPI standard. Both the false positives and the MPI implementations are excluded from further analysis. That leaves a total of 655 repositories considered for further analysis. In these 655 repositories, the sum of all invocations of any MPI function was 913 671.

Wildcard Receives

Wildcard receive operations are those that do not specify either the source rank, tag, or both. They use the special parameters `MPI_ANY_SOURCE` (for rank) and `MPI_ANY_TAG` (for tag). When an MPI process performs a receive using `MPI_ANY_SOURCE`, it will accept a matching

[3] https://github.com/wr-hamburg/AGSearch, accessed July 23, 2023
[4] https://git-scm.com, accessed July 23, 2023
[5] https://github.com, accessed July 23, 2023
[6] https://www.open-mpi.org/doc/current/, accessed February 8, 2023
[7] https://github.com/fuzzball-muck/fuzzball/blob/master/docs/mpi-intro#L27, accessed February 8, 2023

Table 2.1: Usage of MPI Wildcard Parameters

Parameter	Users	% of total
`MPI_ANY_SOURCE`	183	27.9%
`MPI_ANY_TAG`	132	20.2%
Both	115	17.6%
Either	200	30.5%

send from any process, potentially causing non-deterministic communication patterns. Accordingly, a receive with `MPI_ANY_TAG` can potentially match several messages, as any tag will be accepted rather than one specific tag. This can also cause non-determinism. Usage of wildcard receives thus makes modelling MPI communication difficult, as non-determinism can greatly increase the state-space of a model.

The number of repositories using either wildcard parameter, or both, is shown in Table 2.1. Percentage values are rounded to one decimal place.

Most Frequently Used Operations

There are two approaches to identifying the most used MPI communication operations based on the data gathered by AGSearch. One can count the number of repositories that use a given operation at all, or one can count the total number of times an operation is used across all repositories.

Table 2.2 on the next page shows the 20 most common MPI communication operations, both according to how many repositories use them at all (in the left half of the table), and by how many usage instances there are across all analysed repositories (right half). Note that non-communication operations, such as `MPI_Init` and `MPI_File_write`, were filtered out.

Table 2.3 on the following page shows the 25 unique operations extracted from those displayed in Table 2.2 on the next page. The operations are sorted alphabetically.

Table 2.2: The 20 most-used MPI Operations by Users and Usage

Operation	Users	of total	Operation	Usage Instances	of total
MPI_Send	453	69%	MPI_Get	21 857	2.4%
MPI_Barrier	445	68%	MPI_Send	14 977	1.6%
MPI_Recv	444	68%	MPI_Put	12 642	1.4%
MPI_Bcast	409	62%	MPI_Barrier	11 987	1.3%
MPI_Reduce	312	48%	MPI_Wait	11 844	1.3%
MPI_Allreduce	308	47%	MPI_Recv	11 698	1.3%
MPI_Gather	299	46%	MPI_Bcast	10 596	1.2%
MPI_Wait	285	44%	MPI_Reduce	9496	1.0%
MPI_Get	284	43%	MPI_Test	8343	0.9%
MPI_Isend	270	41%	MPI_Alltoall	8163	0.9%
MPI_Irecv	254	39%	MPI_Allreduce	7078	0.8%
MPI_Allgather	219	33%	MPI_Irecv	5415	0.6%
MPI_Scatter	201	31%	MPI_Gather	4558	0.5%
MPI_Waitall	195	30%	MPI_Isend	4494	0.5%
MPI_Gatherv	164	25%	MPI_Win_flush	4423	0.5%
MPI_Test	163	25%	MPI_Win_lock	4212	0.5%
MPI_Sendrecv	157	24%	MPI_Allgather	4102	0.4%
MPI_Alltoall	144	22%	MPI_Win_unlock	3993	0.4%
MPI_Allgatherv	124	19%	MPI_Sendrecv	3735	0.4%
MPI_Scatterv	120	18%	MPI_Alltoallv	3602	0.4%
			Total	167 215	18.3%

Table 2.3: Most-used MPI Operations extracted from Table 2.2

MPI_Allgatherv	MPI_Allgather	MPI_Allreduce	MPI_Alltoallv	MPI_Alltoall
MPI_Barrier	MPI_Bcast	MPI_Gatherv	MPI_Gather	MPI_Get
MPI_Irecv	MPI_Isend	MPI_Put	MPI_Recv	MPI_Reduce
MPI_Scatterv	MPI_Scatter	MPI_Sendrecv	MPI_Send	MPI_Test
MPI_Waitall	MPI_Wait	MPI_Win_flush	MPI_Win_lock	MPI_Win_unlock

2.2 Petri Nets

A Petri Net, first defined by Petri, 1962, and summarized by Peterson, 1977, is a directed graph with two different kinds of nodes: places and transitions. Edges in a Petri Net may only ever connect a place to a transition, or a transition to a place. There can never be an edge between two nodes of the same kind. By convention, edges in Petri Nets are referred to as "arcs". This document will follow this convention.

Places may contain "tokens". Tokens are used to perform simulations with Petri Nets. In any given simulation step, a transition may consume tokens from places that have outgoing arcs to that transition. The transition may also produce new tokens and deposit them into places that have an incoming arc from that transition.

Figure 2.1 shows a straightforward Petri Net. It has two places, P_0, and P_1, as well as one transition, T_0. P_0 contains a single token. T_0 can thus fire once, consuming the token from P_0 and emitting a new token to P_1. Figure 2.2 on the next page depicts the same Petri Net after T_0 has fired once.

Figure 2.3 on the following page shows a Petri Net modelling a traffic light. This net is extended with the notion of *capacity*. Capacity is part of some Petri Net definitions, such as the one presented by Chen, n.d. In this instance, the places "red" and "yellow" are each given a maximum capacity of 1. The place "green" defaults to an infinite capacity.

The traffic light net is deterministic. In the depicted state, only "green" contains a token, and therefore only T_0 can fire. When it does fire, the token from "green" is consumed, and a new token is deposited into "yellow". Then, only T_1 can fire. After it has, "red" will contain the only token in the net. Now, T_2 can fire, consuming the token from "red" and depositing one token each into "red" and "yellow". Now, T_1 and T_2 could fire again, but their target places have already reached the maximum number of tokens they may contain, as specified by their capacity. So only T_3 may fire, consuming both tokens and depositing a token into "green". This puts the net back into its initial state, as depicted in Figure 2.3 on the next page.

Arc weights determine how many tokens a transition needs to consume from its input-places

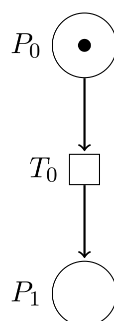


Figure 2.1: A straightforward Petri Net with one place containing a single token, one empty place, and one transition.

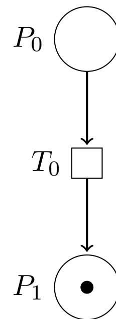


Figure 2.2: The Petri Net from Figure 2.1 on the previous page, after T_0 has fired once.

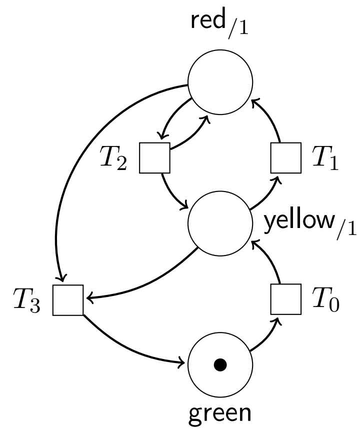


Figure 2.3: A Petri Net modelling a traffic light.

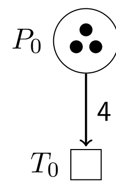


Figure 2.4: A dead Petri Net, due to the weight of the arc from P_0 to T_0.

in order to fire. If these places do not meet the token requirements, the transition cannot fire. When a transition fires, it consumes as many tokens as specified by the arc weights from each input-place. Figure 2.4 on the facing page shows a Petri Net with a place P_0 which contains three tokens. The transition T_0 cannot fire, though, because the arc connecting P_0 to T_0 has a weight of 4.

As showcased in Figure 2.3 on the preceding page, a transition may have more than one incoming arc and more than one outgoing arc. A transition that has incoming arcs from two places, for example, will consume tokens from *both* these places when it fires, not just from either one. Accordingly, the transition will generate and deposit tokens into *all* places to which it connects via outgoing arcs.

2.3 Colored Petri Nets

Colored Petri Nets (sometimes spelled *Coloured* Petri Nets), are what Jensen and Kristensen, 2009 refer to as "high-level Petri Nets". CPNs extend Petri Nets with a number of additional features.

2.3.1 Color

The name-giving feature of CPNs is "color". The "color" feature allows tokens to have a specific value of a specific type. Tokens in CPNs are no longer necessarily indistinguishable. A token can, for example, be of the type "integer" and have the value 42. Or it can be of type "string" and have the value `"hello world"`. The types and values that a token can have depend on the associated Modelling Language of the CPN.

2.3.2 Modelling Language

For the "color" of a token to have a tangible effect, CPNs include modelling language. CPNs, as defined by Jensen and Kristensen, 2009, use a language called CPN ML, derived from Standard ML[8]. The SNAKES project (cf. Pommereau, 2015) uses Python (cf. Python Software Foundation, 2023) as its modelling language. Inspired by this, the CPNs used for this thesis project also use Python.

[8] See e.g.: https://smlfamily.github.io/, accessed July 23, 2023

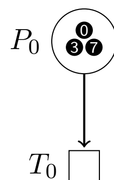


Figure 2.5: A simple CPN, with just one place and a few tokens of type integer.

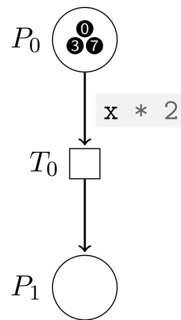


Figure 2.6: A CPN with an arc with an expression.

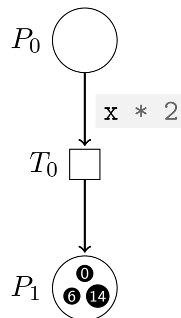


Figure 2.7: The CPN of Figure 2.6 in its final state.

The modelling language is used to define "colors", i.e., types, arc expressions, and transition guards. Arc expressions can perform arbitrary modifications on the tokens that pass through them. Figure 2.6 shows a CPN where the incoming arc to the transition T_0 modifies its token. The expression is written in the Python language and doubles the token value. After T_0 has fired three times, the net is dead. Figure 2.7 depicts the final state.

2.3.3 Transition Guards

Another feature enabled by the inclusion of a modelling language is the transition guard. Each transition may have a guard attached to it, containing an expression that must evaluate to the boolean value `true`. The transition cannot fire if it does not, even if a sufficient number of tokens is available in its input-places.

Figure 2.8 on the facing page shows a variant of the net from Figure 2.5 on the previous page where T_0 has a transition guard. The guard here is displayed as a pseudo-code statement with a light-grey background. Notice also the expression on the arc, binding whichever token is passed along it to the variable name `x`.

T_0 of the net in Figure 2.8 on the facing page can fire twice, since P_0 contains two tokens that satisfy the guard expression of T_0. The third token does not satisfy this condition; thus, the net is dead after two firings.

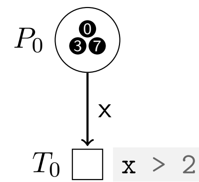


Figure 2.8: A CPN with a transition that has a transition guard.

2.4 Python

This section briefly introduces the Python programming language (cf. Python Software Foundation, 2023), since Python is used as the modelling language for CPNs in this thesis. This section will attempt to cover exactly those features of Python that are required to understand the CPNs presented in future chapters and omit those features that are not necessary for understanding this thesis document.

Python is a high-level, dynamically typed, interpreted programming language. This makes its syntax generally very simple and easy to understand. The Python "hello world" program is as simple as:

```
print("Hello, world!")
```

Variable assignments are simple, too. No keyword or type specifier is required. The following code assigns the value 42 and therefore the type `int` to a variable named `x`:

```
x = 42
```

The `def` keyword defines functions, and blocks are delimited not by braces, such as in C, but by white-space. The following example shows a function definition and subsequent invocation:

```
def do_something(value):
    print(value * 2)

do_something(21)
```

Python has a variety of built-in types, such as the integer type `int`, the floating point number type `float`, and the string type `str`. To define a new type for a Python program, a programmer may use the `class` keyword. For example:

```
class Place:
    def __init__(self, tokens):
        self.tokens = tokens
```

The above defines a class `Place`, and a constructor function (called `__init__` in Python) that accepts the parameter `tokens`. An instance of the type can now be constructed like so:

```
place = Place([1, 2, 3])
```

A more concise way of defining a simple class like the one shown above is to use "dataclasses":

```
from dataclasses import dataclass

@dataclass
class Place:
    tokens: list

place = Place([1, 2, 3])
```

The above is equivalent to the previous declaration of `class Place`.

3 Modelling MPI Communication

This chapter will present the modelling approach central to this thesis. It will show how Colored Petri Nets and their high-level features can be leveraged to model communication in MPI programs.

3.1 Communication Modes

Message Passing Interface Forum, 2021, pp. 48–54 defines four blocking communication modes: *standard*, *buffered*, *synchronous*, and *ready*.

A buffered send, explicitly invoked via `MPI_Bsend`, copies the data to a new buffer and immediately returns control to the process. It can thus return before a corresponding `MPI_Recv` operation has started.

A synchronous send, via `MPI_Ssend`, only returns once the receiving process has started its `MPI_Recv` operation.

A "ready" send operation, via `MPI_Rsend`, may only be called if the matching `MPI_Recv` has already been invoked. If the programmer does not honor this condition, the result is undefined. `MPI_Rsend` uses this extra information for optimization but is otherwise equivalent to a synchronous send.

A standard send, via `MPI_Send`, is among the most popular MPI operations, as shown in Section 2.1.2 on page 11. Standard mode means that the MPI library is allowed to choose whether to perform a buffered or a synchronous send. It is, therefore, challenging to accurately model `MPI_Send` because it may return after buffering the data before any actual communication occurs. Or it may return after the communication is mostly complete.

In addition to these four blocking communication modes, communication can also be non-blocking. Non-blocking sends have the same four possible modes as blocking sends. The important distinction is that a non-blocking send returns immediately with a status object. The object can then be queried for the state of the communication, which then depends on which communication mode was chosen. The primary purpose of non-blocking communication is to avoid deadlocks in complicated communication patterns.

3.2 Modelling Individual MPI Operations

This section will show CPN models of the different send operations described in Section 3.1 on the preceding page. It will then show models of the most common individual MPI operations, as shown in Table 2.3 on page 12. It will thus answer the first leading question posed in Section 1.3 on page 2.

The "v" variants of operations, e.g., `MPI_Allgatherv` for `MPI_Allgather`, are left out, as the communication pattern is equivalent to their regular counterparts. The "v" variants allow each process to send a variable amount of data. Additionally, `MPI_Waitall` is not considered here because it is a special case of `MPI_Wait`. Furthermore, one-sided MPI operations, also known as Remote Memory Access operations, are left out. Section 3.5 on page 39 discusses these in more detail. The "ready" mode variants of `MPI_Send` and `MPI_Isend`, i.e., `MPI_Rsend` and `MPI_Irsend`, are also left out. They are not one of the modes the implementation may choose to use when a "normal" mode send is called, and `MPI_Rsend` and `MPI_Irsend` are only used by 63 and 53 repositories, respectively. Finally, `MPI_Test` is left out because it provides information about a pending communication operation to its calling process but is not actually involved in the communication itself.

This leaves the following operations to consider: the specific blocking sends `MPI_Bsend`, and `MPI_Ssend`; their non-blocking counterparts `MPI_Ibsend`, and `MPI_Issend`; and the popular operations `MPI_Allgather`, `MPI_Allreduce`, `MPI_Alltoall`, `MPI_Barrier`, `MPI_Bcast`, `MPI_Gather`, `MPI_Irecv`, `MPI_Isend`, `MPI_Reduce`, `MPI_Scatter`, `MPI_Send`, `MPI_Waitall`, and `MPI_Wait`.

3.2.1 Point-to-Point Operations

Blocking sends and MPI_Recv

Figure 3.1 on the facing page shows a CPN modelling the operations `MPI_Bsend` and `MPI_Recv`. The places P_0 and P_1 represent the two processes involved in the communication. Place I is an intermediate place, which models the buffering and network traversal of the message. The `MPI_Bsend` transition takes a token from place P_0. It has a transition guard that ensures only orange-colored tokens can be used. The token is deposited unchanged into the intermediate place I. From there, `MPI_Recv` takes it and deposits it into P_1. `MPI_Recv` also takes a token from P_1 when it fires. This models the fact that P_1 must actively call the receive operation; it is not invoked automatically or by P_0 having called a send operation.

`MPI_Bsend` also deposits a new orange token back into P_0. This is used here to illustrate that P_0 becomes ready for its next operation immediately after invoking the buffered send operation. In a more complex net modelling a whole MPI program, perhaps `MPI_Bsend` would generate a different-colored token for P_0, to enable it for whatever operation it would call after `MPI_Bsend`.

P_1 contains a different-colored token than P_0. The teal token from P_1 is consumed by `MPI_Recv`, and immediately deposited back into P_1. This ensures that P_1 can perform

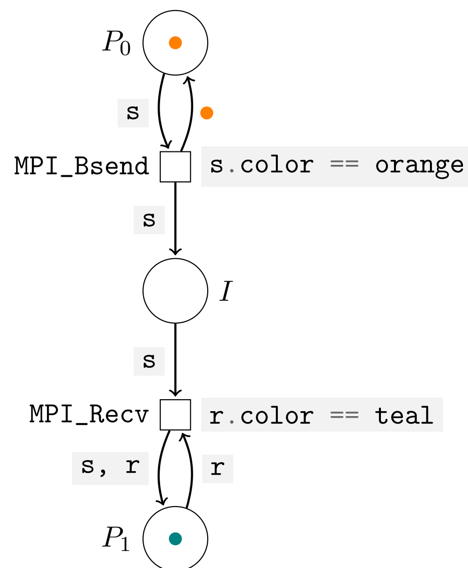


Figure 3.1: `MPI_Bsend` and `MPI_Recv` operations.

another `MPI_Recv` after the first one has concluded. The net can thus fire infinitely often, with P_1 accumulating the orange tokens sent by P_0.

The different token colors, or types, in this CPN are used to symbolize different preconditions for the two operations. The orange token is accepted by `MPI_Bsend` and symbolizes both the data that P_0 wants to send to P_1, and the fact that P_0 is, in fact, ready to perform a send operation. The teal token is consumed immediately by `MPI_Recv`, and represents only the fact that P_1 is ready to perform a receiving operation.

The arcs of the net are annotated. All but one arc in this net pass along their token unmodified. This is symbolized by a lowercase letter (here, `s` or `r`), indicating a variable binding. This is done so that a token is clearly bound to a variable name, which can then be referenced in a guard expression or, later, another arc expression. The arc from `MPI_Recv` to P_0 ignores its input token and instead produces a new orange token to be deposited into P_0.

The different token types will become much more important in larger nets that model entire programs. A single process may call various MPI operations, but they each must be called in the correct order to model the program accurately. A combination of dedicated token types and transition guards can facilitate this.

Figure 3.2 on the next page shows a CPN modelling the operations `MPI_Ssend` and a matching `MPI_Recv`. The key difference between this model and the one in Figure 3.1, is that P_0 may only continue its operation, signified by having a token deposited back into itself, when P_1 begins its `MPI_Recv` operation.

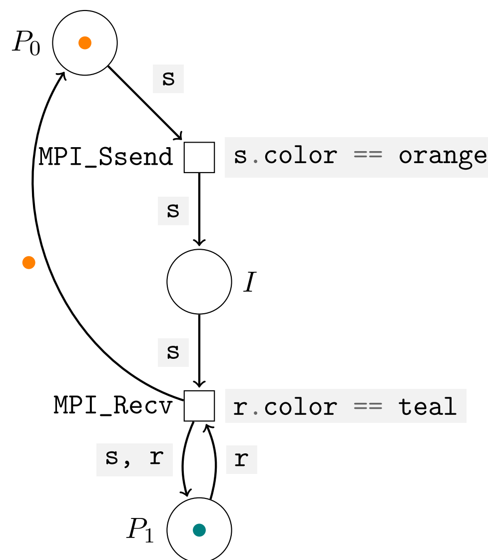


Figure 3.2: MPI_Ssend and MPI_Recv operations.

Non-blocking sends and MPI_Recv

The non-blocking variants of MPI_Send, i.e., MPI_Isend, MPI_Ibsend, and MPI_Issend, can essentially be modelled as MPI_Bsend operations on their own. They return control immediately to the calling process and do not wait on the receiving process, which is what makes them non-blocking. Their communication modes become relevant when paired with an operation that depends on the communication status, such as MPI_Wait. Non-blocking MPI operations return a request object, and its status can be queried by MPI_Test while its completion can be awaited with MPI_Wait.

Figure 3.3 on the next page shows a CPN modelling the operations MPI_Isend, MPI_Recv, and MPI_Wait. Note that whether or not an MPI operation is blocking has no implications for its counterpart called by another process. That is to say, a non-blocking MPI_Isend operation can be paired with a blocking MPI_Recv operation.

The MPI_Ibsend operation is modelled exactly as the MPI_Bsend operation in Figure 3.1 on the preceding page. A new orange token is deposited back into P_0 immediately because the send is non-blocking, and P_0 may continue its operation without waiting for a corresponding receive. The MPI_Wait transition becomes ready immediately upon the MPI_Ibsend invocation because a gray token is deposited into W, an intermediate place used to store the result of MPI_Wait. P_0 may thus call MPI_Wait directly after MPI_Ibsend, because the send operation is buffered. The corresponding receive operation does not need to be started for MPI_Wait to return.

Figure 3.4 on the next page shows a CPN modelling the MPI_Issend operation in conjunction with MPI_Recv and MPI_Wait. The key difference is that MPI_Wait becomes ready only after MPI_Recv has fired, because MPI_Issend uses the synchronous communication mode.

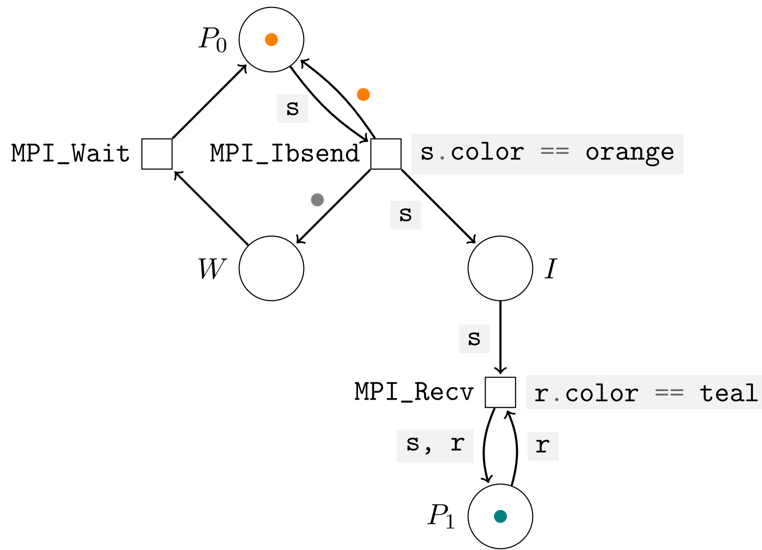


Figure 3.3: MPI_Ibsend, MPI_Recv, and MPI_Wait operations.

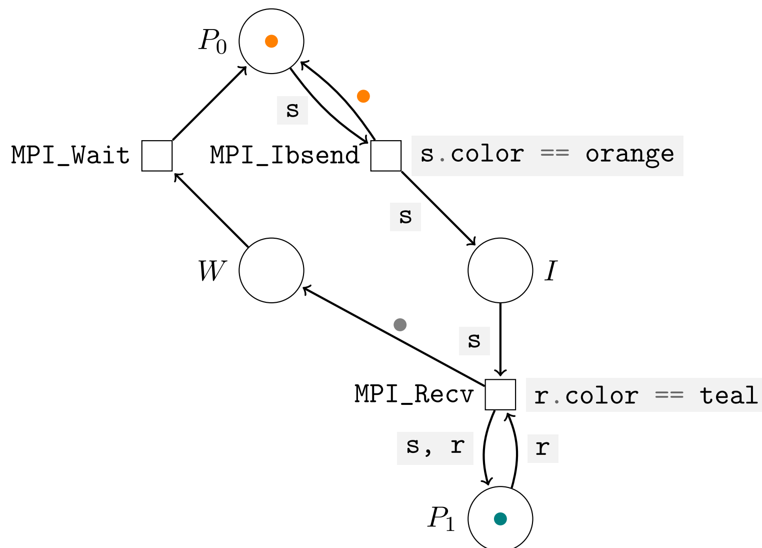


Figure 3.4: MPI_Issend, MPI_Recv, and MPI_Wait operations.

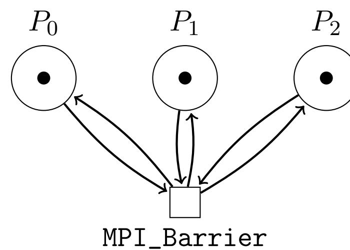


Figure 3.5: Three processes using the MPI_Barrier operation.

MPI_Send and MPI_Isend

The previous sections have shown how to model buffered and synchronous sends, whether they be blocking or non-blocking. As MPI_Send and MPI_Isend can be either buffered or synchronous, at the discretion of the implementation, they can be difficult to model. This thesis project will decide on a case-by-case basis whether to model MPI_Send and MPI_Isend as buffered or synchronous. When translating an MPI program to a CPN, a normal send operation that sends only small buffers or single values will be modelled as buffered sends. All others will be modelled as synchronous. However, for small programs with few calls to either function, one can simply construct models with all permutations of buffered and synchronous sends, and analyse them all. This may be impractical for programs with many different send operations, though.

MPI_Irecv, MPI_Wait, and MPI_Test

Modelling the non-blocking receive operation MPI_Irecv depends on a corresponding MPI_Wait or MPI_Test operation. A call to MPI_Irecv initiates a receive, but the program will need to call MPI_Wait or MPI_Test in order to be sure that the operation has concluded. MPI_Wait will wait for the receive operation to conclude and can thus be modelled exactly like a blocking MPI_Recv. An MPI_Test operation is more difficult to model. If there is a code path that depends on the result of MPI_Test, like an if-statement, which contains further MPI communication operations, two models must be created to model both possible execution paths through this code.

An MPI_Irecv without a corresponding MPI_Wait or MPI_Test is a bug since there is no way for the program to know whether the receive operation has concluded. The modelling approach presented here will not be able to detect it; it must be detected while constructing the CPN, not by evaluating the finished CPN.

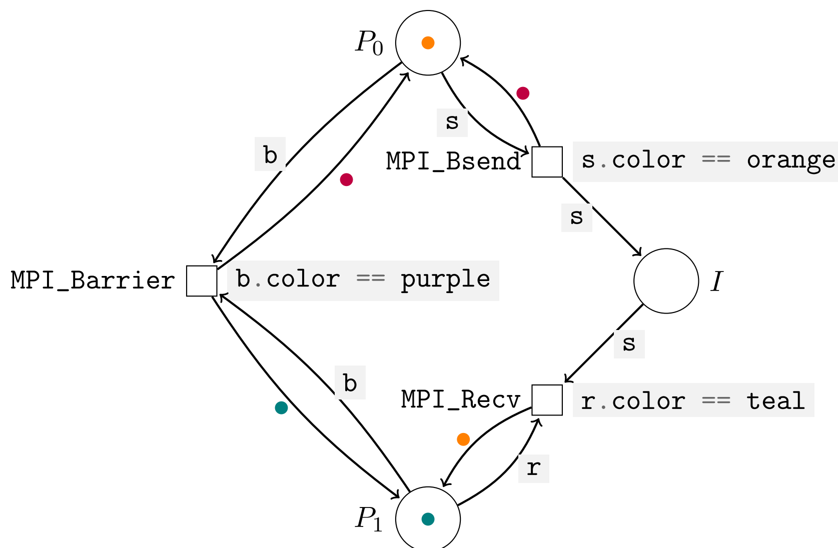


Figure 3.6: Two processes that first perform a send/receive pair of operations and are then synchronized using `MPI_Barrier`.

3.2.2 Collective Operations

MPI_Barrier

Figure 3.5 on the facing page shows an `MPI_Barrier` operation. The individual operation is effortless to model. Each process can only continue when all other processes have entered this same operation. This is easily expressed in a Petri Net by using a transition with an incoming and outgoing arc to and from each process. Figure 3.6 shows a more complex program. Processes P_0 and P_1 perform a send and receive operation in a loop. They are synchronized after the send/receive pair of operations using an `MPI_Barrier`. P_0 uses up its orange send token with every firing of `MPI_Bsend` and gets a purple barrier token. Similarly, P_1 uses up its teal receive token with every firing of `MPI_Recv` and gets a purple barrier token. Once both processes have completed their respective operation, `MPI_Barrier` can fire, resetting the net into its initial state by providing P_0 with another orange send token and P_1 with another teal receive token.

MPI_Gather and MPI_Allgather

Figure 3.7 on the following page shows the `MPI_Gather` operation. One process gathers a value from all processes, including itself. Therefore, processes P_1 and P_2 may immediately continue, while P_0, the gathering process, depends on the operation to complete before it may continue. This makes it necessary to split the single `MPI_Gather` operations into three separate transitions.

Figure 3.8 on the next page shows the `MPI_Allgather` operation, which is much simpler.

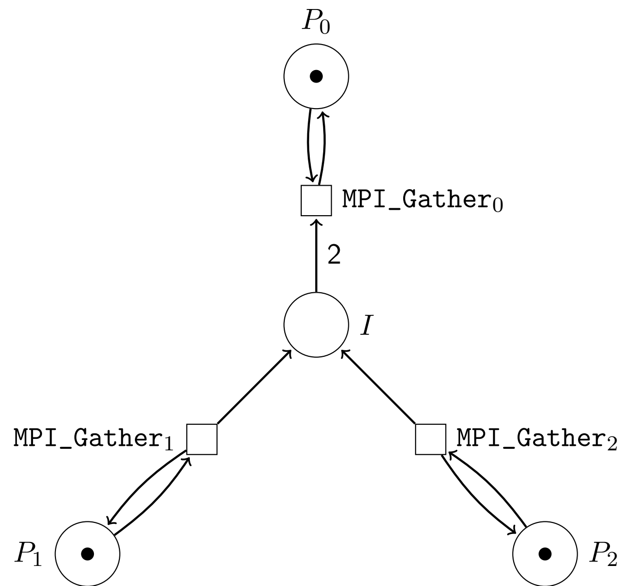


Figure 3.7: Three processes taking part in an MPI_Gather operation. Note that the single operation is modelled as three separate transitions.

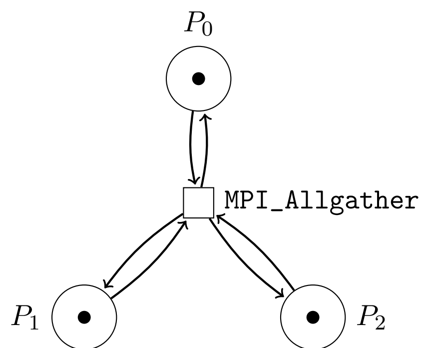


Figure 3.8: Three processes taking part in an MPI_Allgather operation.

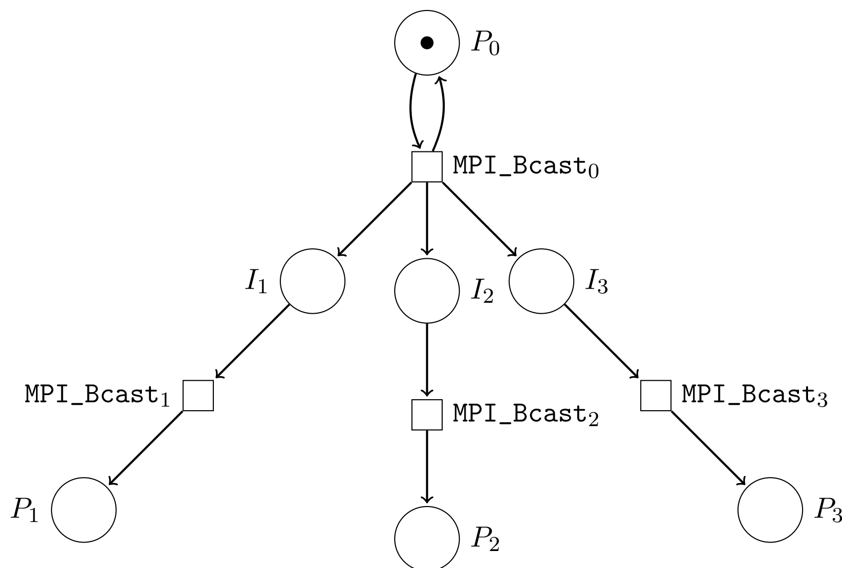


Figure 3.9: Four processes partaking in an MPI_Bcast operation. P_0 is the root process, which broadcasts to all other processes, including itself.

MPI_Allgather has all participating processes send one value and receive all values. As the operation blocks all processes until completion, only a single transition and no intermediate place are required.

MPI_Reduce and MPI_Allreduce

The communication patterns of MPI_Reduce and MPI_Allreduce are equivalent to MPI_Gather and MPI_Allgather, respectively. MPI_Reduce has one process perform a reduction operation on values from all processes, while MPI_Allreduce has each process perform this operation. The operations by themselves can thus be modelled like the ones shown in Figure 3.7 on the facing page and Figure 3.8 on the preceding page.

MPI_Bcast and MPI_Scatter

Figure 3.9 shows a model of an MPI_Bcast operation. MPI_Bcast sends a value from one process, the "root", to all other processes, including the root. The value is immediately available for the root process when it calls MPI_Bcast. This is trivially true since it is the one that sends it. All other processes depend on the MPI_Bcast call made by root, so they are blocked until root enters MPI_Bcast. However, the root does not depend on the other processes. It can call MPI_Bcast and continue, and the remaining processes may make their call at a later time. The CPN in Figure 3.9 models this by using intermediate places for the values sent to the non-root processes and by just having an arc from the initial MPI_Bcast directly back to the root process, here P_0. Thus, processes P_1 through P_3 may call MPI_Bcast

at any time after P_0 has called it. The transitions $\texttt{MPI_Bcast}_0$ through $\texttt{MPI_Bcast}_3$ symbolize the same `MPI_Bcast` call made by the respective process.

`MPI_Scatter` is equivalent to `MPI_Bcast` in its communication. The interesting aspect of `MPI_Scatter` is that a set of values is split into equal pieces, and, as the name implies, scattered from the root process to all processes. This leaves each process with a distinct chunk of the data. But when analyzing just the communication, it can be considered the same operation as `MPI_Bcast`. Therefore, this section does not include a separate CPN for the scatter operation.

3.3 Leveraging Higher-Level CPN Functionality

Some models discussed in Section 3.2 on page 20, while referred to as CPNs, were all also technically Petri Nets. Modelling individual operations is often simple enough not to warrant the use of the high-level features of CPNs. This section will show how some of the high-level features can be used, before Section 3.4 on page 31 will continue with the modelling of entire MPI programs.

Figure 3.10 on the facing page shows a CPN that makes use of multiple high-level CPN features to model a simple combination of `MPI_Bsend` and `MPI_Recv`. The same combination of operations has been modelled previously in Figure 3.1 on page 21. While the previous model used colors to represent different types of tokens, starting with this section, models will use Python code definitions to represent token types. These definitions are required to simulate the models accurately using software.

P_0 thus no longer contains a colored token, as was indicated by the colored dot seen in previous CPN depictions. Instead, its token is displayed next to the place in braces. Its token is an object of type `SendMarker`. Figure 3.11 on page 30 displays the Python definition and the two other classes used in this model. The token contains information about the MPI operation that is to be carried out. It is tagged with the MPI datatype, the source and destination ranks, and a tag. The Python code definitions are not part of the CPN as such; when the model is simulated, the simulation software must load the Python definitions before the simulation.

The transitions in this net now also contain more complex transition guards. While the now more extensive checks do not affect the type state of this model, they will become important in models with more operations. Each transition guard ensures that only the correct type of token may pass through the transition, as seen in previous nets. The guard now also ensures that the right data type, source, destination, and tag properties are set on the token.

The arc from `MPI_Bsend` to P_0 ignores whatever token passes through it and instead creates a new token of type `Finished` before depositing it in P_0. This indicates that P_0 has completed its MPI operation and is thus ready to perform another action. At the same time, it prevents `MPI_Bsend` from firing again, as that would require a token of type `SendMarker`.

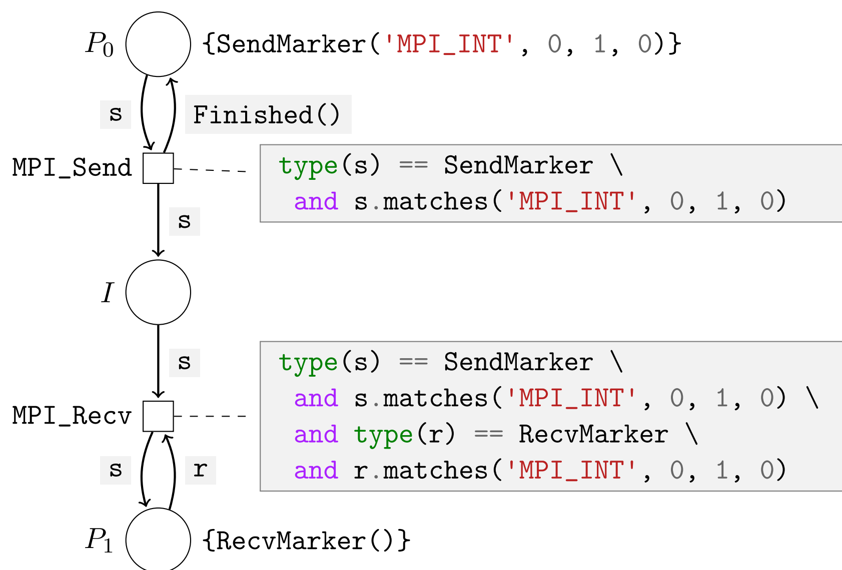


Figure 3.10: A version of Figure 3.1 on page 21 that uses high-level CPN features.

```
from dataclasses import dataclass

class Finished:
    pass

@dataclass
class RecvMarker:
    datatype: str
    src: int
    dest: int
    tag: int

    def matches(self, dtype, src, dest, tag):
        return self.datatype == dtype \
            and self.src == src \
            and self.dest == dest \
            and self.tag == tag

@dataclass
class SendMarker:
    datatype: str
    src: int
    dest: int
    tag: int

    def matches(self, dtype, src, dest, tag):
        return self.datatype == dtype \
            and self.src == src \
            and self.dest == dest \
            and self.tag == tag
```

Figure 3.11: Python class definitions used by the CPN in Figure 3.10 on the previous page.

3.4 Modelling MPI Programs

This section will apply the methods introduced in Section 3.3 on page 28 to model entire MPI programs. It will answer the second leading question posed in Section 1.3 on page 2, how to model complete MPI programs rather than just single operations.

Listing 2.1 on page 6 showed an MPI program that assumes two processes, and sends a value from one process to the other by way of `MPI_Send` and `MPI_Recv`. The CPN shown in Figure 3.12 on the following page models this MPI program more completely than previous models shown in past sections. Besides just the communication operations, transitions for `MPI_Init` and `MPI_Finalize` are now also included. Additionally, the places "Start" and "End" symbolize the start and end of the MPI communication of the program. The start place contains a single Petri Net style token, indicated by the black dot. `MPI_Init` converts this into a `SendMarker`, enabling the `MPI_Send` transition. Once the core operation of the program, namely `MPI_Send` and `MPI_Recv` have concluded, the two participating processes contain tokens of the type `Finished`, which enables the `MPI_Finalize` transition. Once this transition fires, a token is deposited into the end state, indicating the program has terminated.

3.4.1 Modelling Conditional Code

For the purposes of this thesis, conditional code in C comprises if statements (along with else-if and else branches) and loops. Some state of the program must be known to determine which code path is executed whenever such a piece of conditional code is encountered.

Modelling conditional code can be challenging. At times, it can be relatively simple, such as in this example:

```
if (rank == 0)
        MPI_Send(buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

This is trivially an `MPI_Send` operation performed by the process with rank zero. The following is a much more complex example:

```
while (last_value - new_value >= 0.001f) {
        /* perform some manner of MPI communication and other computations */
}
```

Presumably, the two variables are changed in the loop body, and the loop only terminates when `new_value` stops changing significantly. One could attempt to model this program state using CPNs. However, modelling the entire program state has no advantage over just running and evaluating the initial program. The idea of the thesis project is to model just the communication in an effort to identify issues quickly and without running a program for a long time.

The approach taken by this thesis will be to create multiple models of the same program to model the different code paths. For instance, to model a program with MPI operations contained in a loop, three models could be created:

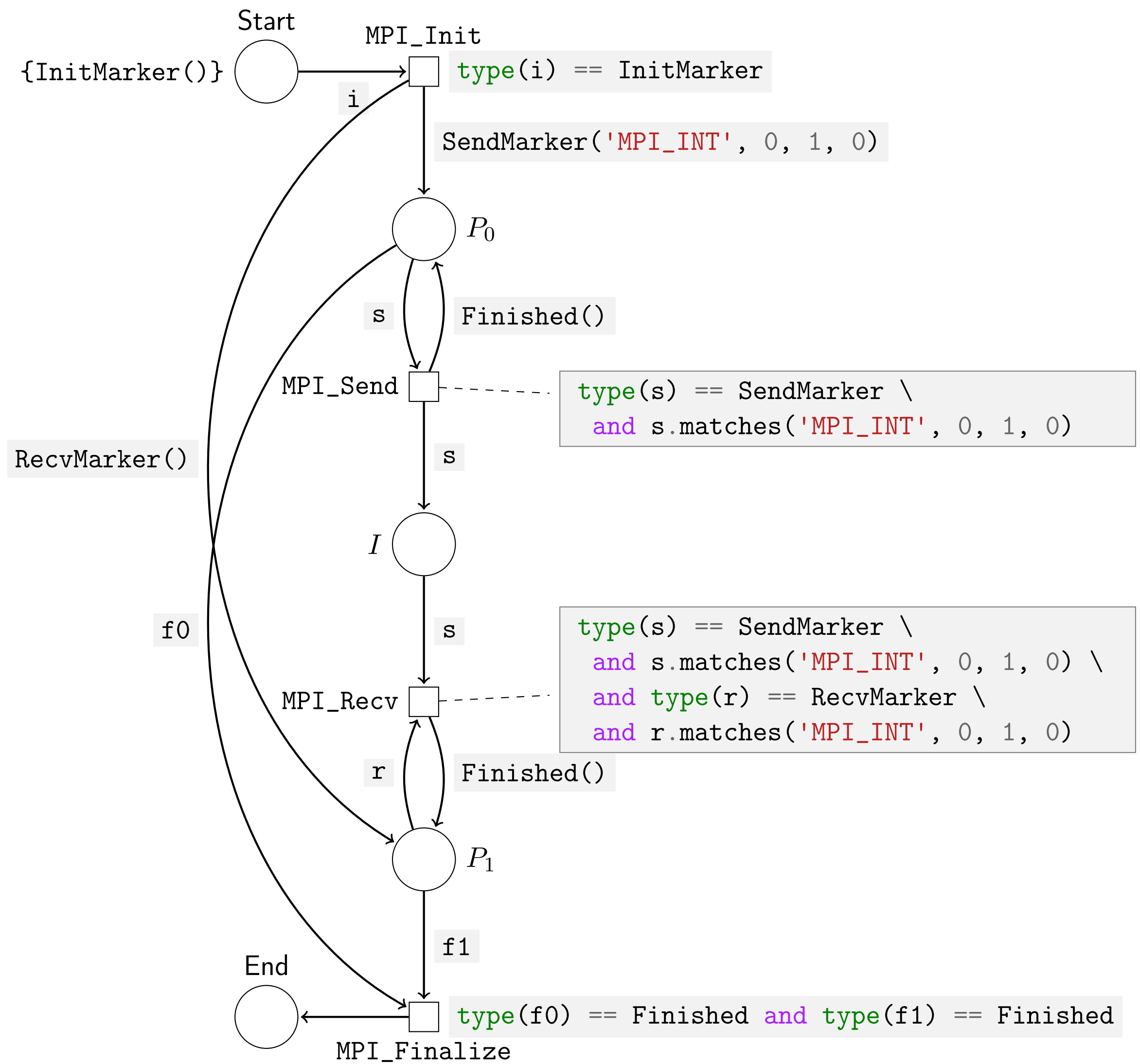


Figure 3.12: A CPN that models the entire MPI program shown in Listing 2.1 on page 6, instead of just individual operations.

- one that skips the loop, assuming the loop condition is never true
- one where the loop is infinite, assuming the loop condition is always true
- one where the loop executes exactly once, to model the effect that the operations inside the loop have on the communication after the loop

For if-statements, static analysis could be used to decide at net-building-time what to do, like in the previous example where it is clear that the statement is used to have a specific rank perform an operation. Otherwise, one net for each possible state would be created and modelled.

To analyze an MPI program, then, multiple nets may be created, and all of them will be analyzed to detect programming errors. This has the major drawback that, for sufficiently complex programs, the number of different nets that need to be created can grow very large. Still, conditional code needs to be handled, and this thesis project will focus on keeping the models themselves relatively simple, at the expense of requiring a large number of them for complex programs. Future work may find ways to minimize the number of nets needed.

3.4.2 Modelling Error Examples

Unmatched Send

Figure 3.13 on the next page shows a CPN modelling the unmatched `MPI_Ssend` operation example of Listing 2.4 on page 9. It is similar to Figure 3.12 on the preceding page but with the `MPI_Recv` transition absent. Therefore, the `MPI_Ssend` operation can never complete, and no `Finished` token can be deposited into P_0. Thus, `MPI_Finalize` is dead, and the model will never reach its intended final state.

Figure 3.14 on page 35 shows a similar CPN to Figure 3.13 on the next page, but using `MPI_Bsend` instead of `MPI_Ssend`. This CPN will reach its intended final state, as the `MPI_Bsend` transition deposits a `Finished` token into P_0, allowing `MPI_Finalize` to fire. This provides an important insight: when analyzing a CPN model of a program, it is insufficient to note whether there are any dead transitions or whether the net has reached its final state with tokens present in the "End" place. It is also necessary to see if any unexpected tokens are left in other places, such as, in this particular case, a `SendMarker` in place I. This "lost" token is the only indication offered by this net that the program contains a logic error.

Unmatched Receive

Figure 3.15 on page 36 shows a CPN modelling the reverse of an unmatched send operation: an unmatched receive operation. Now, P_0 is immediately finished, as it has nothing to do. P_1 is given a `RecvMarker` token by the `MPI_Init` transition, in order to execute its `MPI_Recv`. However, the `MPI_Recv` transition is dead, as it required a token to be present in I, but no token can ever be deposited into I. `MPI_Recv` can thus never deposit a `Finished` token into P_1, and thus `MPI_Finalize` can never fire. A token can never be deposited into "End".

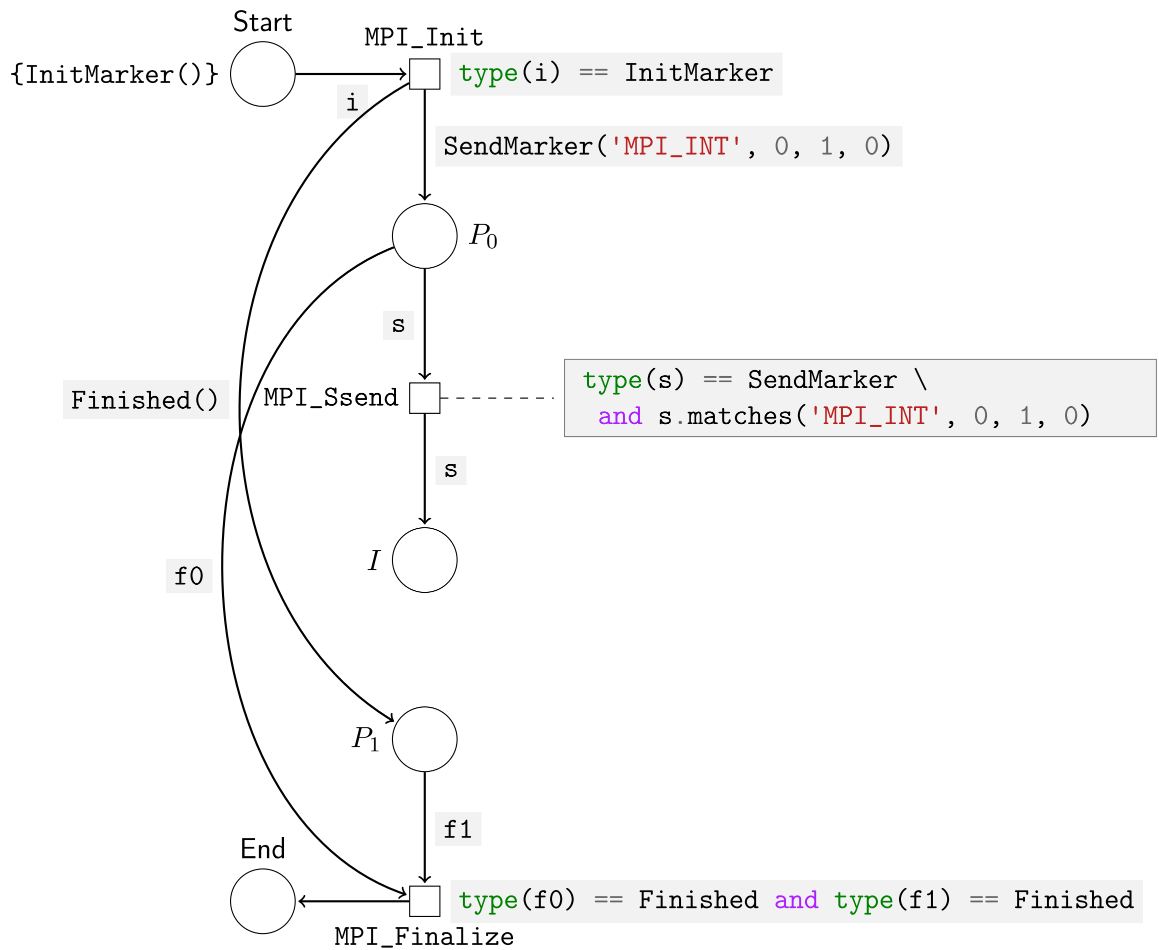


Figure 3.13: A CPN modelling an MPI program with an unmatched MPI_Ssend operation.

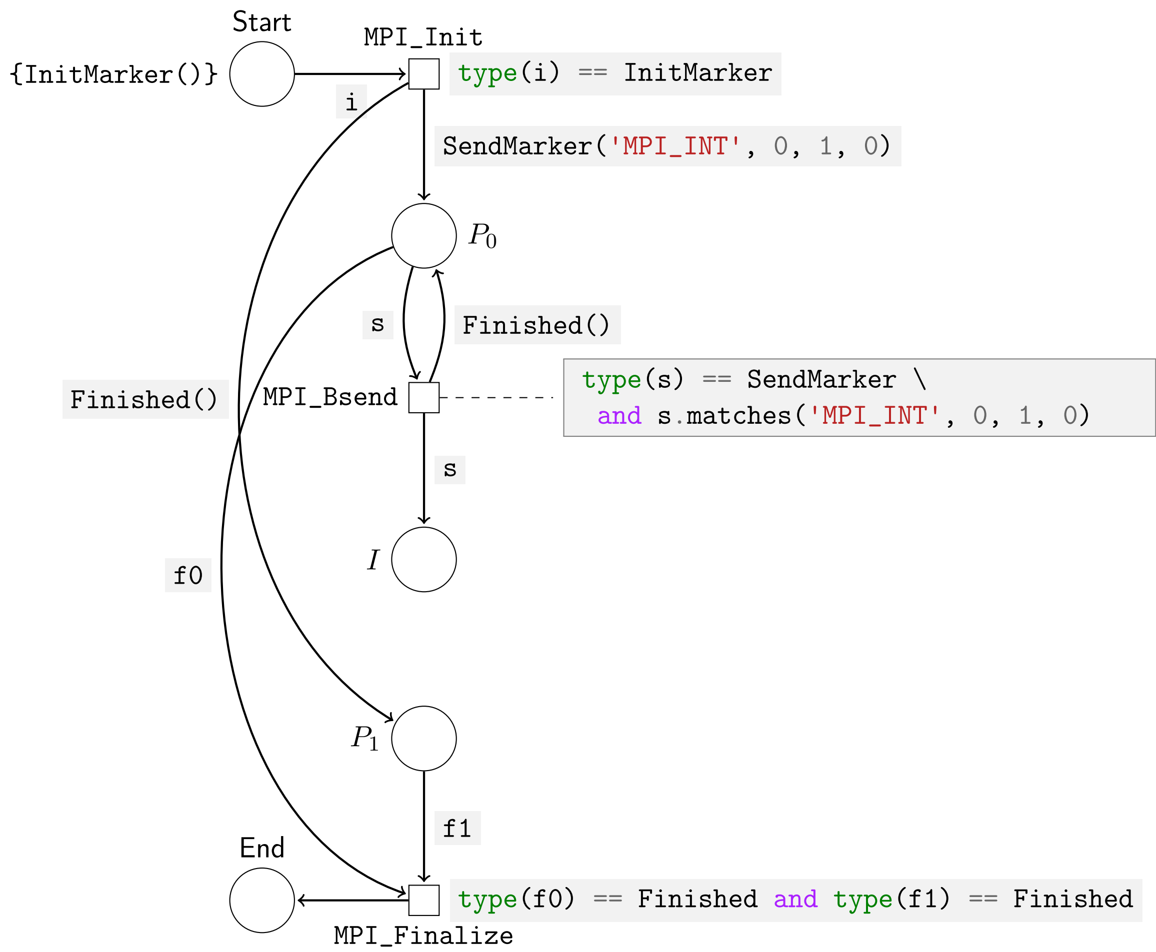


Figure 3.14: A CPN modelling an MPI program with an unmatched MPI_Bsend operation.

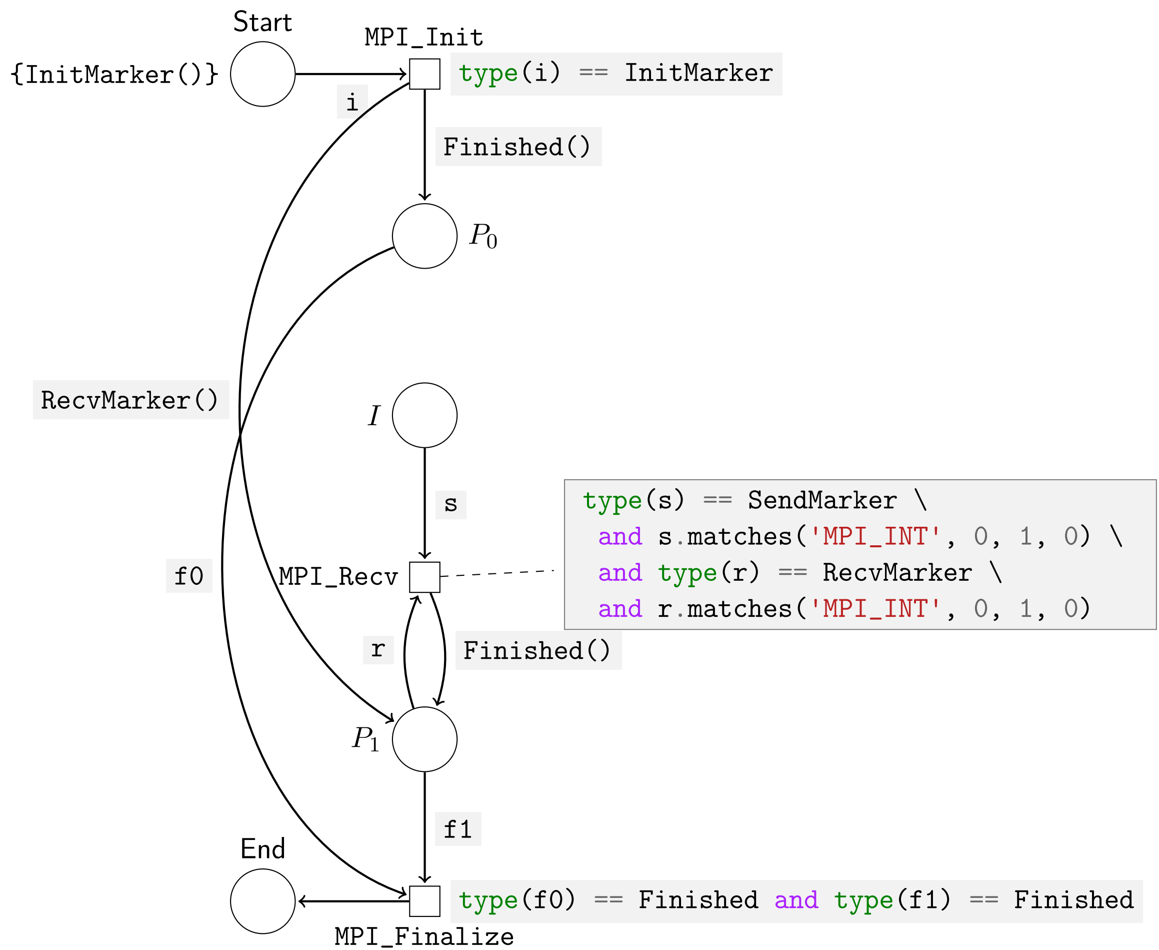


Figure 3.15: A CPN modelling an MPI program with an unmatched `MPI_Recv` operation.

Rank or Tag Mismatch

So long as the transition guards are set correctly, and ensure that any `SendMarker` passing through the transition have the correct source and destination rank, as well as the right tag, any such mismatch will be detected by the corresponding transition not firing when it would be expected to.

Deadlocks

Figure 3.16 on the next page shows a CPN modelling the deadlock example of Listing 2.3 on page 8. It follows the same guidelines as the previously shown models. Due to difficulties laying out the graphic representation of this CPN, most arc annotations, and all transition guards, are not shown. This is purely to aid in understanding the graphic; the actual model follows the same conventions as the previously shown CPNs, including transition guards and arc annotations. In addition to this, all arcs that can never be activated are shown as a thin, dashed line. This illustrates where the net deadlocks and also causes less visual clutter, making it easier to understand the net.

Despite looking quite complicated, with many places and transitions, the net trivially showcases that the program must deadlock. The `MPI_Init` transition can fire, because the place "Start" holds one token of type `InitMarker`. The transition deposits a token of type `RecvMarker` each into P_0 and P_1. Now, the net is dead; no transition can fire. Even though both processes have tokens that would enable them to perform a receive, the receive transitions also require tokens from the intermediate places I_0 and I_1. Those places can only receive tokens from the `MPI_Send` transitions. Those transitions, though, can only fire with a token of type `SendMarker`, but these are only generated once a receive transitions has fired, modelling the fact that the original MPI program first calls `MPI_Recv`, then `MPI_Send`. The CPN thus accurately models a deadlock.

Conclusions

This section has shown how to model simple examples showcasing the programming errors introduced in Section 2.1.1 on page 7. In these simple examples, all errors can be detected by analyzing the state space of a CPN. In most cases, if a transition never fires, an error is indicated. In the case of an unmatched `MPI_Bsend`, a token that would be expected to be consumed in a successful run of the program is left behind in an intermediate place. This indicates the logic error. With these conclusions, MPI programs modelled as CPNs can be analyzed to find these logic errors.

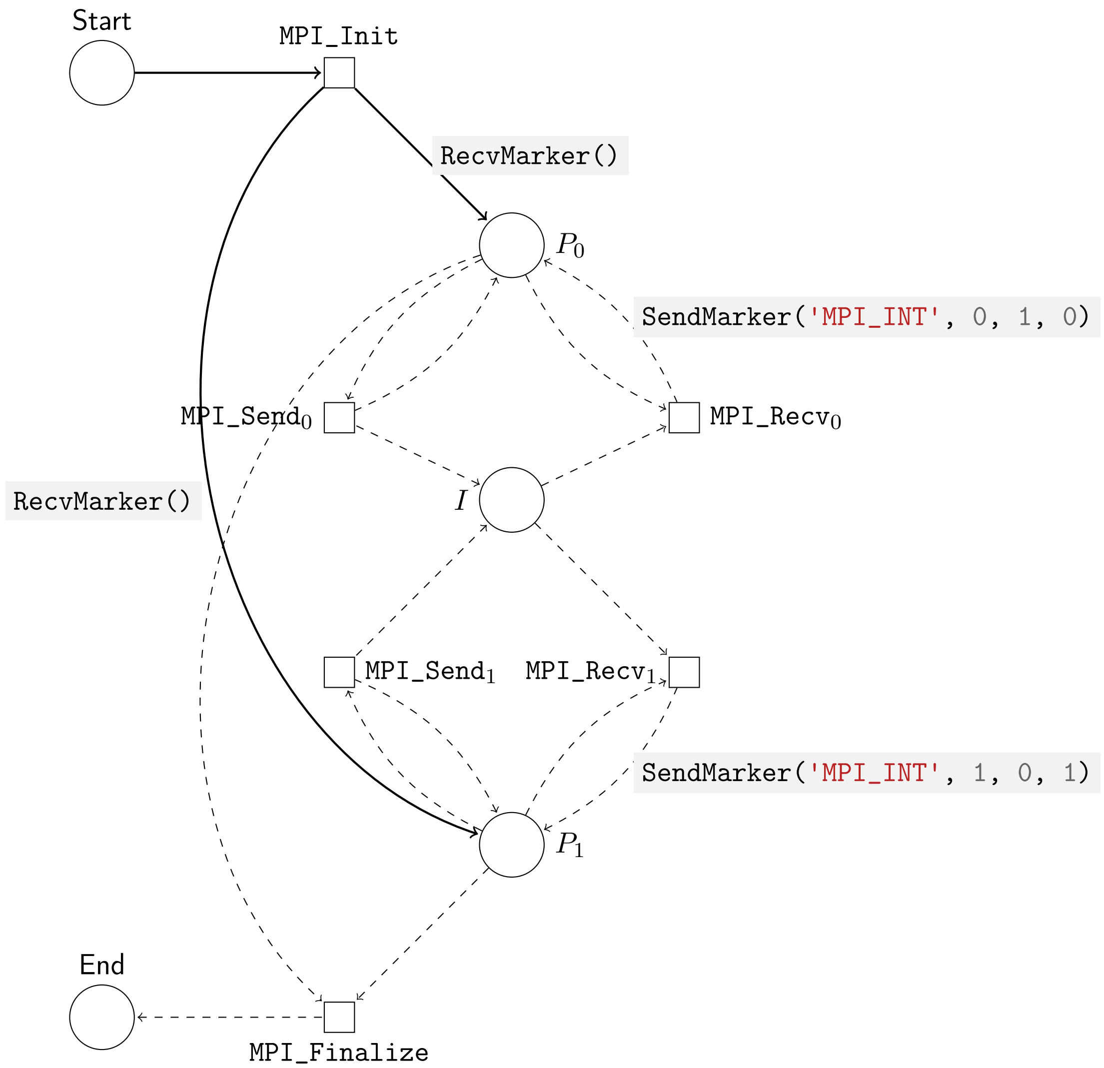


Figure 3.16: A CPN modelling the program displayed in Listing 2.3 on page 8. Most arc annotations and all transition guards are omitted to help make this complicated net easier to understand.

3.5 One-sided MPI Communication

The MPI operations discussed so far have been "two-sided" operations because they necessitate at least one sender and one receiver. One-sided communication uses direct memory access to allow one process to directly read from, or write to, the memory of another process (cf. Message Passing Interface Forum, 2021, Chapter 12). A one-sided MPI communication, e.g., by `MPI_Put`, is roughly equivalent to a pair of `MPI_Send` and `MPI_Recv`. However, only one process "knows" about the communication taking place. The passive process must have previously made the accessed memory available via a window creation function like `MPI_Win_create`. Besides this requirement, which could likely be checked fairly easily using the presented modelling approach, the correct use of one-sided communication mostly depends on ensuring that the accessed memory contains what it is supposed to contain at access time. Since this would necessitate modelling program details besides communication operations, one-sided MPI communication is deemed out of scope for this thesis. Perhaps future work can expand the modelling approach to support one-sided communication and find some types of errors.

4 Intermediate Representation

Modern compilers often use an intermediate representation (IR) in their compilation pipelines. LLVM uses the LLVM Language as defined in "LLVM Language Reference Manual", n.d., while GCC uses two IRs, GENERIC and GIMPLE, depending on which source language is being translated. The C compiler frontend produces GIMPLE directly, while the Fortran frontend produces GENERIC, which is further translated into GIMPLE, as detailed in "Parsing pass (GNU Compiler Collection (GCC) Internals)", n.d.

This chapter will propose an intermediate representation for a potential compilation of MPI programs written in C into Colored Petri Nets. Figure 4.1 displays the basic proposed pipeline. The IR itself, as well as the translation into a CPN, are part of this thesis project. The translation of an MPI program into the IR is not. This chapter defines the IR and Chapter 5 on page 47 discusses the translation of the IR into a CPN.

This chapter will answer the third leading question posed in Section 1.3 on page 2, by specifying exactly what information needs to be extracted from a C-language MPI program to build a CPN model.

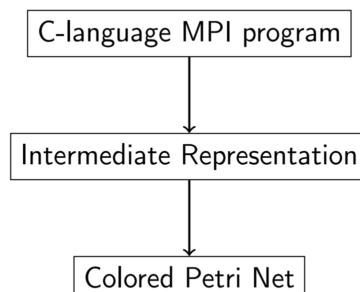


Figure 4.1: A simple visualization of the proposed compilation pipeline.

4.1 Design Considerations

Since the primary purpose of an intermediate representation is to serve as a step in a compilation pipeline, the only hard requirement is that it should be able to represent the source accurately (for the purposes of the final model) and be translatable into the target representation. The IR proposed in this chapter will also be human-readable because this provides several advantages. A human-readable IR is easier to showcase in this document and is easier for a human to generate. Since a translation from MPI program to IR is out of the scope of this thesis, it is critical that translating MPI programs to the IR "by hand" is reasonably straightforward.

A critical choice when generating a CPN representation of an MPI program is the number of processes to be modelled. The design of the IR assumes that this choice is made before the IR is generated. It is thus not generic over the number of processes; this number must be fixed when the IR is generated.

An intermediate representation of an MPI program must contain all information that is then also represented by the CPN. It must thus be possible to generate the following CPN components from the IR:

- places
- transitions, including transition guards
- arcs, including arc expressions

An initial marking is not necessary to include in the IR since the modelling approach presented in Chapter 3 on page 19 simply places an `InitMarker` token into the "Start" place. This marking is trivial and shared among all models and thus does not need to be expressed by the IR.

4.1.1 Places

The modelling approach presented in Chapter 3 on page 19 uses one place for each process. The IR thus only needs to somehow include the number of processes N the model should be generated for, and the places P_i for $0 \leq i < N$ can be created.

4.1.2 Transitions

The transitions used by the modelling approach require more, as well as more complex, information than just how many there are. For point-to-point communication operations, such as `MPI_Bsend`, the initiating and the receiving process must be known for each invocation of

the operation. Consider the C-code snippet below:

```
int dest = (rank + 1) % size;
MPI_Bsend(&buf, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
```

Suppose that buf contains some value to be sent, rank contains the rank of the process, and size the size of the MPI_COMM_WORLD communicator (i.e., the total number of participating processes). This single MPI_Bsend operation would generate size transitions, and each of these transitions would be between distinct places. The IR would need to either contain the computation of dest, and the IR-to-CPN software would then need to interpret it to determine the correct combination of concrete values for rank and dest; or the C-to-IR compiler would need to interpret it while generating the IR, and "unroll" the MPI_Bsend statement into distinct calls with the correct values already set. It would then output the IR with the correct values already set, greatly simplifying the generation of a net from the IR. For example:

```
MPI_Bsend(process=0, to=1);
MPI_Bsend(process=1, to=2);
MPI_Bsend(process=2, to=3);
/* and so on */
```

Transition guards are mostly derived from the modelling approach. Any transition guard depends on the type of operation it applies to. For example, a guard for a send operation checks that any token used to fire its corresponding transition is of the correct type and uses the right values for source and destination. These values are already known when the transition is created. The only additional piece of information used in the guard expression is the tag parameter.

4.1.3 Arcs

Arcs are the most complicated component to generate. Places and transitions can be added to a net essentially in isolation. A name collision must be avoided; otherwise, there is no interaction between two places or two transitions. The modelling approach uses arcs to model program flow. Consider the program modelled by Figure 3.12 on page 32. The MPI_Send transition has two outgoing arcs. One simply moves the input token along to the next place, I. The other arc deposits a Finished token back into the initiating place P_0. This is used to indicate that P_0, the stand-in for one of the processes participating in the MPI program, has completed its operation. In this example, it is obvious that this particular arc must be annotated with the expression `Finished()`. However, a more complex program might have P_0 continue its operation with a different transition. In that case, the arc would need to have a different annotation and perhaps deposit a `SendMarker('MPI_INT', 0, 2, 3)` back into P_0. The complexity increases for an MPI_Ssend operation; the next token would not be deposited back into P_0 by the send transition, but by the MPI_Recv transition, since P_0 must wait for MPI_Ssend to complete before continuing.

This makes generating the arcs for a CPN the most complex aspect. Some arcs are trivial,

but arcs that output special tokens used to model program flow require knowledge of not just the transition and place connected by the arc but also other aspects of the program.

One solution for this might be to use self-referential statements in the IR. A unique identifier accompanies each operation. MPI operations like `MPI_Send` and `MPI_Recv` each contain a reference to whatever operation will follow in the respective process. These identifiers can be used to generate the correct arc annotations.

4.2 Approach

Consider the following pseudo-code that represents the program shown in Listing 2.1 on page 6.

```
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()
```

A unique identifier represents each operation. The name of the operation follows. Its parameters are similar to the original program, but information irrelevant to modelling the communication is left out, and other critical information, like the sending process for `MPI_Send`, is added.

The unique identifier of each operation is written in hexadecimal notation to distinguish it from other numerical values, like process ranks. This representation of the program contains all information required to build the corresponding CPN model.

The tokens deposited by `MPI_Init` can be derived from whichever operation is the first for each participating process. Which process performs which operation can be derived from the `process` parameter. The `next` parameter indicates which operation the process would perform next, and can be used to deduce the token that the current operation needs to deposit back into the place representing the process so that the process may perform its subsequent operation.

Listing 4.1 shows the intermediate representation of Listing 2.3 on page 8. The approach is chosen because all information can be derived either from the arguments to the MPI func-

```
0x0000 MPI_Init()
0x0001 MPI_Recv(process=0, from=1, type='MPI_INT', tag=1, next=0x0002)
0x0002 MPI_Send(process=0, to=1, type='MPI_INT', tag=0, next=0x0005)
0x0003 MPI_Recv(process=1, from=0, type='MPI_INT', tag=0, next=0x0004)
0x0004 MPI_Send(process=1, to=0, type='MPI_INT', tag=1, next=0x0005)
0x0005 MPI_Finalize()
```

Listing 4.1: Intermediate representation of Listing 2.3 on page 8.

tion in the original C-code or from relatively simple surrounding code, such as if-statements that query constant integers, like the rank of the process. This should make it reasonably straightforward to construct a compiler to transform C-code into this IR.

5 Implementation

For this thesis project, a CPN implementation was written in the programming language Rust[1]. The implementation is a library crate[2] called "cpnets". It is published on the Rust package hosting site "crates.io". Its source code is available on GitLab, and will be bundled with digital distributions of this thesis. The version of the crate used in this thesis is `v1.1.0`.

Additionally, a program that can parse the IR defined in Chapter 4 on page 41, generate a CPN, and analyze this CPN, was written. It is named "mpi-cpn-ir-parser" and serves only as a proof-of-concept for the IR and the modelling approach. The version of the crate used in this thesis is `v0.1.0`.

5.1 Accessing the Source Code

The "cpnets" crate can be accessed in one of these ways:

- On crates.io: https://crates.io/crates/cpnets
- On GitLab: https://gitlab.com/tronje/cpn-rs
  - The version used for this thesis project, `v1.1.0`, is available here: https://gitlab.com/tronje/cpn-rs/-/tree/v1.1.0
  - A release package ready for download is available here: https://gitlab.com/tronje/cpn-rs/-/releases/v1.1.0

The "mpi-cpn-ir-parser" crate can be accessed on GitLab at https://gitlab.com/tronje/mpi-cpn-ir-parser:

- The version used for this thesis project, `v0.1.0`, is available here: https://gitlab.com/tronje/mpi-cpn-ir-parser/-/tree/v0.1.0
- A release package ready for download is available here: https://gitlab.com/tronje/mpi-cpn-ir-parser/-/releases/v0.1.0

[1] https://rust-lang.org, accessed July 23, 2023

[2] A crate is essentially a Rust package. See also: https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html, accessed July 23, 2023

```
pub struct Net {
    name: String,
    pub(crate) places: HashSet<Place>,
    pub(crate) transitions: Vec<Transition>,
    rng: StdRng,
}
```

Listing 5.1: Rust definition of the "Net", the principal component of the cpnets crate.

```
pub enum Capacity {
    Limited(usize),
    Infinite,
}

pub struct Place {
    name: String,
    capacity: Capacity,
}
```

Listing 5.2: Rust definition of the "Place" struct, and "Capacity" enum.

5.2 The cpnets Crate

This section will introduce the CPN library written for the thesis project. Even though it was written specifically to model MPI communication, it is a general-purpose CPN implementation, which is not limited to implementing the modelling approach presented by this thesis. It allows building generic CPNs for any purpose.

Listings displaying source code in this section will generally have code comments and derive macros[3] omitted for the sake of brevity.

5.2.1 Net

[3]https://doc.rust-lang.org/reference/procedural-macros.html#derive-macros, accessed July 17, 2023

```
pub struct Transition {
    pub(crate) name: String,
    pub(crate) guard: Option<Guard>,
    pub(crate) in_arcs: Vec<Arc<PlaceToTransition>>,
    pub(crate) out_arcs: Vec<Arc<TransitionToPlace>>,
    meta: Option<Box<dyn Any>>,
}
```

Listing 5.3: Rust definition of the "Transition" struct.

```rust
pub struct PlaceToTransition {
    binding: String,
}

pub struct TransitionToPlace {
    expression: Expression,
}

pub(crate) struct Arc<T> {
    pub(crate) place: String,
    pub(crate) transition: String,
    pub(crate) inner: T,
}
```

Listing 5.4: Rust definition of the "Arc" struct and its two inner types.

The `struct Net` is the principal component of the cpnets crate. It contains all places, transitions, and arcs. Places are simple structs comprised of just a name and a capacity, the latter of which defaults to infinite. Transitions have a name and an optional guard. Their incoming and outgoing arcs are also kept inside the transition struct. This is hidden from the user; the user primarily interacts with the net struct, which exposes methods to add the various components. The net additionally allows running a full simulation or single simulation steps. The user can pass a flag to enable non-determinism. If it is emabled, a random number generator is used to determine which transition to fire if the net is in a state in which multiple transitions can. If non-determinism is not allowed, the program returns an error whenever it encounters such a state. The user may create their own instance of a pseudo-random number generator to achieve repeatable simulation results. This simulation functionality is not used for this thesis project. The more interesting state graph, described in Section 5.2.3 on the following page, is used instead. Listing 5.1 on the preceding page shows the definition of the Net structure. Listing 5.2 on the facing page shows the definition of the Place structure and the Capacity enum it uses. Note that enums in Rust may have optional values attached to them, allowing instances of `Capacity` to signify either infinity or some specific value.

Listing 5.3 on the preceding page shows the definition of the Transition structure, which contains, among other members, two types of arcs. When adding arcs to a net, a difference is made between arcs from places to transitions and arcs that go from transitions to places. "Incoming" arcs (from the point of view of a transition) are given a "binding", i.e., a name that will be assigned to the token that passes through it. The token will then be accessible in expressions in transition guards and outgoing arcs under that name. It follows that multiple incoming arcs may not have the same bindings to avoid name collisions. Listing 5.4 shows the definition of the Arc structure.

"Outgoing" arcs may have arbitrary expressions attached to them. The output of their expression will determine the type of token deposited into their terminal place when their transition

fires.

As already outlined in Section 2.4 on page 17, Python is used as the "modelling language" of the CPN implementation provided by cpnets. That means that arc expressions and transition guards are written in Python. Tokens are also Python objects.

Listing 5.5 on the facing page shows a Rust program that utilizes cpnets to build the traffic light example CPN shown in Figure 2.3 on page 14. In the function `build_net`, first, a new `Net` is created. Methods defined for the `Net` struct are then used to add places, transitions, and arcs. Finally, the function returns the built `Net`.

5.2.2 Marking

A net is independent from its marking. A `struct Marking` can be generated from a net, and it maps places to collections of tokens. This way, a net can be reused, and only new instances of markings need to be created to run multiple simulations. This is handy for the state graph, which uses markings as its nodes, and only keeps one copy of a net.

Listing 5.6 on the facing page shows the definition of the Marking structure and of a private type called "TokenVec". This inner type for the `Marking` is used so that the marking may be hashable and have certain comparison semantics. For the state graph computation to work, two markings must be considered equal when both contain the same tokens in the same places. The *order* of the tokens is irrelevant. So the marking is implemented as having one `TokenVec` per place, and the `TokenVec` struct contains its own specialized implementations of the `Hash`[4] and `PartialEq`[5] traits. The `TokenVec` keeps a vector of the hashes of all its members in a sorted state. This allows efficient comparison, as only the hashes need to be compared when comparing two instances of `TokenVec`.

Listing 5.7 on page 52 shows how a marking is constructed for a net and subsequently used in a simulation step. Marking construction utilizes the "Builder" pattern (cf. Gamma et al., 1995). The net can create an empty marking, already initialized with the required places. The `with` method can then be used to add tokens to places; in the shown example, a token of type string, with the value `"on"`, is added to the place with the name "red".

The simulation step will cause one transition to fire, modifying the marking. The `build_net` function is not shown; one may imagine it to be the same as the one shown in Listing 5.5 on the facing page.

5.2.3 State Graph

The implementation includes a `struct StateGraph`, which can be used to compute the complete state space of a CPN. It is represented as a graph because it may be the case that multiple different states may lead to the same following state, and it is also possible that

[4] https://doc.rust-lang.org/std/hash/trait.Hash.html, accessed July 17, 2023
[5] https://doc.rust-lang.org/std/hash/trait.Hash.html, accessed July 17, 2023

```
use cpnets::prelude::*;
use std::error::Error;

fn build_net() -> Result<Net, Box<dyn Error>> {
    let mut net = Net::new("traffic_light");

    net.add_place(Place::new("green").with_capacity(1))?;
    net.add_place(Place::new("yellow").with_capacity(1))?;
    net.add_place(Place::new("red").with_capacity(1))?;

    net.add_transition(Transition::new("T0"))?;
    net.add_transition(Transition::new("T1"))?;
    net.add_transition(Transition::new("T2"))?;
    net.add_transition(Transition::new("T3"))?;

    net.add_in_arc("green", "T0", "x")?;
    net.add_out_arc("T0", "yellow", Expression::new("x", &["x"]))?;

    net.add_in_arc("yellow", "T1", "x")?;
    net.add_out_arc("T1", "red", Expression::new("x", &["x"]))?;

    net.add_in_arc("red", "T2", "x")?;
    net.add_out_arc("T2", "yellow", Expression::new("x", &["x"]))?;
    net.add_out_arc("T2", "red", Expression::new("x", &["x"]))?;

    net.add_in_arc("yellow", "T3", "x")?;
    net.add_in_arc("red", "T3", "y")?;
    net.add_out_arc("T3", "green", Expression::new("x", &["x"]))?;

    Ok(net)
}
```

Listing 5.5: A Rust program utilizing the cpnets crate to create the traffic light net depicted in Figure 2.3 on page 14.

```
pub(crate) struct TokenVec {
    hashes: Vec<u64>,
    inner: Vec<Token>,
}

pub struct Marking {
    inner: Vec<(Place, TokenVec)>,
}
```

Listing 5.6: Rust definition of the “Marking” struct.

```
use cpnets::prelude::*;
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
    let net = build_net()?;
    let mut marking = net.empty_marking().with("red", [token!("\"on\"")])?;
    net.simulation_step(&mut marking, false)?;
    println!("new marking: {:?}", marking);
    Ok(())
}
```

Listing 5.7: An example of how a marking is constructed for a net.

```
pub struct StateGraph {
    net: Net,
    vertices: HashSet<Rc<Marking>>,
    edges: HashSet<(Rc<Marking>, Rc<Marking>)>,
    terminal_states: HashSet<Rc<Marking>>,
    dead_transitions: HashSet<String>,
}
```

Listing 5.8: Rust definition of the state graph struct.

there are cycles between states. Therefore, a tree is not suitable. A tree would be able to represent the state space by including duplicate states, but if a model did not terminate, the tree would become infinite. Thus, a bidirectional, unweighted graph is used.

The state graph is the primary means of evaluating a model. It can be queried for various properties, which allow various conclusions on the properties of the modelled MPI program. For example, the modelling approach generally expects a model of a sound program to have exactly one final marking, with a `Finished` token in the "End" place. If the state graph does not contain such a marking, there must be an error in the program. It is also possible that this end state is reached but that certain transitions have never fired. The state graph keeps track of the transition that fired to reach a given state and can thus be used to determine whether any transitions can never fire. These so-called dead transitions also indicate an error.

The nodes of the state graph are markings since a marking completely describes the state of a CPN. Markings are kept in a hash set to ensure no duplicates. An initial marking must be provided to construct a state graph. This marking is added to a queue of states to be evaluated. Then, for every state in the queue, all follow-up states are computed by simulating the firing of all transitions that can fire. Any of these follow-up markings that were not already nodes in the graph are added to the graph and to the queue of states to be evaluated so that their following states may also be found. Thus, all possible states of the CPN are computed.

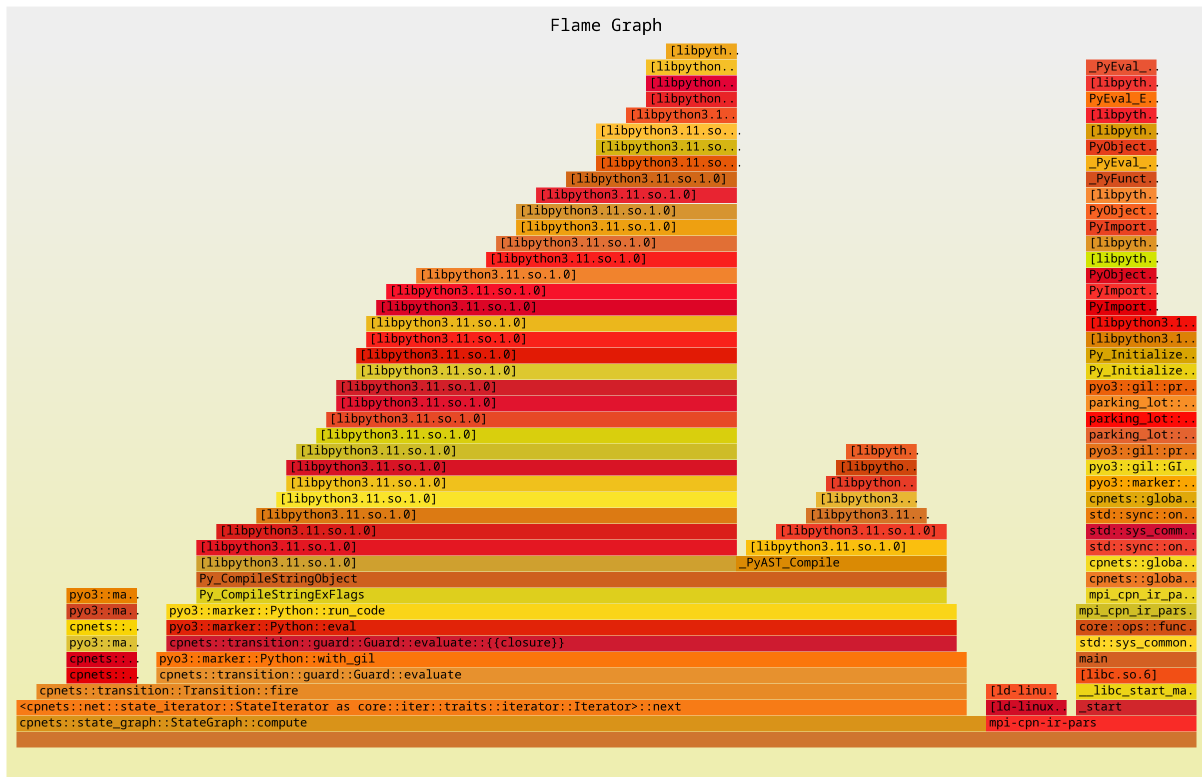


Figure 5.1: "Flamegraph" showing that the code constructing a state graph spends the majority of its time in libpython3.

5.2.4 Python Integration

As the modelling language of cpnets is Python, it must include some way to interpret Python code. To this end, the "pyo3" crate[6] is used. It is essentially a wrapper around libpython3, and so to evaluate Python code, simply calls into the Python shared library. No custom interpreter was written. This has the advantage of saving a large amount of work but also causes performance limitations. The proof-of-concept program which parses the IR, builds the CPN, and computes its state graph, spends about 87% of its CPU cycles in libpython3. In absolute terms, this would be about 75 ms out of its 86 ms runtime. It is speculated that this scales linearly; the larger the state space, the longer the program runs, but the relative time spent in libpython3 should remain about the same. This was determined by using the perf[7] tool.

Figure 5.1 shows a "flamegraph"[8] which displays the relative time spent by the program in different functions. The program run was the proof-of-concept IR evaluator, described in

[6] https://crates.io/crates/pyo3, accessed July 23, 2023
[7] https://perf.wiki.kernel.org/index.php/Main_Page, accessed July 23, 2023
[8] The flamegraph was constructed using https://github.com/flamegraph-rs/flamegraph, accessed July 23, 2023

Section 5.3. The particular IR evaluated produced a state graph of order 226, i.e., with 226 different states. It clearly shows that the program spends the majority of its time inside libpython3. Section 6.4 on page 68 will address this and show how performance can be improved.

5.3 Proof-of-concept IR Evaluator

Combining the modelling approach and the proposal for an intermediate representation, a proof-of-concept program called "mpi-cpn-ir-parser" was written, which parses an IR file, constructs a CPN, computes its state graph, and thereby attempts to find errors. This is not a fully-featured program; it only supports the MPI operations `MPI_Bsend`, `MPI_Ssend`, `MPI_Recv`, and `MPI_Allreduce`, along with the obviously necessary `MPI_Init` and `MPI_Finalize`. Its source code is available on GitLab, at https://gitlab.com/tronje/mpi-cpn-ir-parser.

Parsing of the IR is implemented using "pest"[9]. The pest grammar of the IR is shown in Listing 5.9 on the facing page. During its operation, the program constructs a CPN from the IR. For each operation, a transition, along with a corresponding guard and input/output arcs, is added. To avoid naming collisions transitions are named after the MPI operation they represent, with a short random string added to the end. The number of processes must be passed as an argument to the program, and a place is added to the net for each process. A possible future improvement would be to infer the number of processes automatically from the IR.

After the net has been constructed, the state graph is computed, and information is simply printed to the standard output of the program. Specifically, the program prints:

- the order (number of vertices) and size (number of edges) of the state graph
- whether there are any terminal states
- the marking or markings of terminal states, if any
- any "dead" transitions, i.e., transitions that never fire

5.3.1 Inferring Error Locations in Code

Inferring the exact location of an error in the source code can be accomplished by finding the transition involved in the error and giving the location of the MPI operation it was generated from in the C source code. For a deadlock, this is less straight-forward, but perhaps the last transition that fired before the deadlock occurred can be provided.

The implementation does not support this yet, but the `Transition` struct of the cpnets crate does allow adding arbitrary information to it via its `meta` member, as shown in Listing 5.3 on

[9] https://pest.rs/, accessed July 23, 2023

```
WHITESPACE = _{ " " }

char = { ASCII_ALPHANUMERIC | "_" }
name = @{ char+ }

string = ${ "'" ~ string_contents ~ "'" }
string_contents = @{ (!"'" ~ ANY)* }

int = @{ "-"? ~ ("0" | ( ASCII_NONZERO_DIGIT ~ ASCII_DIGIT* )) }
unique_id = @{ "0x" ~ ASCII_HEX_DIGIT+ }

value = _{ unique_id | string | int }

param = { name ~ "=" ~ value }
params = { param ~ ("," ~ param)* }

op = { name ~ "(" ~ params? ~ ")" }

record = { unique_id ~ op }
file = { SOI ~ (record ~ NEWLINE)* ~ EOI }
```

Listing 5.9: "pest" grammar for the CPN IR defined in Chapter 4 on page 41.

page 48. This could be used to add a code location struct to each transition as the net is generated; such a struct might look like the following example:

```
struct CodeLocation {
    file: String,
    line: u32,
}
```

The IR could be expanded to contain the required information, e.g., like in this example snippet:

```
0x0002 MPI_Send(process=0, to=1, type='MPI_INT', tag=0, file='main.c',
↪   line=42, next=0x0005)
```

The evaluator would then have all the necessary information to provide the user with the MPI operation that caused, or at least that was involved in, a detected error.

6 Evaluation

This chapter will evaluate the modelling approach described in Chapter 3 on page 19 by using the implementation described in Chapter 5 on page 47. To this end, multiple models of various MPI programs will be evaluated, to determine whether they accurately identify any and all mistakes from Section 2.1.1 on page 7. Models should also avoid producing false positives; they must not find errors where there are none.

6.1 Error Examples

This section will show that the models of erroneous programs presented in Section 3.4 on page 31 can indeed be used to detect the contained errors. It will thus answer the fourth and final leading question posed in Section 1.3 on page 2. The Rust code to construct the nets in this section is omitted. It follows the same principles as shown in Chapter 5 on page 47.

6.1.1 Deadlocks

Listing 6.1 shows the intermediate representation of the deadlock example. Running the proof-of-concept evaluator produces the output displayed in Listing 6.2 on the next page. This shows that no transitions besides `MPI_Init` fire, and the only terminal state has no markers in the "End" place. The error is detected.

6.1.2 Unmatched Send

The IR of the elementary example shown in Listing 2.4 on page 9 is displayed in Listing 6.3 on the next page. Provided with this IR, the evaluator produces the output shown in Listing 6.4 on the following page. Since there are two involved processes, but one process performs no

```
0x0000 MPI_Init()
0x0001 MPI_Recv(process=0, from=1, type='MPI_INT', tag=1, next=0x0002)
0x0002 MPI_Ssend(process=0, to=1, type='MPI_INT', tag=0, next=0x0005)
0x0003 MPI_Recv(process=1, from=0, type='MPI_INT', tag=0, next=0x0004)
0x0004 MPI_Ssend(process=1, to=0, type='MPI_INT', tag=1, next=0x0005)
0x0005 MPI_Finalize()
```

Listing 6.1: IR of the deadlock example shown in Listing 2.3 on page 8.

```
Net produced a state graph of order 2 and size 1
Net terminates.
The following transitions never fire: {
    "MPI_Ssend eqd4XX",
    "MPI_Recv HI3y1O",
    "MPI_Recv tWeirX",
    "MPI_Finalize",
    "MPI_Ssend LbL5yN",
}
Markings of terminal states:
        End: [ ], I: [ ], P0: [ RECV(MPI_INT, from=1, to=0, tag=1) ], P1:
        ↪  [ RECV(MPI_INT, from=0, to=1, tag=0) ], Start: [ ]
```

Listing 6.2: Output of the evaluator when given the IR shown in Listing 6.1 on the previous page.

```
0x0000 MPI_Init()
0x0001 MPI_Ssend(process=0, to=1, tag=0, type='MPI_INT', next=0x0002)
0x0002 MPI_Finalize()
```

Listing 6.3: IR of a very simple unmatched MPI_Ssend program.

```
Error: Failed to parse IR!

Caused by:
    0: failed to add operation: Operation { name: "MPI_Init", params:
    ↪  None }
    1: Process 1 has no operations!
```

Listing 6.4: Output produced by the evaluator when given the IR shown in Listing 6.3

```
0x0000 MPI_Init()
0x0001 MPI_Allreduce(process=0, next=0x0003)
0x0002 MPI_Allreduce(process=1, next=0x0004)
0x0003 MPI_Bsend(process=0, to=1, tag=0, type='MPI_INT', next=0x0004)
0x0004 MPI_Finalize()
```

Listing 6.5: IR of an unmatched `MPI_Bsend` example program.

```
Net produced a state graph of order 5 and size 4
Net terminates.
Markings of terminal states:
        End: [ DONE ], I: [ SEND(MPI_INT, from=0, to=1, tag=0) ], P0: [
         ↪  ], P1: [ ], Start: [ ]
```

Listing 6.6: Evaluator output when given the IR from Listing 6.5.

operations besides `MPI_Init` and `MPI_Finalize`, an error is produced immediately before the CPN is even assembled. One could argue that this behavior is not correct. After all, a program that calls only `MPI_Init` and `MPI_Finalize` is free of logic errors, regardless of the number of participating processes. However, this program also contains no actual MPI *communication*. It is, therefore, pointless to check using a tool like the evaluator used here. In this particular case, the produced error message indicates a logic error in the program.

In order to force the evaluator to actually produce and evaluate a CPN, an `MPI_Allreduce` operation will be inserted for the examples evaluated in the following sections.

Unmatched `MPI_Bsend`

The IR for this example is shown in Listing 6.5. The output of the evaluator is shown in Listing 6.6. No dead transitions are detected. This makes sense because a buffered send will conclude immediately, causing process 0 to finish. Process 1 does nothing, and so also finishes. However, the logic error is still detected: a "send" token remains in the intermediate place, which is not consistent with a successful evaluation of the program.

Unmatched `MPI_Ssend`

The IR for this example is almost identical to the one in Listing 6.5. Only the send operation is changed from `MPI_Bsend` to `MPI_Ssend`. Listing 6.7 on the following page shows the output of the evaluator. The error is found: `MPI_Finalize` never fires, and an unexpected token remains in the intermediate place.

6.1.3 Unmatched Receive

```
Net produced a state graph of order 4 and size 3
Net terminates.
The following transitions never fire: {
    "MPI_Finalize",
}
Markings of terminal states:
        End: [ ], I: [ SEND(MPI_INT, from=0, to=1, tag=0) ], P0: [ ], P1:
        ↪  [ DONE ], Start: [ ]
```

Listing 6.7: Evaluator output when given the unmatched `MPI_Ssend` IR.

```
0x0000 MPI_Init()
0x0001 MPI_Allreduce(process=0, next=0x0003)
0x0002 MPI_Allreduce(process=1, next=0x0004)
0x0003 MPI_Recv(process=0, from=1, tag=0, type='MPI_INT', next=0x0004)
0x0004 MPI_Finalize()
```

Listing 6.8: IR of an unmatched `MPI_Recv` example program.

```
Net produced a state graph of order 3 and size 2
Net terminates.
The following transitions never fire: {
    "MPI_Finalize",
    "MPI_Recv 8KjLR5",
}
Markings of terminal states:
        End: [ ], I: [ ], P0: [ RECV(MPI_INT, from=1, to=0, tag=0) ], P1:
        ↪  [ DONE ], Start: [ ]
```

Listing 6.9: Evaluator output when given the IR from Listing 6.8.

```
0x0000 MPI_Init()
0x0001 MPI_Bsend(process=0, to=1, tag=0, type='MPI_INT', next=0x0003)
0x0002 MPI_Recv(process=1, from=0, tag=42, type='MPI_INT', next=0x0003)
0x0003 MPI_Finalize()
```

Listing 6.10: IR of a send operation with a tag mismatch.

```
Net produced a state graph of order 3 and size 2
Net terminates.
The following transitions never fire: {
    "MPI_Recv sNJMFe",
    "MPI_Finalize",
}
Markings of terminal states:
        End: [ ], I: [ SEND(MPI_INT, from=0, to=1, tag=0) ], P0: [ DONE
        ↪ ], P1: [ RECV(MPI_INT, from=0, to=1, tag=42) ], Start: [ ]
```

Listing 6.11: Evaluator output when given the IR from Listing 6.10.

The IR for this example is shown in Listing 6.8 on the preceding page. The evaluator output, shown in Listing 6.9 on the facing page, shows that `MPI_Finalize` and `MPI_Recv` never fire, indicating that the error is found.

6.1.4 Rank, Tag or Type Mismatch

All these examples produce essentially the same output. The marking of the terminal state varies slightly, depending on whether a synchronous or buffered send is used. For the sake of brevity, only a tag mismatch example is shown here.

Listing 6.10 shows the evaluated IR, and Listing 6.11 shows the output of the evaluator. Neither `MPI_Recv` nor `MPI_Finalize` fire, and tokens remain in places other than "End". Clearly, the error is detected.

6.1.5 Conclusions

This section has shown that all errors shown in Section 3.4 on page 31 can be detected. It has not shown that these errors can be detected in all circumstances; it may be the case that, in larger programs with more operations, some of these errors could go undetected. To investigate this further, it would be prudent to evaluate a large number of both correct and erroneous programs. This is left for future work, as it will be much easier once a full translation from C code to CPN is implemented and can be done automatically.

The following is speculation on whether the individual error types can be detected in larger, more complex programs:

- Deadlock: the modelling approach is designed to model the communication of MPI programs accurately; if a communication pattern can cause a deadlock, the state graph should contain a corresponding terminal state, and the deadlock should be found regardless of program size and complexity.
- Unmatched send: again, program complexity should be irrelevant; if a send operation is performed without a matching receive, any terminal state of the model will have a remaining send token left in an intermediate place, indicating the error.
- Unmatched receive: an unmatched receive operation will cause the executing process to stop since the operation can never complete. This, again, is independent of program complexity, and the error should always be detectable.
- Rank, tag, or type mismatch: as this type of error should also cause a transition to never be able to fire, it should be detectable regardless of program size or complexity.

It is conceivable that a program contains a subtle bug caused by two tag mismatches, which cancel each other. So two full send and receive operations are performed, but the intended destinations are swapped. This specific issue would not be detected by the modelling approach.

6.2 Jacobi Method

The Jacobi method is a popular[1] algorithm to be implemented using MPI. This section will consider the implementation for a static number of processes (four) provided by MCS Division, n.d. This implementation can be trivially altered to work with other numbers of processes by simply removing the check of the communicator size. This section will work with this program using 2, 4, 6, 8, and 10 processes. Altering the number of participating processes alters the values computed by the program, but the communication pattern remains the same. This will be sufficient for evaluating the modelling approach and implementation.

As this program contains conditional code, i.e., a loop, multiple models must be produced and evaluated. The MPI operations guarded by if-statements are trivial, though. They are simple computations made with constant values and should be simple to evaluate when compiling the C-code to the IR.

Four different IRs must be generated to fully check the program. The loop is modelled twice. Once as running for infinite iterations, to cover the jump from the last operation of the loop to the first. And once as running exactly one iteration to cover the jump from the last operation of the loop to the first operation after the loop. Additionally, the send operations can either be synchronous or buffered. Since all operations send the same kind of data, it is assumed that they are either all synchronous or all buffered. Therefore, the four IRs are:

- infinite loop, buffered send, shown in Listing 6.12 on the facing page

[1] A rudimentary search on GitHub finds more than one hundred repositories and hundreds of commits, using the query https://github.com/search?q=jacobi%20mpi

```
0x0000 MPI_Init()
0x0001 MPI_Bsend(process=0, to=1, type='MPI_DOUBLE', tag=0, next=0x0004)
0x0002 MPI_Recv(process=1, from=0, type='MPI_DOUBLE', tag=0, next=0x0003)
0x0003 MPI_Bsend(process=1, to=0, type='MPI_DOUBLE', tag=1, next=0x0006)
0x0004 MPI_Recv(process=0, from=1, type='MPI_DOUBLE', tag=1, next=0x0005)
0x0005 MPI_Allreduce(process=0, type='MPI_DOUBLE', next=0x0001)
0x0006 MPI_Allreduce(process=1, type='MPI_DOUBLE', next=0x0002)
```

Listing 6.12: IR of the Jacobi program, using an infinite loop.

```
0x0000 MPI_Init()
0x0001 MPI_Bsend(process=0, to=1, type='MPI_DOUBLE', tag=0, next=0x0004)
0x0002 MPI_Recv(process=1, from=0, type='MPI_DOUBLE', tag=0, next=0x0003)
0x0003 MPI_Bsend(process=1, to=0, type='MPI_DOUBLE', tag=1, next=0x0006)
0x0004 MPI_Recv(process=0, from=1, type='MPI_DOUBLE', tag=1, next=0x0005)
0x0005 MPI_Allreduce(process=0, type='MPI_DOUBLE', next=0x0007)
0x0006 MPI_Allreduce(process=1, type='MPI_DOUBLE', next=0x0007)
0x0007 MPI_Finalize()
```

Listing 6.13: IR of the Jacobi program, using a single-iteration loop.

- infinite loop, synchronous send
- single-iteration loop, buffered send, shown in Listing 6.13
- single-iteration loop, synchronous send

For the sake of brevity, only IRs for two participating processes and using a buffered send operation are included here. Analogous IRs using buffered sends or more processes may be trivially derived.

6.2.1 State Graph Size

Table 6.1 on the following page shows the order (number of vertices) and size (number of edges) of the state graph for each of the four different kinds of IR evaluated in this chapter. The state graphs were computed for two, four, six, eight, and ten processes. The order of the state graph is the number of possible states the CPN can be in, while the size is the number of possible state transitions. The order of the state graph primarily depends on the type of send operation used and the number of processes. Whether or not the main loop of the program is modelled as having one iteration or infinitely many has little effect on the order.

The significant difference in state space size between buffered and synchronous send operations is caused by the fact that with a buffered send, the sending process can continue immediately. This means that whatever state the net is in after the send is performed has more following states than with a synchronous send. This effect compounds when there are

Table 6.1: Order and size of state graphs for the four different CPN variants.

infinite loop, MPI_Ssend			single iteration, MPI_Ssend		
processes	order	size	processes	order	size
2	6	6	2	8	7
4	40	74	4	42	75
6	224	642	6	226	643
8	1152	4482	8	1154	4483
10	5632	27 650	10	5634	27 651

infinite loop, MPI_Bsend			single iteration, MPI_Bsend		
processes	order	size	processes	order	size
2	6	6	2	8	7
4	72	152	4	74	153
6	990	3346	6	992	3347
8	13 776	64 044	8	13 778	64 045
10	191 862	1 135 262	10	191 864	1 135 263

multiple processes, all of which perform send operations. Therefore, the models assuming buffered sends have much larger state spaces.

Figure 6.1 on the next page visualizes the relationship between state graph size and process number and compares these to mathematical functions. For the model using synchronous sends, the state graph grows faster than x^3, but slower than x^4 or the exponential function e^x. However, for the model using buffered sends, the state graph grows faster than all these functions, even e^x.

It follows that the simulation complexity depends on the number of processes simulated, which was to be expected. Trivially, the more processes participate, the more possible states the whole program can be in. However, that the simulation complexity also depends significantly on the type of communication used. Using buffered sends causes the state graph to grow substantially faster than synchronous sends.

6.2.2 Conclusions

This section has shown how the Jacobi implementation can be modelled using the modelling approach and implementation. It has further demonstrated the state space size for different instances of the program. The state space grows significantly faster when using buffered sends, as explained above. This is an interesting takeaway by itself; MPI programmers may wish to use synchronous sends explicitly in order to be more able to reason about the runtime characteristics of their programs.

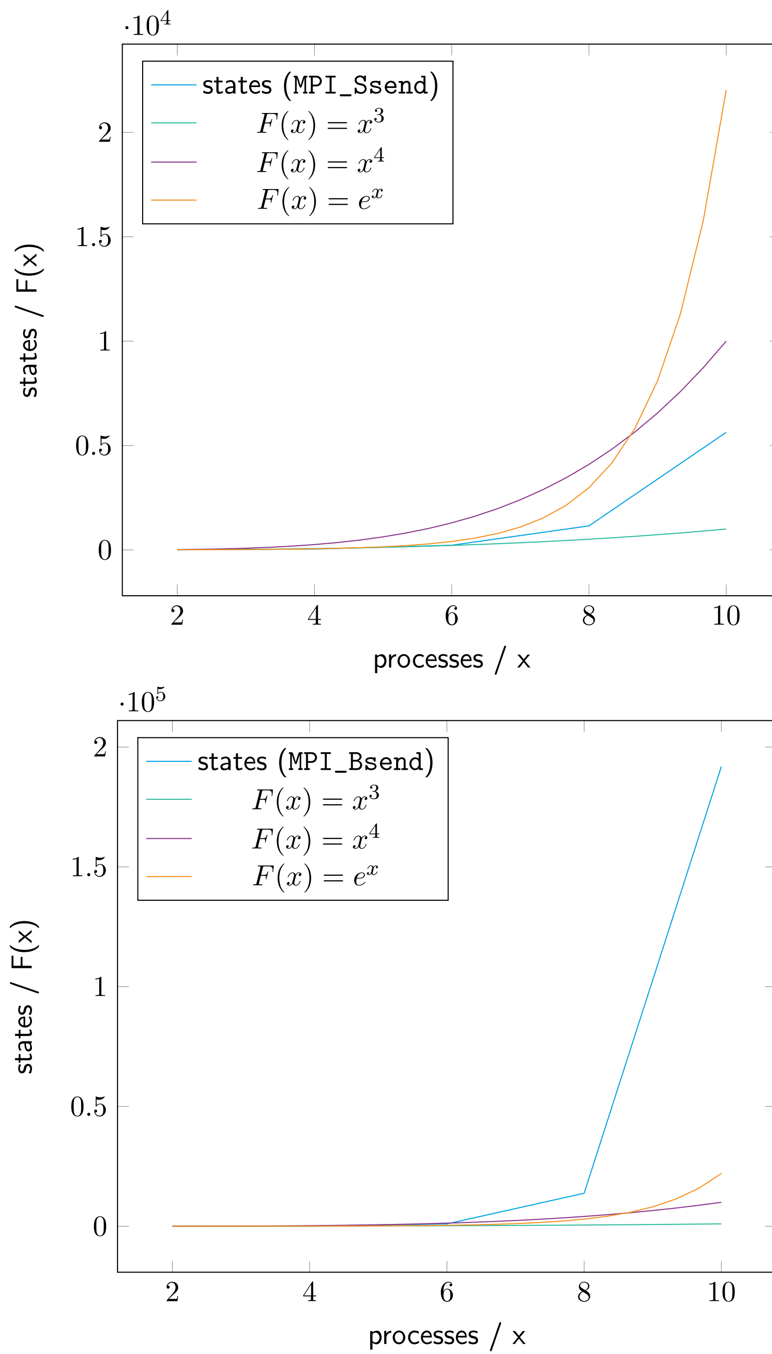


Figure 6.1: State graph order, by number of processes, compared to mathematical functions.

The rapid growth in state space for programs using buffered sends may be problematic. Larger programs will most likely produce even faster-growing state spaces, likely making it impractical to model them for more than ten processes or possibly even fewer.

6.3 Performance

This section will give an overview of the performance of the implementation, showing both execution speed and memory usage. The data was gathered while computing the various state graphs described in the previous section.

All measurements were taken on a system running a Linux operating system using Rust 1.70.0, and an AMD Ryzen 5 3600X CPU[2]. Table 6.2 on the facing page shows the execution time of the evaluator when given the corresponding IR, along with the maximum memory use of the program. The memory use was determined by using the "massif"[3] tool of the Valgrind framework.

6.3.1 Conclusions

Program execution time and memory use scale above linearly with the state graph order. Table 6.3 on the next page shows the total state graph order together with the relative execution time and memory use per state. To speculate, a larger state graph likely also means more cache misses and allocations. Additionally, modern CPUs can run at a "boosted" clock speed for short amounts of time, which means a short-running program can be run at a higher average clock speed, while a long-running program will be run at a slower average clock speed. The CPU used for these evaluations has a base clock speed of 3.8 GHz, and a boost clock speed of 4.4 GHz, an increase of 15%.

Memory use per state grows less quickly and is likely caused by the fact that a state graph of a higher order also has a higher size, i.e., more edges connecting the nodes of the graph. Each additional edge costs a few additional bytes of memory.

[2]https://www.amd.com/en/support/cpu/amd-ryzen-processors/amd-ryzen-5-desktop-processors/amd-ryzen-5-3600x, accessed July 17, 2023

[3]https://valgrind.org/info/tools.html#massif, accessed July 23, 2023

Table 6.2: Evaluation speeds and peak memory usage.

infinite loop, MPI_Ssend				single iteration, MPI_Ssend			
processes	order	time	memory	processes	order	time	memory
2	6	0.07 s	1.1 MiB	2	8	0.07 s	1.1 MiB
4	40	0.07 s	1.2 MiB	4	42	0.07 s	1.2 MiB
6	224	0.14 s	1.8 MiB	6	226	0.14 s	1.8 MiB
8	1152	0.74 s	6.5 MiB	8	1154	0.75 s	6.6 MiB
10	5632	5.42 s	39.6 MiB	10	5634	5.45 s	39.7 MiB

infinite loop, MPI_Bsend				single iteration, MPI_Bsend			
processes	order	time	memory	processes	order	time	memory
2	6	0.07 s	1.1 MiB	2	8	0.07 s	1.1 MiB
4	72	0.08 s	1.2 MiB	4	74	0.08 s	1.2 MiB
6	990	0.73 s	4.5 MiB	6	992	0.76 s	4.6 MiB
8	13 776	17.50 s	79.9 MiB	8	13 778	17.85 s	79.8 MiB
10	191 862	392.38 s	1.6 GiB	10	191 864	397.22 s	1.6 GiB

Table 6.3: Performance per state, relative to total state graph size.

state graph order	time per state	memory per state
1152	642 μs	5916 bytes
5632	962 μs	7373 bytes
13 776	1270 μs	6082 bytes
191 864	2070 μs	8954 bytes

6.4 Revised Implementation

The performance metrics shown in Section 6.3 on page 66 indicate that it is mostly infeasible to simulate even a moderate number of processes for relatively simple programs. A revised implementation was devised to address this. As displayed in Figure 5.1 on page 53, the implementation spends most of its running time inside libpython3. It is thus reasonable to assume that eliminating the Python dependency would enhance performance. This section will present a revised implementation of the "cpnets" crate, which removes the Python dependency, but also sacrifices its general-purpose applicability.

6.4.1 Removing Python as the Modelling Language

Arc expressions and transition guards in CPNs can be arbitrarily complex. However, the modelling approach used in this thesis actually uses relatively simple expressions for both types of expression.

Arc expressions either pass along their input token unmodified or they ignore their input token and generate a new token. Consider, for example, the CPN shown in Figure 3.12 on page 32. All arcs have one of two different kinds of expression: either a simple name binding, which causes the token to be passed on unmodified or a Python constructor that constructs a new token.

Transition guards ensure that the type of token matches the transition, and that the token parameters match those of the transition, as well. For example, for an `MPI_Ssend` transition, the incoming token must be of type `SendMarker`, and its type, source, destination, and tag must match those of the `MPI_Ssend` operation in the source program.

These simple types of expression do not require a complex modelling language and can be realized in pure Rust. While the `struct Expression` of the general purpose cpnets crate was a simple wrapper around a String containing a Python expression, the revised implementation uses a Rust enum. Listing 6.14 on the next page shows the definition of the new expression struct.

Guard expressions depend on the type of transition that they guard, so a specific guard implementation must be written for each MPI operation supported by the software. Listing 6.15 on the facing page shows the new guard definition, which currently only supports the same operations as the proof-of-concept IR parser and evaluator. Like `Expression`, `Guard` is now a Rust enum type rather than a struct.

Finally, the `struct Token` previously wrapped a Python object. For the revised implementation, an enum type is used once again. The token can be one of the supported types of marker. The definition is shown in Listing 6.16 on page 70. As with transition guards, new variants will need to be added to support more MPI operations.

```rust
use crate::token::Token;

pub struct PassThroughExpression {
    pub binding_name: String,
}

pub struct GeneratorExpression {
    pub token: Token,
}

pub enum Expression {
    PassThrough(PassThroughExpression),
    Generator(GeneratorExpression),
}
```

Listing 6.14: Revised `Expression` definition. Code comments and `derive` macros are omitted for brevity.

```rust
use crate::token::Token;

pub struct SendRecvGuard {
    pub src: i32,
    pub dest: i32,
    pub tag: i32,
    pub datatype: String,
}

pub enum Guard {
    Init,
    Send(SendRecvGuard),
    Recv(SendRecvGuard),
    Allreduce,
    Finalize,
}
```

Listing 6.15: Revised `Guard` definition. Code comments and `derive` macros are omitted for brevity.

```
pub struct SendRecvMarker {
    pub src: i32,
    pub dest: i32,
    pub tag: i32,
    pub datatype: String,
}

pub enum Token {
    InitMarker,
    SendMarker(SendRecvMarker),
    RecvMarker(SendRecvMarker),
    AllreduceMarker,
    FinishedMarker,
}
```

Listing 6.16: Revised `struct Token` definition. Code comments and `derive` macros are omitted for brevity.

Table 6.4: Revised implementation speeds and peak memory usage.

infinite loop, `MPI_Ssend`				single iteration, `MPI_Ssend`			
processes	order	time	memory	processes	order	time	memory
2	6	0.05 s	18.0 KiB	2	8	0.05 s	19.3 KiB
4	40	0.05 s	88.6 KiB	4	42	0.05 s	90.6 KiB
6	224	0.05 s	837.2 KiB	6	226	0.05 s	836.2 KiB
8	1152	0.08 s	6.8 MiB	8	1154	0.08 s	6.8 MiB
10	5632	0.33 s	50.3 MiB	10	5634	0.33 s	50.3 MiB

infinite loop, `MPI_Bsend`				single iteration, `MPI_Bsend`			
processes	order	time	memory	processes	order	time	memory
2	6	0.05 s	18.0 KiB	2	8	0.05 s	19.3 KiB
4	72	0.05 s	168.5 KiB	4	74	0.05 s	169.3 KiB
6	990	0.07 s	4.4 MiB	6	992	0.09 s	4.4 MiB
8	13 776	0.88 s	107.5 MiB	8	13 778	0.90 s	107.5 MiB
10	191 862	19.80 s	2.3 GiB	10	191 864	22.67 s	2.3 GiB

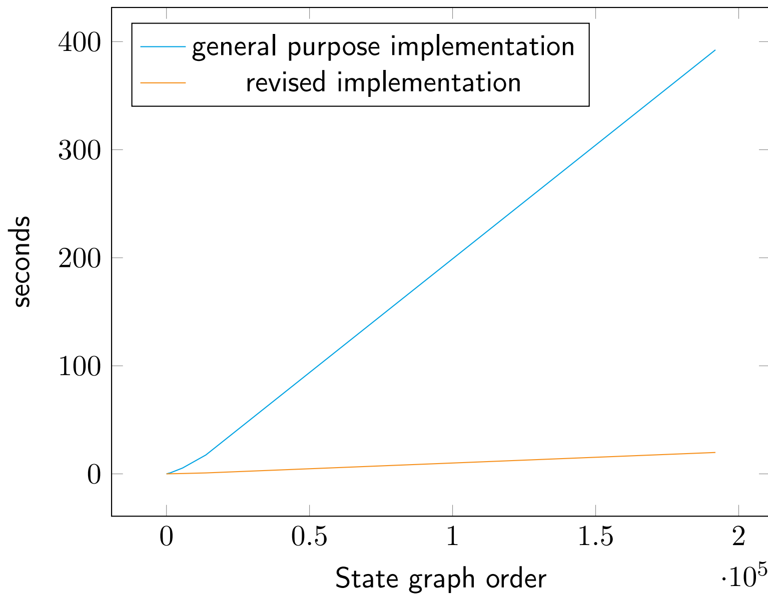


Figure 6.2: Execution times of revised implementation compared to general purpose implementation.

6.4.2 Improved Performance

Table 6.4 on the facing page shows the execution speed and peak memory use of the proof-of-concept evaluator, when it is rebuilt with the special-purpose revised implementation of the cpnets crate. For large state graphs, peak memory use is slightly higher, making the revised implementation worse in that respect. However, peak memory use for smaller state graphs is improved, and execution speed is vastly improved for state graphs of all orders.

Figure 6.2 visualizes the difference in performance between the general purpose implementation and the revised implementation. The revised implementation is faster by a factor of about 20.

It should be noted that since the implementation is not parallelized, it would be possible to evaluate multiple IRs at once, if the CPU that runs the evaluation has a sufficient number of cores. This would allow, for example, evaluating all four IR variants of the ten-process Jacobi program in the same time it would take to evaluate just the slowest one.

6.4.3 Accessing the Source Code

The revised implementation of the cpnets crate is not published on crates.io. It is available on GitLab at https://gitlab.com/tronje/mpi-cpnets. The version used in this thesis, `v0.1.0`, is available here: https://gitlab.com/tronje/mpi-cpnets/-/tree/v0.1.0. A release package ready for download is available here: https://gitlab.com/tronje/mpi-cpnets/-/releases/v0.1.0. An adapted version of the "mpi-cpn-ir-parser" software that uses the revised implementation of

cpnets is available on the "mpi-cpnets" branch of its repository, here: https://gitlab.com/tronje/mpi-cpn-ir-parser/-/tree/mpi-cpnets.

7 Related Work

This chapter will present previous work related to the topic of modelling or evaluating MPI software.

7.1 CIVL

There are several publications by Stephen F. Siegel et al., culminating in the creation of "CIVL: The Concurrency Intermediate Verification Language": Siegel, 2007; Siegel and Avrunin, 2004, 2005; Siegel and Zirkel, 2011; Siegel et al., 2015. CIVL includes a language based on C, CIVL-C, to which multiple different types of parallel C programs may be compiled, including MPI software. The project additionally includes an intermediate representation from which a model is built. This model is then checked using model checking and symbolic execution. CIVL is available via a web application[1] and can be used easily via the web browser.

CIVL can check "the absence of deadlocks, race conditions, assertion violations, illegal pointer dereferences and arithmetic, memory leaks, divisions by zero, and out-of-bound array indexing" (Siegel et al., 2015).

Dwyer et al., 2021 explain the central role played by *scopes* and *processes* in CIVL models. *Static scopes* are analogous to the lexical scope of a program, i.e., they roughly correspond to blocks indicated by curly braces ({ and }) in C. A *dynamic scope* or *dyscope* is an instantiation of a scope and keeps track of the values of variables that live in the scope. A process keeps track of a call stack, which contains references to dyscopes, which tracks the current location of the process within the program.

During an evaluation, CIVL manages a tree of static scopes, a graph of dyscopes, each with a mapping of variables to values, and a set processes which contain references to dyscopes. Dyscopes are instances of static scopes, and there can be more than one dyscope for a given static scope. A state in CIVL is thus much more complicated than a state of a CPN, which is just a mapping of tokens to places.

CIVL relies on the "Symbolic Algebra and Reasoning Library" (Verified Software Laboratory, University of Delaware, 2015) for symbolic execution, and optionally depends on the automated theorem provers CVC3[2], CVC4[3], and Z3[4]. CIVL will work without these theorem

[1] http://civl.cis.udel.edu/app/, accessed November 10, 2022
[2] https://cs.nyu.edu/acsys/cvc3/, accessed July 10, 2023
[3] https://cvc4.github.io/, accessed July 10, 2023
[4] https://github.com/Z3Prover/z3, accessed July 10, 2023

```
$ civl verify examples/concurrency/locksBad.cvl
CIVL v1.21 of 2021-11-04 -- http://vsl.cis.udel.edu/civl
Error: Type cycle detected in $int_iter. Field sequence:

at civlc.cvl:11.0-15.0
struct $int_iter {
^^^^^^^^^^^^^^^^^^^^
        ...
};
^
```

Listing 7.1: CIVL 1.21 invocation and output with an included example file, already in the CIVL language.

provers, "but the results will not be very precise" (Dwyer et al., 2021).

Symbolic execution (cf. King, 1976) is, in simple terms, the idea that instead of evaluating a program for all possible inputs, it is evaluated for classes of inputs. To give a brief example, consider a simple "if" statement, which checks if an integer input is greater than some other constant integer, e.g.:

```
if (input > 1000) {
        do_something();
} else {
        do_something_else();
}
```

Obviously, `input` may contain one of a great number of integer values, but there are only two code paths. So the program is evaluated not based on the value of `input`, but the value of the expression `input > 1000`. It is thus symbolically executed for the class of values that causes the condition of the if statement to be true and the class of values that causes the condition to be false. This is similar to the approach taken by the thesis project, where IR is intended to be generated for each possible path through a program.

7.1.1 Evaluation

Two versions of CIVL have been tested. The latest stable release, CIVL 1.21 Revision 5476, does not seem to work on C programs or even on provided example files already in CIVL language. Attempting to verify a locking example, included with the source code release of CIVL 1.21, produces an error, shown in Listing 7.1

Attempting to verify an MPI program in C produces a similar error, as shown in Listing 7.1. The same error appears to occur for any MPI C program and is not specific to the deadlock example. Indeed, the CIVL tool appears to translate the entire C program, including included header files, into the CIVL language and produces an error at the definition of the `MPI_Status`

```
$ civl verify -input_mpi_nprocs=2 deadlock.c
CIVL v1.21 of 2021-11-04 -- http://vsl.cis.udel.edu/civl
Error: Type cycle detected in MPI_Status. Field sequence:

at mpi.h:208.8-213.0
typedef struct MPI_Status{
        ^^^^^^^^^^^^^^^^^^^^
        ...
} MPI_Status;
^
```

Listing 7.2: CIVL 1.21 invocation and output with the example program shown in Listing 2.3 on page 8.

```
$ civl verify -input_mpi_nprocs=2 unmatched_bsend.c
CIVL v1.20 of 2019-09-27 -- http://vsl.cis.udel.edu/civl
Syntax error: Function MPI_Bsend doesn't have a definition.
at mpi.h:251.4-12 "MPI_Bsend" included from unmatched_bsend.c:1.
```

Listing 7.3: CIVL 1.20 invocation and output with a program that uses `MPI_Bsend`.

struct. This makes it impossible to verify any MPI program, as they all must include the "mpi.h" header file. A different MPI implementation may yield different results. For the evaluations in this section, OpenMPI 4.1.5 was used.

The previous stable CIVL release, CIVL 1.20 Revision 5259, does work and successfully identifies the deadlock in the provided "locksBad.cvl" example. It also successfully identifies the deadlock produced by the program shown in Listing 2.3 on page 8. Additionally, it successfully finds the issue with example programs for rank mismatch, tag mismatch, an unmatched `MPI_Recv`, and an unmatched `MPI_Send`. However, programs containing either `MPI_Ssend` or `MPI_Bsend` produce the error displayed in Listing 7.3. The same error is also produced by the web app, which appears to be running CIVL version 1.18. Siegel et al., 2015 mention that "standard mode blocking point-to-point operations and all collective operations are supported; support for non-blocking operations and more advanced MPI features is in progress", which likely means that the rarely used (cf. Table 2.2 on page 12) explicit send operations are among the "in progress" operations, and not yet implemented.

7.1.2 Performance

Turning again to the implementation of the Jacobi method by MCS Division, n.d., the performance of the CIVL tool can be measured and compared to the proof-of-concept tool developed for this thesis. It should be noted, of course, that the two tools do not do the same thing. CIVL can operate directly on the C program, while the IR evaluator must operate on IR generated by hand. The future implementation of a compiler from C code to the IR

Table 7.1: Performance comparison between CIVL and the IR evaluator written for this thesis.

processes	tool	
	CIVL	IR evaluator
4	112 s	0.20 s
10	701 s	43.13 s

proposed in Chapter 4 on page 41 will likely be very fast; it is reasonable to assume that it would not take longer than compilation into, say, LLVM IR. Its performance impact will, therefore, likely be negligible. Still, it must be pointed out that any comparison made at this time will be incomplete.

The following presents two performance comparisons. One which assumes four participating processes and one which assumes ten. For CIVL, one invocation is timed, whereas the IR evaluator will be invoked four times to cover all possible IR variants.

Table 7.1 displays the results of the performance comparison. Clearly, the IR evaluator is much faster. For four processes, it is faster by a factor of over 500, while for ten processes, the advantage decreases. However, it is still faster by a factor of more than 16.

7.1.3 Conclusion

CIVL can detect more errors than even the modelling approach described in Chapter 3 on page 19 is conceptually able to, such as assertion violations and illegal pointer dereferences. CIVL can also analyze other types of parallel software, such as OpenMP, and combinations, like a program that utilizes both MPI and OpenMP.

The software written for this thesis also exists in a proof-of-concept state, greatly limiting the real-world applications it can evaluate. Still, CIVL appears to be flaky in terms of supported MPI operations, too, as Section 7.1.1 on page 74 shows. And, assuming translation of C code to the proposed IR does not incur significant overhead, the evaluation tool proposed in this thesis will likely be much faster than CIVL for most inputs.

Due to the better performance, a fully featured tool based on the work in this thesis may be used often during the development of an MPI program, and CIVL could then be used as a final test after the programmer deems the program completed.

7.2 MPI-SV

MPI-SV, by Yu et al., 2020, is a symbolic verification tool for checking MPI software. MPI-SV uses Cloud9 (cf. Ciortea et al., 2010), a parallel symbolic execution engine, which is based on Klee (cf. Cadar et al., 2008), and PAT, "a toolkit for flexible and efficient system analysis under fairness" (Sun et al., 2009).

```
$ export KLEE_ROOT=$(pwd)
$ export KLEE_LIBMPI=$(pwd)/libs/Azequialib/lib/
$ export KLEE_UCLIBC_ROOT=$(pwd)/libs/libc/
$ export KLEEROOT=$(pwd)
$ export AZQROOT=$(pwd)/libs/Azequialib
$ export LD_LIBRARY_PATH=$(pwd)/syslibs
$ ./mpisvcc jacobi.c
$ ./mpisv 4 jacobi.bc
F0711 09:41:18.780552 73060 Init.cpp:743] error loading program
↪  jacobi.bc:
*** Check failure stack trace: ***
    @     0x7f4db7a0b9fd  google::LogMessage::Fail()
    @     0x7f4db7a0d89d  google::LogMessage::SendToLog()
    @     0x7f4db7a0b5ec  google::LogMessage::Flush()
    @     0x7f4db7a0e1be  google::LogMessageFatal::~LogMessageFatal()
    @           0x63ebd9  klee::loadByteCode()
    @           0x5bec9e  main
    @     0x7f4db6839850  (unknown)
    @     0x7f4db683990a  (unknown)
    @           0x5be596  (unknown)
./mpisv: line 14: 73060 Aborted                 (core dumped) ./klee
↪  -lib-MPI -threaded-all-globals $*
```

Listing 7.4: Setup, invocation, and output of the current "master" branch state of the pre-compiled distribution of MPI-SV.

MPI-SV uses Cloud9 for symbolic execution to find code paths through the input program. It is possible that violations, such as deadlocks, are detected at this stage. If not, a Communicating Sequential Processes (Hoare, 1978) model is constructed for each path through the project and checked using PAT.

Yu et al., 2020 compare their work to CIVL. The primary differences are that CIVL builds a model for the whole program, while MPI-SV builds models for each path through the program, found via symbolic execution. MPI-SV also supports more MPI operations than CIVL; in particular, it supports non-blocking MPI operations.

7.2.1 Evaluation

MPI-SV is available as a pre-built program at https://github.com/mpi-sv/mpi-sv and its source code is available at https://github.com/mpi-sv/mpi-sv-src (both accessed July 11, 2023).

The project appears to be abandoned. Even though the paper by Yu et al., 2020 was published in 2020, the source code repository has seen only three commits in that year, and the most

recent commit before then is from 2013. As shown in Listing 7.4 on the previous page, precompiled distribution fails to execute. It is possible that building the program from source would fix the issue, but the build fails. One of the primary dependencies, Cloud9[5], also appears abandoned, and fails to build. There is a `depot_tool` manifest, which contains a dependency that is no longer reachable via the specified URL[6] The code repository for Cloud9 contains no build instructions and invocations of `make` or the included `configure` script both fail. It is likely possible to fix the build errors, but this effort was not made due to time constraints. Evaluating MPI-SV is thus deemed out of scope for this thesis.

7.3 MPI-Checker

MPI-Checker, by Droste et al., 2015, is a static analysis tool for checking the correct use of the MPI API. Its checks include checks related to using valid arguments in MPI functions. This means it can check that the specified datatype in, say, an `MPI_Send` invocation matches the type of the specified buffer. These checks are not part of the communication pattern of an MPI program and thus, explicitly, not covered by this thesis. MPI-Checker further claims to be able to detect deadlocks in blocking communication and issues with non-blocking MPI functions. The latter include “double request usage of non-blocking calls without intermediate wait”, “waiting for a request that was never used by a non-blocking call”, and “nonblocking call without matching wait”.

Following the instructions[7] in the source code repository with the error examples outlined in Section 2.1.1 on page 7, though, produces no error messages, as shown in Listing 7.5 on the next page.

MPI-Checker could be a promising project to integrate a completed IR evaluator with. Its static analysis capabilities may allow it to generate the IR specified in Chapter 4 on page 41, and then the checks supported by the modelling approach could be integrated into MPI-Checker.

[5]https://github.com/dslab-epfl/cloud9, accessed July 11, 2023

[6]The manifest is available at https://github.com/dslab-epfl/cloud9-depot/blob/master/DEPS, and the “gyp” dependency is not available at the specified URL.

[7]https://github.com/0ax1/MPI-Checker/blob/main/examples/Readme.md, accessed July 11, 2023

```
$ intercept-build make
$ # output of `make`` omitted for brevity
$ analyze-build --enable-checker optin.mpi.MPI-Checker
analyze-build: Removing directory
↪  '/tmp/tronje/scan-build-2023-07-11-16-15-43-688305-k8bu1lcw' because
↪  it contains no report.
$ run-clang-tidy -p=`pwd` -clang-tidy-binary=`which clang-tidy`
↪  -checks='-*,mpi-type-mismatch,mpi-buffer-deref'
Enabled checks:
    mpi-buffer-deref
    mpi-type-mismatch

/usr/bin/clang-tidy -checks=-*,mpi-type-mismatch,mpi-buffer-deref
↪  -p=/home/tronje/uni/msc-thesis/mpi-errors
↪  /home/tronje/uni/msc-thesis/mpi-errors/unmatched_bsend.c
/usr/bin/clang-tidy -checks=-*,mpi-type-mismatch,mpi-buffer-deref
↪  -p=/home/tronje/uni/msc-thesis/mpi-errors
↪  /home/tronje/uni/msc-thesis/mpi-errors/unmatched_send.c
/usr/bin/clang-tidy -checks=-*,mpi-type-mismatch,mpi-buffer-deref
↪  -p=/home/tronje/uni/msc-thesis/mpi-errors
↪  /home/tronje/uni/msc-thesis/mpi-errors/deadlock.c
/usr/bin/clang-tidy -checks=-*,mpi-type-mismatch,mpi-buffer-deref
↪  -p=/home/tronje/uni/msc-thesis/mpi-errors
↪  /home/tronje/uni/msc-thesis/mpi-errors/unmatched_recv.c
/usr/bin/clang-tidy -checks=-*,mpi-type-mismatch,mpi-buffer-deref
↪  -p=/home/tronje/uni/msc-thesis/mpi-errors
↪  /home/tronje/uni/msc-thesis/mpi-errors/rank_mismatch.c
/usr/bin/clang-tidy -checks=-*,mpi-type-mismatch,mpi-buffer-deref
↪  -p=/home/tronje/uni/msc-thesis/mpi-errors
↪  /home/tronje/uni/msc-thesis/mpi-errors/tag_mismatch.c
/usr/bin/clang-tidy -checks=-*,mpi-type-mismatch,mpi-buffer-deref
↪  -p=/home/tronje/uni/msc-thesis/mpi-errors
↪  /home/tronje/uni/msc-thesis/mpi-errors/unmatched_ssend.c
```

Listing 7.5: Running MPI-Checker on the repository of MPI error examples described in Section 2.1.1 on page 7. Note that no errors are reported.

7.4 In-Situ Partial Order

Vo et al., 2009 use a method to verify MPI software without building a model at all. Their approach, In-Situ Partial Order (ISP), involves running the MPI program within a scheduler application. One of its advantages is that no model needs to be generated; ISP analyses MPI code directly and can find deadlocks or verify their absence.

The website noted in the paper, http://www.cs.utah.edu/formal_verification/ISP (accessed July 11, 2023) returns an HTTP 404 error. A search on GitHub returns a source code repository, https://github.com/cogumbreiro/isp (accessed July 11, 2023), but the code fails to compile. Listing 7.6 on the next page shows the produced error message. Note the statement `MPI_Type_lb was removed in MPI-3.0` which indicates that ISP is not compatible with the MPI 3.0 standard, much less the latest MPI standard (cf. Message Passing Interface Forum, 2021). Neither Vo et al., 2009 nor the source code repository specifies instructions on building or running ISP, either. The project is therefore considered abandoned and will not be evaluated here.

```
In file included from isp.h:24,
                 from Profiler.c:21:
Profiler.c: In function ‘void copyBufferNonBlockingSend(void*, void**,
↪  int, MPI_Datatype, int*)’:
Profiler.c:832:5: error: static assertion failed: MPI_Type_lb was removed
↪  in MPI-3.0.  Use MPI_Type_get_extent instead.
  832 |     MPI_Type_lb(datatype, (MPI_Aint*)&lowerbound);
      |     ^~~~~~~~~~~
Profiler.c:838:5: error: static assertion failed: MPI_Type_extent was
↪  removed in MPI-3.0.  Use MPI_Type_get_extent instead.
  838 |     MPI_Type_extent(datatype, (MPI_Aint*)&extent);
      |     ^~~~~~~~~~~~~~~
```

Listing 7.6: Excerpt of the compiler output when attempting to build the In-Situ Partial Order (ISP) software. The compilation was triggered by first running the included `configure` script, followed by an invocation of `make`.

7.5 Kaira

Böhm and Běhálek, 2012 have come up with what is essentially the opposite of this thesis project. Their software "Kaira" can be used to create CPNs and to convert them into MPI programs.

Unfortunately, the official Kaira website[8] fails to load, and the source code repository[9] has seen only one single-line change since 2016. The tool fails to build on a modern Linux installation.

7.6 SNAKES

SNAKES, by Pommereau, 2015, is a Python library that implements CPNs, also using Python as the modelling language of the nets. SNAKES differs from cpnets in that it does not keep its marking independent from its net object. Each net contains places, and places contain their tokens. Markings can be extracted from this configuration of the net. To build the state graph, the net is put into a given state, and then a transition is fired. This puts the net into its follow-up state, at which point a marking can be extracted. The net will then need to be reset into its previous state in order to compute any other follow-up states that may exist.

Table 7.2 on the following page shows a brief performance comparison. The table shows the time it takes each library to compute the state graphs of the CPN for the Jacobi model, assuming a single-iteration loop and a synchronous send operation. SNAKES is slower than cpnets for small state graphs and slightly faster for larger graphs. The revised implementation

[8] http://verif.cs.vsb.cz/kaira/, accessed July 5, 2023
[9] https://github.com/spirali/kaira, accessed July 5, 2023

Table 7.2: Comparison of state graph build time between SNAKES and cpnets.

state graph order	SNAKES	cpnets
8	838 μs	289 μs
42	9.07 ms	5.96 ms
226	88.08 ms	72.51 ms
1154	668.53 ms	675.13 ms
5632	4.58 s	5.28 s

is much faster (cf. Table 6.4 on page 70), though this is not quite a fair comparison since it is no longer a general-purpose CPN implementation, like SNAKES is.

A related project, "neco" (Fronc, n.d.), can compile SNAKES nets into Cython[10] code. This promises better performance when evaluating the net. However, the project appears to be abandoned. The last commit was made in 2013, and the source code is written in Python 2, which has been "sunset" on January 1, 2020[11].

[10] https://cython.org/, accessed July 17, 2023
[11] https://www.python.org/doc/sunset-python-2/, accessed July 16, 2023

8 Outlook

This thesis has proposed a way of modelling MPI communication as CPNs and implemented a proof-of-concept software of this modelling approach. This chapter will summarize the shortcomings and future work of the theoretical modelling approach and the practical software.

8.1 Modelling

8.1.1 Missing MPI Operations

Chapter 3 on page 19 has proposed ways of modelling the most popular MPI operations in Colored Petri Nets. However, that still leaves a large number of operations unexplored. The mere size of the MPI standard made proposing a model for all these impractical. Additionally, not all proposed models of individual operations were integrated into a model of a full program. Finally, one-sided operations were not explored, as briefly discussed in Section 3.5 on page 39. All of this is left for future work.

8.1.2 Symbolic Execution

The modelling approach focuses on modelling MPI communication. Dealing with language-level program constructs like if-statements and loops is solved by constructing a model for each path through the code. Applying the approach to complex problems may prove impractical due to the likely vast state space. Perhaps future work can extend the CPN models to model control flow in addition to MPI communication.

The currently proposed approach is similar to symbolic execution, a concept used by related work as outlined in Chapter 7 on page 73. Perhaps a better way of dealing with branching code paths would be to pursue this approach, for example, by using a symbolic execution engine and then constructing a CPN for every possible path through the program. This is about equivalent to the proposal in Section 3.4.1 on page 31 and to the approach taken by Yu et al., 2020.

8.2 Implementation

8.2.1 Missing MPI Operations

The implemented IR parser and evaluator software includes support for only a minimal subset of MPI operations, namely `MPI_Bsend`, `MPI_Ssend`, `MPI_Recv`, `MPI_Allgather`, and the always necessary `MPI_Init` and `MPI_Finalize`. More operations must be supported to make this a practical tool for analyzing MPI code. It is left to future work to extend the modelling approach accordingly and add the corresponding functionality to the implementation.

8.2.2 Compilation

The implementation outlined in Chapter 5 on page 47 includes a parser of the IR proposed in Chapter 4 on page 41, which can construct a CPN. However, the link between a C-language MPI program and the IR is missing. A fully-featured implementation would need to have the ability to compile such a program to the IR. This will pose further challenges, such as determining which operation is executed by which process and which MPI operation will be performed next by the given process. It is possible that symbolic execution and static analysis can be used to solve these issues, as is done by similar projects as outlined in Chapter 7 on page 73.

8.2.3 Performance Optimizations

Early Termination

The current implementation constructs the entire state graph of a model, which is wasteful. Only terminal states, that is, states in which the net is dead and for which no following state exists, are evaluated for error detection. A terminal state indicates an error if, for example, a token exists in some place other than the "End" place, or the "End" place does not contain any "Finished" tokens. Therefore, if it finds such an error-indicating state, the computation of the state graph can terminate early. This may occur long before all possible states have been computed, thus saving time by not visiting uninteresting states.

Parallelization

Both the cpnets library and the IR evaluator are sequential programs. No parallelism is used when parsing the IR, generating the CPN, or computing and evaluating the state space. Parallelizing the state space computation may improve performance, especially for large state spaces. However, threads running in parallel need to synchronize in order to know about already visited states, so that no redundant work is done. This will likely lead to extensive use of locking, e.g., via mutexes, incurring significant overhead. Evaluating how well the software can be parallelized is left for future work.

8.2.4 Large-scale Evaluation

The software written for this thesis project was evaluated only on small, hand-crafted error examples and a relatively simple implementation of the Jacobi method. To truly assess the modelling approach and its implementation, an automated evaluation of a large number of open-source MPI software can be performed. This, of course, requires the ability to compile a C-program to a CPN, which does not exist yet. In the current state, it is impractical to evaluate many projects because of the necessity to create hand-written IR as input for the evaluator.

Evaluating Partial Programs

To reduce the size of the state space that the implementation needs to compute, discrete sections of an MPI program could be compiled to IR and evaluated, rather than entire programs. For example, if a program contains an `MPI_Barrier` operation, which all processes execute, then this would be roughly equivalent to an `MPI_Finalize` since all processes will eventually perform this operation. The program could therefore be split in two at the barrier operation. Splitting programs and evaluating the parts in parallel would be another way to parallelize evaluation and improve performance.

Whether or not evaluating partial programs is feasible, and what modifications to the modelling approach, if any, this necessitates, is left for future work.

8.2.5 Integration into other Tools

In its current state, the implementation is not very useful. IR needs to be generated by hand to evaluate a program. The envisioned fully-featured implementation would be able to analyze whole MPI programs automatically and could perhaps be used integrated into other software. Like MPI-Checker (Droste et al., 2015), a finished implementation could be integrated into the analysis tools of LLVM.

9 Conclusion

The goal of this thesis was to explore the feasibility of using Colored Petri Nets to model Message Passing Interface communication. When compiling a C-language program that uses MPI, no checks about whether or not the implemented communication pattern is sound are made. It would therefore be useful to have some analysis tool that can assist in developing correct message passing applications. One future goal kept in mind throughout the thesis was to eventually build a tool that can build a CPN out of a C-language MPI program and perform analyses on the resulting net. Section 1.3 on page 2 proposed four leading questions to guide the thesis, repeated here:

1. Which individual MPI operations can be modelled, and how?
2. Can more complex MPI programs with multiple MPI operations be modelled, and if so, how?
3. What information from an MPI program is relevant for creating a CPN representation?
4. What kinds of errors can be detected using the CPN-based modelling approach, and how?

The modelling approach proposed in Chapter 3 on page 19 partially answered the first two questions, showing both how to model the semantics of individual MPI operations, as well as how to model the interaction of multiple operations in whole MPI programs. To keep the scope of the thesis manageable, though, only some of the most widely used operations were considered.

To answer the third question, an intermediate representation was designed and described in Chapter 4 on page 41. Since the IR needs to contain all information needed to construct a CPN, its design necessarily answers the question of what information is required to be extracted from a source C program.

The thesis project contains a practical aspect. Two major programs were written. First, "cpnets", a general-purpose CPN library. Second, a proof-of-concept IR evaluator, which takes as input an IR file, translates it into a CPN following the proposed modelling approach and evaluates it by computing its state space. The implementation was evaluated in Chapter 6 on page 57, and it was shown that the errors gathered in Section 2.1.1 on page 7 can be detected. It was therefore demonstrated that the modelling approach works in principle. The evaluation has also revealed undesirable performance characteristics of the "cpnets" library, and so a special-purpose variant was conceived and implemented, which greatly improved performance.

Chapter 7 on page 73 has shown that several other projects with a similar purpose exist, but most are incomplete, unmaintained, or challenging to build and use. This justifies the development of a new MPI analysis tool, such as the one partly created in this thesis.

The modelling approach and its implementation are not yet fully-featured or fully evaluated. As Chapter 8 on page 83 outlined, much work needs to be done to arrive at a tool that can check an arbitrary MPI program. Not nearly all operations were considered in the theoretical part of the thesis, and fewer operations still have been implemented in software. The approach has also only been evaluated on a small number of small programs.

Thus, while the work shown in this thesis is only a first step towards a fully-featured MPI analyser and evaluator tool, both the theoretical approach and the software implementation look promising. With a lot of future work, a very useful tool could be created.

Bibliography

Böhm, S., & Běhálek, M. (2012). Usage of Petri Nets for High Performance Computing. *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing*, 37–48. https://doi.org/10.1145/2364474.2364481

Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI*, *8*, 209–224.

Chen, M. (N.d.). *Petri Nets*. Retrieved February 5, 2023, from https://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq.html

Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., & Candea, G. (2010). Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, *43*(4), 5–10.

Droste, A., Kuhn, M., & Ludwig, T. (2015). MPI-Checker: Static Analysis for MPI. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. https://doi.org/10.1145/2833157.2833159

Dwyer, M. B., Edenhofner, J., Gopalakrishnan, G., Marianiello, A., Luo, Z., Rakamaric, Z., Rogers, M., Siegel, S. F., Zheng, M., & Zirkel, T. K. (2021). *The Concurrency Intermediate Verification Language Reference Manual* (v1.21). https://vsl.cis.udel.edu/lib/sw/civl/civl-manual.pdf

Fronc, Ł. (N.d.). *Neco net compiler*. Retrieved June 27, 2023, from https://code.google.com/archive/p/neco-net-compiler/

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). Elements of Reusable Object-Oriented Software. *Design Patterns*.

Gorlatch, S. (2004). Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *26*(1), 47–56.

Gropp, W. D. (2001). Learning from the Success of MPI. *International Conference on High-Performance Computing*, 81–92.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, *21*(8), 666–677.

Jensen, K., & Kristensen, L. M. (2009). *Coloured Petri Nets*. Springer Berlin, Heidelberg. https://doi.org/10.1007/b95112

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394.

LLVM Language Reference Manual. (N.d.). Retrieved April 19, 2023, from https://llvm.org/docs/LangRef.html

MCS Division. (N.d.). *A simple Jacobi iteration*. Retrieved May 24, 2023, from https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobi/C/main.html

Message Passing Interface Forum. (2021). *MPI: A Message-Passing Interface Standard*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

New Mexico State University. (2021). *MPI Introduction*. Retrieved October 15, 2022, from https://hpc.nmsu.edu/discovery/mpi/introduction/

Nguyen Ba, T., & Arora, R. (2018). Towards Developing a Repository of Logical Errors Observed in Parallel Code for Teaching Code Correctness. *2018 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, 69–77. https://doi.org/10.1109/EduHPC.2018.00011

Parsing pass (GNU Compiler Collection (GCC) Internals). (N.d.). Retrieved April 19, 2023, from https://gcc.gnu.org/onlinedocs/gccint/Parsing-pass.html

Peterson, J. L. (1977). Petri nets. *ACM Computing Surveys (CSUR)*, *9*(3), 223–252.

Petri, C. A. (1962). Kommunikation mit Automaten.

Pommereau, F. (2015). SNAKES: A Flexible High-Level Petri Nets Library (Tool Paper). In R. Devillers & A. Valmari (Eds.), *Application and theory of petri nets and concurrency* (pp. 254–265). Springer International Publishing.

Python Software Foundation. (2023). *Python*. Retrieved February 5, 2023, from https://python.org

Siegel, S. F. (2007). Model checking nonblocking MPI programs. *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 44–58.

Siegel, S. F., & Avrunin, G. S. (2004). *Modeling MPI programs for verification* (tech. rep.). Technical Report UM-CS-2004-75, Department of Computer Science, University ...

Siegel, S. F., & Avrunin, G. S. (2005). Modeling wildcard-free MPI programs for verification. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 95–106.

Siegel, S. F., Zheng, M., Luo, Z., Zirkel, T. K., Marianiello, A. V., Edenhofner, J. G., Dwyer, M. B., & Rogers, M. S. (2015). CIVL: the concurrency intermediate verification language. *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12. https://doi.org/10.1145/2807591.2807635

Siegel, S. F., & Zirkel, T. K. (2011). Automatic formal verification of MPI-based parallel programs. *ACM Sigplan Notices*, *46*(8), 309–310.

Sun, J., Liu, Y., Dong, J. S., & Pang, J. (2009). PAT: Towards flexible verification under fairness. *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*, 709–714.

The Open MPI Project. (2021). *mpicc(1) man page (version 4.0.7)*. Retrieved July 11, 2023, from https://www.open-mpi.org/doc/v4.0/man1/mpicc.1.php

Verified Software Laboratory, University of Delaware. (2015). *The Symbolic Algebra and Reasoning Library*. Retrieved July 10, 2023, from http://vsl.cis.udel.edu/sarl/

Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R. M., & Thakur, R. (2009). Formal verification of practical MPI programs. *ACM Sigplan Notices*, *44*(4), 261–270.

Yu, H., Chen, Z., Fu, X., Wang, J., Su, Z., Sun, J., Huang, C., & Dong, W. (2020). Symbolic verification of message passing interface programs. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 1248–1260.

List of Figures

2.1 A straightforward Petri Net with one place containing a single token, one empty place, and one transition. 13
2.2 The Petri Net from Figure 2.1 on page 13, after T_0 has fired once. 14
2.3 A Petri Net modelling a traffic light. 14
2.4 A dead Petri Net, due to the weight of the arc from P_0 to T_0. 14
2.5 A simple CPN, with just one place and a few tokens of type integer. 15
2.6 A CPN with an arc with an expression. 16
2.7 The CPN of Figure 2.6 on page 16 in its final state. 16
2.8 A CPN with a transition that has a transition guard. 17

3.1 `MPI_Bsend` and `MPI_Recv` operations. 21
3.2 `MPI_Ssend` and `MPI_Recv` operations. 22
3.3 `MPI_Ibsend`, `MPI_Recv`, and `MPI_Wait` operations. 23
3.4 `MPI_Issend`, `MPI_Recv`, and `MPI_Wait` operations. 23
3.5 Three processes using the `MPI_Barrier` operation. 24
3.6 Two processes that first perform a send/receive pair of operations and are then synchronized using `MPI_Barrier`. 25
3.7 Three processes taking part in an `MPI_Gather` operation. Note that the single operation is modelled as three separate transitions. 26
3.8 Three processes taking part in an `MPI_Allgather` operation. 26
3.9 Four processes partaking in an `MPI_Bcast` operation. P_0 is the root process, which broadcasts to all other processes, including itself. 27
3.10 A version of Figure 3.1 on page 21 that uses high-level CPN features. 29
3.11 Python class definitions used by the CPN in Figure 3.10 on page 29. 30
3.12 A CPN that models the entire MPI program shown in Listing 2.1 on page 6, instead of just individual operations. 32
3.13 A CPN modelling an MPI program with an unmatched `MPI_Ssend` operation. 34
3.14 A CPN modelling an MPI program with an unmatched `MPI_Bsend` operation. 35
3.15 A CPN modelling an MPI program with an unmatched `MPI_Recv` operation. . 36
3.16 A CPN modelling the program displayed in Listing 2.3 on page 8. Most arc annotations and all transition guards are omitted to help make this complicated net easier to understand. 38

4.1 A simple visualization of the proposed compilation pipeline. 41

5.1 “Flamegraph” showing that the code constructing a state graph spends the majority of its time in libpython3. 53

6.1 State graph order, by number of processes, compared to mathematical functions. 65
6.2 Execution times of revised implementation compared to general purpose implementation. 71

List of Listings

2.1 A simple MPI program that demonstrates MPI_Send and MPI_Recv. 6
2.2 A simple MPI program that demonstrates MPI_Allreduce. 7
2.3 Two processes deadlock because they each call MPI_Recv and wait for one another indefinitely. 8
2.4 An example of an unmatched send operation in an MPI program. 9
2.5 AGSearch repository query file. 10

4.1 Intermediate representation of Listing 2.3 on page 8. 44

5.1 Rust definition of the "Net", the principal component of the cpnets crate. . . . 48
5.2 Rust definition of the "Place" struct, and "Capacity" enum. 48
5.3 Rust definition of the "Transition" struct. 48
5.4 Rust definition of the "Arc" struct and its two inner types. 49
5.5 A Rust program utilizing the cpnets crate to create the traffic light net depicted in Figure 2.3 on page 14. 51
5.6 Rust definition of the "Marking" struct. 51
5.7 An example of how a marking is constructed for a net. 52
5.8 Rust definition of the state graph struct. 52
5.9 "pest" grammar for the CPN IR defined in Chapter 4 on page 41. 55

6.1 IR of the deadlock example shown in Listing 2.3 on page 8. 57
6.2 Output of the evaluator when given the IR shown in Listing 6.1 on page 57. . 58
6.3 IR of a very simple unmatched MPI_Ssend program. 58
6.4 Output produced by the evaluator when given the IR shown in Listing 6.3 on page 58 58
6.5 IR of an unmatched MPI_Bsend example program. 59
6.6 Evaluator output when given the IR from Listing 6.5 on page 59. 59
6.7 Evaluator output when given the unmatched MPI_Ssend IR. 60
6.8 IR of an unmatched MPI_Recv example program. 60
6.9 Evaluator output when given the IR from Listing 6.8 on page 60. 60
6.10 IR of a send operation with a tag mismatch. 61
6.11 Evaluator output when given the IR from Listing 6.10 on page 61. 61
6.12 IR of the Jacobi program, using an infinite loop. 63
6.13 IR of the Jacobi program, using a single-iteration loop. 63
6.14 Revised Expression definition. Code comments and derive macros are omitted for brevity. 69

6.15 Revised `Guard` definition. Code comments and `derive` macros are omitted for brevity. 69
6.16 Revised `struct Token` definition. Code comments and `derive` macros are omitted for brevity. 70

7.1 CIVL 1.21 invocation and output with an included example file, already in the CIVL language. 74
7.2 CIVL 1.21 invocation and output with the example program shown in Listing 2.3 on page 8. 75
7.3 CIVL 1.20 invocation and output with a program that uses `MPI_Bsend`. . . . 75
7.4 Setup, invocation, and output of the current "master" branch state of the pre-compiled distribution of MPI-SV. 77
7.5 Running MPI-Checker on the repository of MPI error examples described in Section 2.1.1 on page 7. Note that no errors are reported. 79
7.6 Excerpt of the compiler output when attempting to build the In-Situ Partial Order (ISP) software. The compilation was triggered by first running the included `configure` script, followed by an invocation of `make`. 81

List of Tables

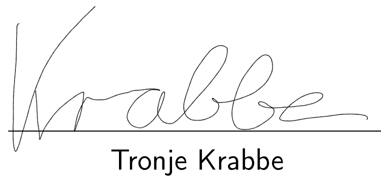
2.1 Usage of MPI Wildcard Parameters 11
2.2 The 20 most-used MPI Operations by Users and Usage 12
2.3 Most-used MPI Operations extracted from Table 2.2 on page 12 12

6.1 Order and size of state graphs for the four different CPN variants. 64
6.2 Evaluation speeds and peak memory usage. 67
6.3 Performance per state, relative to total state graph size. 67
6.4 Revised implementation speeds and peak memory usage. 70

7.1 Performance comparison between CIVL and the IR evaluator written for this thesis. 76
7.2 Comparison of state graph build time between SNAKES and cpnets. 82

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.



Hamburg, den 24.07.2023

Tronje Krabbe

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 24.07.2023

Tronje Krabbe