Imperial College London
Department of Computing

Safe and Scalable Parallel Programming with Session Types

Chun Wang Nicholas Ng

April 2015

Supervised by Nobuko Yoshida

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Chun Wang Nicholas Ng

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Abstract

Parallel programming is a technique that can coordinate and utilise multiple hardware resources simultaneously, to improve the overall computation performance. However, reasoning about the communication interactions between the resources is difficult. Moreover, scaling an application often leads to increased number and complexity of interactions, hence we need a systematic way to ensure the correctness of the communication aspects of parallel programs.

In this thesis, we take an interaction-centric view of parallel programming, and investigate applying and adapting the theory of *Session Types*, a formal typing discipline for structured interaction-based communication, to guarantee the lack of communication mismatches and deadlocks in concurrent systems. We focus on scalable, distributed parallel systems that use message-passing for communication. We explore programming language primitives, tools and frameworks to simplify parallel programming.

First, we present the design and implementation of *Session C*, a programming toolchain for message-passing parallel programming. Session C can ensure deadlock freedom, communication safety and global progress through static type checking, and supports optimisations by refinements through session subtyping. Then we introduce Pabble, a protocol description language for designing parametric interaction protocols. The language can capture scalable interaction patterns found in parallel applications, and guarantees communication-safety and deadlock-freedom despite the undecidability of the underlying parameterised session type theory. Next, we demonstrate an application of Pabble in a workflow that combines Pabble protocols and computation kernel code describing the sequential computation behaviours, to generate a Message-Passing Interface (MPI) parallel application. The framework guarantees, by construction, that generated code are free from

communication errors and deadlocks. Finally, we formalise an extension of binary session types and new language primitives for safe and efficient implementations of multiparty parallel applications in a binary server-client programming environment.

Our exploration with session-based parallel programming shows that it is a feasible and practical approach to guaranteeing communication aspects of complex, interaction-based scalable parallel programming.

Acknowledgements

It has been a wonderful journey, during which I was very fortunate to have worked with and learnt from a lot of amazing people. I take this opportunity to thank you for all the guidance and support I have received:

I thank my supervisor Nobuko Yoshida for offering me the chance to work with her. Through her guidance and patience, I managed to stay on track most of the time despite my tendency to run into the problem of state explosion in research explorations! The energy and work ethics are something I can only dream of matching, and she was right about the long nights and working weekends – the results were well worth the effort.

I very much appreciate the detailed and constructive feedback on my thesis from my examiners Paul Kelly and Fritz Henglein. Most of their suggestions were included in this much improved final thesis.

I thank my second supervisor Wayne Luk for his many helpful advices. Speaking to Wayne is always positive and encouraging, and often open up new perspectives to look at my own work.

Kohei Honda, who was a collaborator on my first work, but unfortunately passed away in 2012 due to illness, will always be remembered. Kohei was a visionary and widely regarded as the father of Session Types – I am honoured to have worked with him, his passion and insight have had a long lasting influence on me and my research.

It would be a dull and lonely journey without my colleagues and friends, especially everyone from the *Mobility Reading Group*, the *Custom Computing Group* and the team from our *ACM Student Chapter*. In particular, I have had a lot of productive discussions with my colleagues Pierre-Malo Deniélou, Raymond Hu, Dimitrios Kouzapas, Rumyana Neykova, Olivier Pernet and my collaborators Xin Yu Niu and José G.F. Coutinho, and also at the University of Lisbon: Francisco Martins, Eduardo R. B. Marques and Vasco T. Vasconcelos.

Last but not least, I thank my family for supporting me to take on this challenge. I am unlikely to complete this journey without the love and understanding (of my grumpiness or abruptly hung up phone calls) from afar throughout all the years of me studying abroad.

Contents

1. Introduction **18**
1.1. Motivation and Objectives 18
1.2. Thesis Summary 21
1.3. Thesis Structure 23
1.4. Publications and Software 23
1.4.1. Workshops 25
1.4.2. Conferences 25
1.4.3. Journals 27

2. Background **28**
2.1. Formal Models of Concurrency and Session Types 29
2.1.1. Session Calculus 29
2.1.2. Operational Semantics of the Session Calculus 32
2.1.3. Duality and Session Types 32
2.1.4. Multiparty Session Types 35
2.1.5. Parameterised Multiparty Session Types 37
2.1.6. Subtyping and Asynchronous Subtyping 39
2.1.7. Scribble Protocol Description Language 40
2.2. Parallel Hardware Architectures and Programming Models . . 41
2.2.1. Superscalar Processors 41
2.2.2. Multicore Processors 42
2.2.3. Distributed and Cluster Computing 44
2.3. State-of-the-art Languages and Tools for Parallel Programming 45
2.3.1. Threads and Locks 45
2.3.2. Message-Passing 46
2.3.3. PGAS Languages 47
2.3.4. Heterogeneous Computing 47

2.3.5. Session-based Languages 48
2.4. Summary 49

3. The Session C Programming Framework 50
3.1. Introduction 50
3.2. Protocols and Programming in Session C 54
3.2.1. The Scribble Protocol Description Language 54
3.2.2. Session C: Programming and Runtime 58
3.3. Type Checking and Message Optimisation 61
3.3.1. Session Type Checker 61
3.3.2. Asynchronous Message Optimisation 62
3.4. Parallel Algorithms in Session C 66
3.4.1. N-body Simulation: Optimised Ring topology 67
3.4.2. Linear Equation Solver: Wraparound Mesh Topology . 68
3.5. Performance Evaluation 70
3.6. Summary and Discussion 73

4. Pabble: Parameterised Scribble 75
4.1. Introduction 75
4.1.1. Overview 77
4.2. Design aims 78
4.3. Pabble: Parameterised Scribble 81
4.3.1. Syntax of Pabble 81
4.3.2. Well-formedness Conditions: Index Binding 85
4.3.3. Well-formedness Conditions: Constants 86
4.3.4. Endpoint Projection 88
4.3.5. Collective Operations 91
4.3.6. Termination and Correctness of the Projection 91
4.3.7. Correctness of projection 95
4.4. Pabble Examples 98
4.4.1. Projection Example: Ring Protocol 98
4.4.2. Projection Example: ScatterGather Protocol 99
4.4.3. Use Case: Web Services 101
4.4.4. Use Case: RPC Composition 104
4.4.5. Implementation Example: Linear Equation Solver . . 105
4.5. Summary and Discussion 107

5. Scalable High-performance Session Programming 109
5.1. Introduction 109
5.2. Case Studies 110
5.2.1. N-body Simulation 110
5.2.2. K-means clustering 111
5.2.3. Dense matrix-vector multiplication 113
5.2.4. Sparse matrix-vector multiplication 115
5.3. Runtime Analysis 118
5.4. Summary and Discussion 120

6. Safe MPI Code Generation with Pabble 121
6.1. Introduction 121
6.2. Application Development Workflow 124
6.2.1. Interaction protocols with Pabble 124
6.2.2. Computation kernels 125
6.3. Compilation Step 1: Protocol to MPI backbone 128
6.3.1. MPI backbone generation from `Stencil` protocol . . . 129
6.3.2. MPI backbone generation from Pabble 130
6.4. Compilation Step 2: Aspect-Oriented Design-Flow 136
6.4.1. Merging process 137
6.4.2. Performance optimisation for overlapping communication and computation by MPI immediate operators 139
6.5. Evaluation 141
6.5.1. Productivity and Reusability 141
6.5.2. Performance 144
6.5.3. Limitations 146
6.6. Summary and Discussion 147

7. Multi-channel Session Types 148
7.1. Introduction 148
7.2. Session-Typed Programming in SJ 150
7.3. Parallel Algorithms in SJ 155
7.3.1. N-body simulation: Ring Topology 156
7.3.2. Jacobi solution of the discrete Poisson equation: Mesh Topology 158
7.3.3. Linear Equation Solver: Wraparound Mesh Topology . 161

7.4. Multi-channel Session Calculus 162
7.4.1. Syntax 163
7.4.2. Operational Semantics 164
7.4.3. Types, Typing System and Well-Formed Topologies . 167
7.4.4. Typing Rules 170
7.4.5. Well-formed Topologies 170
7.4.6. Subject Reduction, Communication Safety and Deadlock Freedom 173
7.5. Performance Evaluation 174
7.6. Summary and Discussion 175

8. Conclusion and Future Work 177
8.1. Related Work 177
8.1.1. Formally-founded Communications Programming Languages 177
8.1.2. Parameterised Multiparty Session Types 178
8.1.3. Formal Analysis of Parallel Applications 178
8.1.4. MPI code generation 180
8.1.5. Communication Optimisations in MPI 180
8.1.6. Dependent Typing Systems 181
8.2. Conclusion 181
8.2.1. Session C 181
8.2.2. Pabble 182
8.2.3. Multi-channel Primitives 182
8.2.4. Challenges 183
8.3. Future work 183
8.3.1. Session C and Type Checking 183
8.3.2. Pabble and Code Generation 184
8.3.3. Multi-channel Primitives 185

Bibliography 186

A. Appendix 199
A.1. Appendix for Chapter 7 199
A.1.1. Multi-channel Session Typing Rules 199
A.1.2. Well-Formed Ring and Mesh Topologies 199

A.1.3. Proofs for Multi-channel Session Calculus Subject Reduction, and Deadlock Freedom 203
A.1.4. Full Process Definitions 220
A.1.5. 3-node n-body simulation 220
A.1.6. Jacobi solution 221

List of Figures

1.1. Safe and Scalable Parallel Programming with Session Types. 24

2.1. Session Calculus: Syntax. 30
2.2. Session Calculus: *SumServer* Example. 31
2.3. Session Calculus: Structural Congruence. 32
2.4. Session Calculus: Reduction Rules. 33
2.5. Session Type: Syntax. 34
2.6. Multiparty Session Types Framework. 36
2.7. Multiparty Session Calculus: Syntax. 37
2.8. Multiparty Session Types: Type Syntax. 38
2.9. Global Type Reduction of Parameterised MPST. 39
2.10. The Scribble Protocol Description Language. 40
2.11. Superscalar execution. 42
2.12. Multicore Processor and Multi-processor Architecture. 43
2.13. Two examples of distributed systems. 45

3.1. Session C Programming Framework. 52
3.2. Session C workflow. 53
3.3. Ring topology of the 3-role `Nbody` protocol. 67
3.4. Benchmark results. 71
3.5. Butterfly Pattern. 72
3.6. FFT butterfly pattern with 8 processes. 73

4.1. Pabble programming workflow. 77
4.2. Pabble syntax. 82
4.3. Point-to-point communication and Pabble representation. . . 92
4.4. Scatter pattern and Pabble representation. 92

4.5. Gather pattern and Pabble representation. 92
4.6. All-to-all pattern and Pabble representation. 93
4.7. Topology of the ScatterGather protocol. 101
4.8. Web Services Quote-Request interaction. 103
4.9. RPC request/response chaining. 105
4.10. N^2-node wraparound mesh topology. 107

5.1. Partitioning of input matrix. 114
5.2. Comparing Session C and MPI performance for different number of spawned processes from the same protocol, testing how the two runtimes scale 119

6.1. Pabble-based code generation workflow. Shaded boxes indicate user inputs. 123
6.2. Messages received by a process in a stencil protocol. 125
6.3. Global view of `Label(T)from Sender to Receiver;`. 126
6.4. N-body simulation (nbody). 144
6.5. Linear Equation Solver (LEsolver). 145
6.6. Word Count (wordcount) and AdPredictor (adpredictor) . . . 145
6.7. Parallelisation speedup. 146

7.1. Simple chaining of session iterations across multiple pipeline process. 152
7.2. Multi-channel Iteration in a simple grid topology. 153
7.3. The SJ tool chain. 154
7.4. Communication patterns in n-body implementations. 157
7.5. Implementation of the ring topology, single-channel primitives only. 157
7.6. Improved implementation of the ring topology using multi-channel primitives. 157
7.7. Communication patterns in Jacobi method implementations. 159
7.8. Jacobi method implementation in a 2D mesh topology with single-channel primitives. 160
7.9. Jacobi method implementation in a 2D mesh topology with *multi-channel* primitives. 160
7.10. Linear Equation Solver: Wraparound Mesh Topology. 162
7.11. Multi-channel Session Calculus: Syntax. 165

7.12. Multi-channel Session Calculus: Reduction rules. 166
7.13. Multi-channel ordering. 172
7.14. Benchmark results: 3 nodes n-body simulation. 175
7.15. Benchmark Results: 9 nodes Jacobi solution. 176

A.1. Multi-channel Session Types: Typing rules. 200
A.2. Ring, mesh, and wraparound mesh topologies, with rank annotations. 203

List of Tables

2.1. Dual Actions in the Session Calculus. 33

2.2. Duality in Session Types. 35

3.1. Projection from global protocol to endpoint protocol of `myrole`. 56

3.2. Session C primitives. 59

4.1. Projection of G onto $\boldsymbol{R}$ at the end-point role R_e. 89

4.2. Examples of `apply()` and `inv()`. 90

6.1. Pabble interaction statements and their corresponding code. . 131

6.2. Pabble control-flow statements and their corresponding code. 133

6.3. Pabble choice and their corresponding code. 133

6.4. Pabble collective operations and their corresponding code. . . 135

6.5. Annotations in backbone and kernel. 138

List of Acronyms

API Application Programming Interface. 21, 23, 45, 46, 53, 70, 73, 79, 106, 119, 120, 126

CCS Calculus of Communicating Systems. 29

CPU Central Processing Unit. 18, 41, 42, 118

CSP Communicating Sequential Processes. 29, 177, 178

CUDA Compute Unified Device Architecture. 47

FFT Fast Fourier Transformation. 38, 72

FPGA Field-Programmable Gate Array. 47, 122, 185

GPU Graphics Processing Unit. 47

HPC High Performance Computing. 46, 47, 149

HTM Hardware Transactional Memory. 44

LLVM Low-Level Virtual Machine. 61

MPI Message-Passing Interface. 13, 21–27, 46–48, 50, 51, 54, 55, 59, 70–72, 75, 78–81, 84, 98, 106, 109–114, 116–124, 126, 128–132, 134, 136–141, 143, 144, 175–180, 182–184

MPST Multiparty Session Types. 10, 12, 22, 28, 36, 38–41, 51, 56, 73, 78, 95, 96, 107, 108, 121, 147, 148, 175, 178, 181–183

OOI Ocean Observatories Initiatives. 41

OpenCL Open Computing Language. 47

OpenMP Open Multi-Processing. 46, 137

PE Processing Element. 48

PGAS Partitioned Global Address Space. 7, 44, 45, 47, 121

SJ Session Java. 9, 13, 22, 26, 48, 148–156, 163, 167, 174–176, 182

SPMD Single Program, Multiple Data. 47, 79

STM Software Transactional Memory. 44

VCC A Verifier for Concurrent C. 25, 180

VHDL VHSIC Hardware Description Language. 47

1 Introduction

Concurrency is the composition of independently executing processes; Parallelism is the simultaneous execution of computations.
Concurrency is about dealing with lots of things at once; Parallelism is about doing lots of things at once.

– Rob Pike, *Concurrency is not Parallelism*

1.1. Motivation and Objectives

Computer Science is the study of solving problems with computers. Programmers express their imagination to codify everyday repetitive and mundane tasks into algorithmic recipes for computer hardware, which can execute them efficiently and accurately. These algorithms, like the human thought process, are structured, step-by-step, and inherently sequential.

For a very long period of development in computing, performance and efficiency had relied on improvements on increasing speed and throughput of sequential hardware. This was coupled with the fact that, the majority of computer programs written for these hardware are sequential. While parallelism has been exploited in micro scale as instruction-level parallelism and hidden from users, mainstream parallel computing did not take off before the turn of the century, when the growth of microprocessor performance was slowed down due to a number of factors such as the *power wall* — the reduction of performance due to overheating and high power consumption of the microprocessor – and the physical limits of shrinking CPU die size to reduce power consumption. This resulted in the rise of utilising parallel hardware to further increase performance.

Parallel programming, or using multiple resources to work on a problem simultaneously to find a solution, is not as simple as it sounds. Apart from a class of *embarrassingly parallel* problems with completely independent sub-problems, parallelisation involves splitting a problem into smaller, related sub-components and synchronising between the sub-components to reach a consensus on a single, complete solution. A real world analogy is organising a party: *some participants will bake a cake, some will prepare salad and some will buy drinks and cups. Since everyone acts independently, if there is no coordination between the participants, the party may end up with 3 cakes and no drinks because everyone assumed someone else will be responsible for the drinks. The process of coordination and synchronisation can be solved by everyone having access to shared resources, e.g. a shared whiteboard, which everyone can state what they plan to do for the party; or if the participants have to organise the party without meeting physically, they can interact explicitly, e.g. discuss over the phone.* This closely resembles what happens in computing: replace *participants* with *processes* and *shared whiteboard* with *shared memory*, there is the shared memory paradigm for parallel programming; and interpreting *discussions over the phone* as a form of point-to-point *message-passing* between processes, gives a typical scenario of the distributed memory programming paradigm.

This work focuses on the latter model — interaction-centric concurrency by explicit message-passing. Interaction is a general and powerful concept, in fact, most everyday objects are compositions of independent entities interacting with each other. Because of the distributed nature of the model, it can also emulate the shared resources model: just imagine *having a whiteboard at one of the party participant's site to write down the complete plan of the party.*

Unfortunately, interaction-based concurrency is difficult to get right. As interaction is just a means of coordination, there are no guarantees about correctness or compatibility between each participant. If we revisit the party example, *two participants talking on the phone may have different expectations from each other: both of them start the conversation by saying what they are going to buy without listening to what their counterparts are saying, so at the end of the conversation, both assumed they are baking a cake* – this is incompatible communication; *if both ends of the phone wait politely wait for the other side to speak first; both wait indefinitely with no progress* – this is a communication deadlock. These two communication-related problems are the central issues this thesis tackles, with the help of explicitly specified communication types, known in the literature as Session Types (see Section 2.1), or communication protocols. Intuitively, the idea is no different from having a shared *social protocol* between the individuals, where *both participants speak one after another, and follow a pre-agreed order of speaking.* Just as the concept of types for computations are modelled with mathematics and logics, types for communication are modelled with process calculi, and they are defined to govern the behaviours of interacting systems as formal specifications. With a correct model of concurrency, the difficulty of putting parallelism in practice is greatly reduced.

Another aspect of our interaction-centric approach to parallel programming is scalability. In parallel programming terms, given a problem and a solution for the problem, if we have N-times the resources to solve a problem, can we solve it N-times faster? The increased number of components means that there will be more coordination and synchronisation by interactions between the components to collaboratively devise a solution. A concurrency abstraction for a parallel application needs to cope with such scaling without losing expressiveness.

Ultimately, understanding concurrency in Computer Science is not only for the peace of mind, but for applying them on computers to solve problems. Providing tools to help programmers understand and apply concurrency correctly is just as important as coming up with the abstraction. A programming primitives and language-based approach ensures that implementing the correct interactions is not an afterthought but a part of the design thought process.

The primary hypothesis of this thesis can be summarised in two questions: *Is it feasible to apply and adapt Session Types, as a high-level specification and verification technique, for parallel programming?* and *How can we apply our methodology practically to scalable parallel applications, for correct interaction-based parallel programming?* This thesis describes two approaches to achieve the goals: static type checking on custom programming primitives and code generation from communication protocols. We also describe a formal extension of existing session typing discipline to support efficient and safe parallel programming.

1.2. Thesis Summary

This thesis describes how the theory of Session Types can be applied to parallel computing, to guarantee the correctness of communication within a parallel application.

First, we introduce **Session C**, a programming framework for message-passing parallel algorithms centering on explicit, formal description of global protocols, and examines its effectiveness through an implementation of a toolchain for C. The framework is based on the theory of *Multiparty* Session Types, and uses a protocol description language Scribble for describing multiparty session types in a Java-like syntax. The programming environment of Session C is made up of two main components, the runtime library and the session type checker. The Session C runtime is a simple and lightweight communication library, providing basic primitives for message-passing programming. The session type checker is a static analyser for verifying the source code given conforms to its corresponding protocol specification in Scribble. Scribble, combined with the Session C runtime and the static type checker supports a full guarantee of deadlock-freedom, type-safety, communication-safety and global progress for all well-typed programs.

Next, we present an extension of Scribble with dependent types called *Parameterised Scribble* (**Pabble**). In Pabble, multiple participants can be grouped in the same role and indexed, where *parameterised* refers to the number of participants in a role that can be changed by parameters. This extension greatly enhances the expressive power and the modularity of the protocols for describing *scalable* parallel applications, such as ones that are written in Message-Passing Interface (MPI), the standard API for developing message-passing based parallel applications. Both the indexed dependent type theory in the λ-calculus and the parameterised session type theory, which is the theoretical basis of Pabble, shows that the projection and type checking with general indices are undecidable. Pabble's approach of extending Scribble with index notation overcame the tension between termination and expressiveness to make the theory more practical. Our compact notation is expressive enough to represent communication topologies in parallel applications as well as distributed web services, and offers a solution to cope with the undecidability of parameterised multiparty session types.

With Pabble, we present a protocol-driven parallel application code generation workflow. The workflow leverages the safety guarantees of valid Pabble protocols to generate a communication-safe-by-construction MPI parallel application backbone. Based upon the backbone, computation kernels are developed in C to describe the sequential behaviour of the application, which are then merged automatically and systematically by an aspect-oriented framework to output a complete MPI parallel application. The MPI backbone can be source-transformed to perform asynchronous optimisations, which allows effective overlapping of communication and computation without compromising the safety guarantees of the MPI backbone. Through a number of examples we show the performance and also the flexibility of the approach by separating the communication and computation concerns. This confirms that session-based code generation can be applied to scalable parallel applications and simplify the parallel development process.

Finally, we describe **Multi-channel Session Types** and an implementation of the theory as **multi-channel session primitives** in the session-typed programming language Session Java (SJ). This represents a compositional approach to parallel applications development, introducing new language primitives for *chained iteration* and *multi-channel communication*. SJ only guarantees progress for each session in isolation, but communication errors and deadlocks can still arise from interleaving of multiple sessions in a process. The combination of multi-channel session primitives for binary sessions, and a well-formed communication topology checker brings the benefits of type-safe, structured communications to SJ. We formalise the primitives as extensions of the session calculus and the correctness conditions on the 'shape' of program communication topology. We then prove the communication-safety and deadlock-freedom properties of our approach as a compositional, lightweight alternative to Multiparty Session Types for global type-safety.

Note on terminologies The terms *protocol* and *types* are used interchangeably throughout the thesis, depending on the context and perspective when using the terms: *Types* refers to the theoretical underpinnings of Session Types as a mathematical concept, whereas *protocol* refers to the practical realisation of the types as a developer-friendly language (such as Scribble and Pabble). We sometimes refer to *local types* in multiparty sessions as *endpoint protocols* for a similar reason.

1.3. Thesis Structure

- Chapter 2 presents the background materials on session types, parallel architectures and models.
- Chapter 3 presents the Session C programming framework. Session C is a session-based programming tool chain which consists of a runtime API and a static type checker, which guarantees deadlock freedom, type and communication safety at compile time.
- Chapter 4 presents a parametric protocol description language Pabble, designed for expressing parallel interactions between processes.
- Chapter 5 presents case studies to show how session-based static type checking and code generation can tackle the challenges of communication-safe and deadlock-free parallel programming. The evaluation results show that MPI is more suitable as a target runtime API for scalable parallel applications.
- Chapter 6 presents a complete code generation approach to session programming, combining Pabble, and aspect-oriented programming for a top-down workflow for safely parallelising MPI applications.
- Chapter 7 presents the formalisation and implementation of a new pair of session primitives to synchronise multiple sessions running in parallel. The primitives allow session types to express parallel topologies as multiple interleaved sessions such that each session can be defined separately.
- Chapter 8 presents the conclusion and future work of the thesis.

Figure 1.1 shows how the chapters are related.

1.4. Publications and Software

The work presented in this thesis resulted in several publications and software.

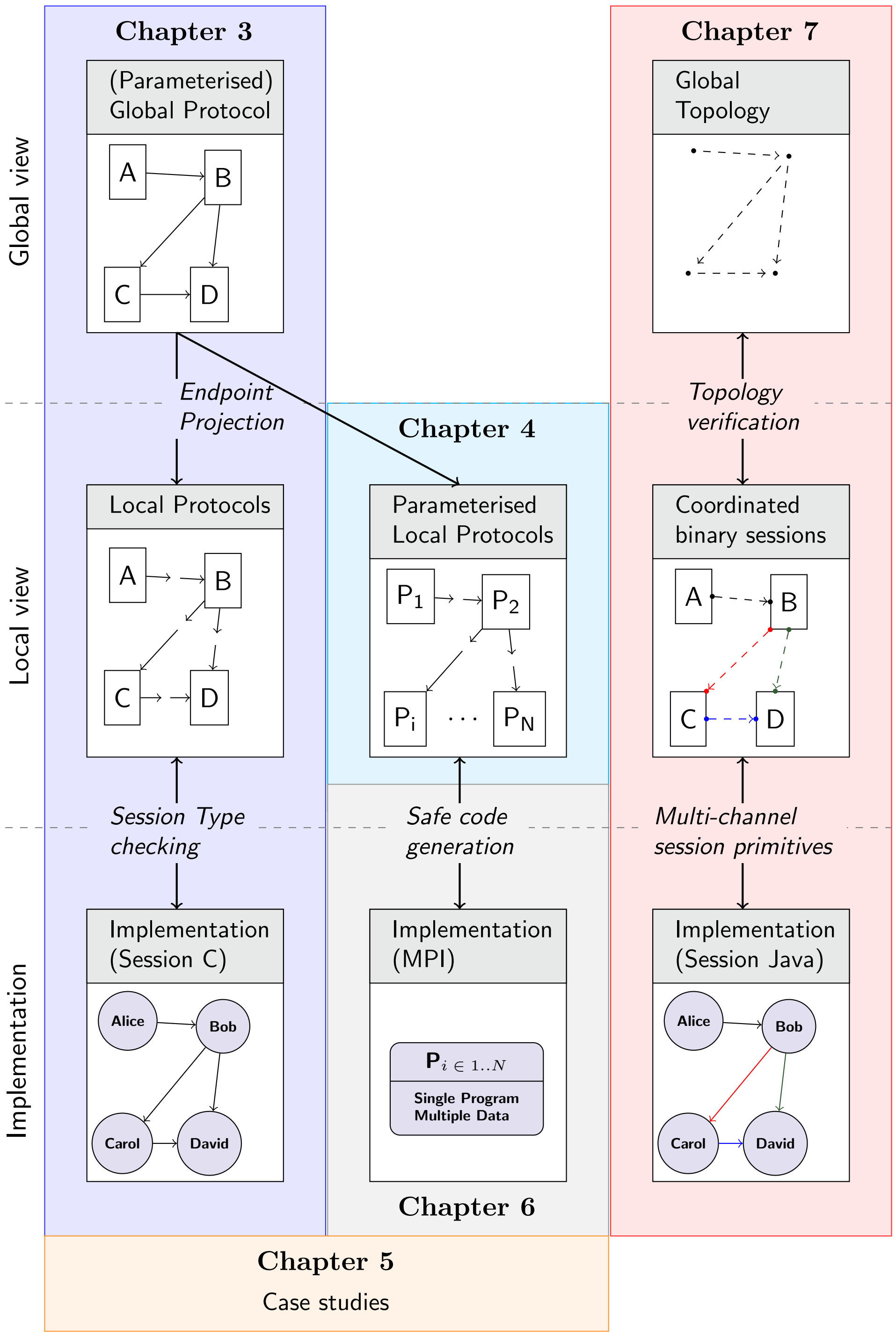


Figure 1.1.: Safe and Scalable Parallel Programming with Session Types.

1.4.1. Workshops

- **Nicholas Ng**, Nobuko Yoshida, Xin Yu Niu, K.H. Tsoi and Wayne Luk. *Session Types: Towards safe and fast reconfigurable programming* [NYN$^+$12]. In 3rd International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2012), May 2012, published in ACM SIGARCH Computer Architecture News (CAN) Volume 40 Issue 5.

  This work presents a new approach to describing safe communication topologies for heterogeneous computing with Scribble and a case study to applying the Session C programming framework in conjunction with high performance acceleration hardware in a heterogeneous parallel computing environment. This work forms part of Chapter 5.

- Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, **Nicholas Ng** and Nuno Martins. *Towards deductive verification of MPI programs agains session types* [MMV$^+$13]. In Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'13), March 2013.

  This work introduces an annotated MPI library to be used with A Verifier for Concurrent C (VCC) to ensure communication safety and deadlock free in MPI applications. Author was involved in the discussion on common parallel programming patterns and challenges of static type-checking.

- **Nicholas Ng**, Nobuko Yoshida and Wayne Luk. *Scalable Session Programming for Heterogeneous High-Performance Systems* [NYL13]. In 2nd International Workshop on Behavioural Types, September 2013, published in SEFM 2013 Collocated Workshops, LNCS 8368.

  This work describes two approaches to apply session programming in scalable heterogeneous computing. This work forms part of Chapter 5.

1.4.2. Conferences

- **Nicholas Ng**, Nobuko Yoshida, Olivier Pernet, Raymond Hu and Yiannos Kryftis. *Safe Parallel Programming with Session Java* [NYP$^+$11].

In 13th International Conference on Coordination Models and Languages (COORDINATION), June 2011.

This work introduces the multi-channel session calculus and the multi-channel Session Java (SJ) which allows the chaining of binary sessions in SJ for developing parallel applications. Chapter 7 is an expanded version of this work.

- **Nicholas Ng**, Nobuko Yoshida and Kohei Honda. *Multiparty Session C: Safe Parallel Programming with Message Optimisation* [NYH12]. In 50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe 2012), May 2012.

  This work presents a efficient programming toolchain for message passing-based parallel algorithms which can ensure, for any typable programs and for any execution path, a full guarantee of deadlock-freedom, communication safety and global progress through satatic checking. The methodology is embodied as a multiparty session-based programming environment for C and its runtime libraries which we call Session C. The source code Session C programming framework is available online at `https://github.com/nickng/sessc`. Chapter 3 is an expanded version of this work.

- Kohei Honda, Eduardo R B Marques, Francisco Martins and **Nicholas Ng**, Vasco T. Vasconcelos and Nobuko Yoshida. *Verifications of MPI Programs using Session Types* [HMM$^+$12]. In 19th European MPI Users' Group Meeting (EuroMPI 2012), September 2012.

  This extended abstract with poster is a proposal for developing a session-based protocol language similar to Scribble based on the Message-Passing Interface (MPI) primitives.

- **Nicholas Ng** and Nobuko Yoshida. *Pabble: Parameterised Scribble for Parallel Programming* [NY14b]. In 22nd Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2014), February 2014.

  This work presents a parameterised protocol description language, Pabble, which can guarantee safety and progress in a large class of practical, complex parameterised message-passing programs through static checking. The supporting tool is available online at `https://github.com/sessionc`. Chapter 4 is an expanded version of this work.

- **Nicholas Ng**, Jose G.F. Coutinho and Nobuko Yoshida. *Protocols by Default: Safe MPI Code Generation based on Session Types* [NCY15]. In 24th International Conference on Compiler Construction (CC 2015), April 2015.

  This work presents a code generation workflow from Pabble protocols. The workflow systematically combines an MPI communication backbone generated from Pabble with user-defined computation kernels, resulting in a safe-by-construction parallel application. This paper is a extended version of Chapter 6 with additional benchmark results and minor refinements on the annotation syntax.

1.4.3. Journals

- **Nicholas Ng** and Nobuko Yoshida. *Pabble: parameterised Scribble* [NY14a]. In Journal of Service Oriented Computing and Applications (SOCA), December 2014 (Special Issue).

  This is an expanded version of [NY14b], which include more examples use cases of the Pabble language in the context of web services, and a more complete survey of related works. Chapter 4 is an expanded version of this work.

2 Background

Types are the leaven of computer programming; they make it digestible.
– Robin Milner

In the last chapter, we discussed briefly the importance of interactions and having a formal model of concurrency in order to apply concurrency to parallel hardware. This chapter presents the background materials for understanding the contributions of this thesis.

First, we introduce the theory of Session Types (Section 2.1), which is the formal foundation of this work. Session Types is a typing discipline that guarantees type and communication safety between two interacting processes. We present a major extension of Session Types, the Multiparty Session Types (MPST), in Section 2.1.4, which can type more than two processes in a single session, providing a global view of all interactions. We cover a parametric variant, *Parameterised* Multiparty Session Types in Section 2.1.5 where processes are indexed by integer indices to increase expressiveness. We also present a theory extension for asynchronous communication in Section 2.1.6, where the rules and conditions of correct asynchronous interaction in MPST are discussed. The background of session types is concluded by introducing the Scribble protocol description language. Scribble is a developer-friendly language designed to be a practical counterpart of MPST, which this thesis uses and extends for MPST-based programming.

Next, we present an overview of parallel architectures and programming models relevant to this work (Section 2.2).

2.1. Formal Models of Concurrency and Session Types

Process calculi are a family of formal approaches for modelling interactions in concurrent, communicating systems. They are the fundamentals for reasoning about concurrency, similar to what λ-calculus is to reasoning about computation. In particular, Tony Hoare's Communicating Sequential Processes (CSP) [Hoa78] and Robin Milner's Calculus of Communicating Systems (CCS) [Mil80] and π-calculus [MPW92] were the earliest and most influential approaches.

Session Types was originally proposed by Honda, Vasconcelos and Kubo as a type discipline for session language primitives in [HVK98] and is closely related to process calculi. The proposed session language consists of basic primitives for interactions, and structuring constructs such as conditionals (if-then-else) and iterations (loops) for combining them, and can be easily encoded into asynchronous π-calculus [HT91]. It proves to be a powerful high-level abstraction as process calculi lack elements for describing program control flows. A *session* is the sequence of structured interactions by message exchanges, and the language is formalised as a *session calculus*, which is the building block of session types, defining its operational semantics as reduction rules of the calculus. The session calculus presented here is specific to binary session types in [HVK98]. Most decendents and extensions of session types, such as the ones we present later in Chapter 7, are based upon or are similar to this minimal calculus.

2.1.1. Session Calculus

The syntax of session calculus is presented in Figure 2.1. The calculus is defined recursively as prefixed actions and combine as processes P. *Session request* and *session acceptance* are primitives for symmetric initialisation, which setup a new shared channel k over an existing channel a for subsequent interactions in their continuations P. In the syntax presented, a, b, ... are *names*; k, k', ... are *channels*; e, e', ... are *expressions*; c, c', ... are *constants*; l, $l_{i\in\mathbb{N}}$, ... are *labels*; X, Y, ... are *processed variables*; u, u', ... are *names and channels*; and P, Q, ... are *processes.* $\tilde{}$ represents a potentially empty vector.

P	$::= \mathsf{request}\, a(k)\ \mathtt{in}\, P$	session request
	$\mid \mathsf{accept}\, a(k)\ \mathtt{in}\, P$	session acceptance
	$\mid k![\tilde{e}]; P$	data sending
	$\mid k?(\tilde{x})\ \mathtt{in}\, P$	data reception
	$\mid k \lhd l; P$	label selection
	$\mid k \rhd \{l_i : P_i\}_{i \in \{1..n\}}$	label branching
	$\mid \mathsf{throw}\, k[k']; P$	channel sending
	$\mid \mathsf{catch}\, k(k')\ \mathtt{in}\, P$	channel reception
	$\mid \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q$	conditional branch
	$\mid P \mid Q$	parallel composition
	$\mid \mathsf{inact}$	inaction
	$\mid (\boldsymbol{\nu}\, u)(P)$	name/channel hiding
	$\mid \mathsf{def}\, D\, \mathsf{in}\, P$	recursion
	$\mid X[\tilde{e}\tilde{k}]$	process variables
D	$::= \{X_i(\tilde{x}_i\tilde{k}_i) = P_i\}_{i \in \{1..n\}}$	declaration for recursion

Figure 2.1.: Session Calculus: Syntax.

Basic Primitives Processes exchange messages by pairs of *data sending* and *data reception* actions. Note that send ($k![e]$) uses sequential composition (;) to connect with its continuation P and the syntax of receive ($k?(x)\ \mathtt{in}\, P$) includes direct continuation. Along with *inaction*, *parallel composition* and *name/channel hiding*, these basic constructs can be directly translated to asynchronous π-calculus.

Branching and Selection *Label branching* and *label selection* are features in session calculus for structured external choice. The constructs represent conditional statements for communication, such that depending on the condition (usually runtime expressions), sessions can exhibit different interaction behaviours, in addition to simple serial communication.

The choice of branch is communicated by sending and receiving of a label l, which matches l_i on the receiver. Both the sender and the receiver will continue the process with the chosen branch (P_i and P). Label branching and selection differs from *conditional branch* which is internal to the process and the choice of branch is not sent to another process.

Iterations Iteration in session calculus is supported by *recursion.* Recursion in the calculus does not involve sending and receiving and thus is internal to the process, unless the body of recursion contains external interactions. This implies that either a process recurses forever (for example, in a server application), or the decision of whether or not to exit a recursion (possible by ending the process with inact) must be explicitly communicated in the body.

Session Delegation The π-calculus is a mobile calculus – by passing names as well as values in messages, π-calculus can pass a channel as a name to another process and the process that receives the name can use it as a channel. The session calculus accommodates the mobility property of π-calculus by *channel sending* and *channel reception*, also called session delegation. By offloading parts of responsibilities of the parent process to subprocesses, processing can be distributed to lower level or smaller processes hence increasing the flexibility of the processes. More importantly, the top-level process does not need to be informed about the delegation which allows a higher-order view when designing distributed processes.

Session Calculus Example

As an example of session calculus, Figure 2.2 is a definition of a simple sum server system that adds and returns the sum of two numbers in *SumServer* supplied by *SumClient*.

$$
\begin{aligned}
SumServer &= \mathsf{accept}\, a(k)\ \mathtt{in}\, k?(\tilde{x})\ \mathtt{in}\, k?(\tilde{y})\ \mathtt{in}\, k![x+y]; \mathsf{inact} \\
SumClient &= (\boldsymbol{\nu}\, k)(\mathsf{request}\, a(k)\ \mathtt{in}\, k![42]; k![77]; k?(result)\ \mathtt{in}\, \mathsf{inact}) \\
SumSystem &= (\boldsymbol{\nu}\, a)(\text{SumClient} \mid \text{SumServer})
\end{aligned}
$$

Figure 2.2.: Session Calculus: *SumServer* Example.

2.1.2. Operational Semantics of the Session Calculus

Operational semantics of the calculus is defined by reduction rules. The reduction rules define how processes, when composed together, can proceed by inspecting the action prefixes of each process. A correct reduction is one that all of the processes are reduced to the terminal state, i.e. inact. Figure 2.4 details the structural congruence (for transforming processes in a canonicalised structure) and the reduction rules of the Session Calculus.

Below we define the structural congruence relation $\equiv$, which is the smallest congruence relation on processes. $\equiv_\alpha$ is the standard alpha equality, and $\mathrm{fn}(P)$, $\mathrm{fc}(P)$, $\mathrm{fv}(P)$, $\mathrm{fpv}(P)$ stands for sets of free *names*, *channels*, *variables* and *process variables* of process P respectively.

$P \equiv Q \text{ if } P \equiv_\alpha Q$	α equality
$P \mid \mathsf{inact} \equiv P$	Inaction
$P \mid Q \equiv Q \mid P$	Commutative
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	Associative
$(\boldsymbol{\nu}\, u)(\mathsf{inact}) \equiv \mathsf{inact}$	Unused name
$(\boldsymbol{\nu}\, u\ u)(P) \equiv (\boldsymbol{\nu}\, u)(P)$	Duplicated name
$(\boldsymbol{\nu}\, u\ u')(P) \equiv (\boldsymbol{\nu}\, u'\ u)(P)$	Ordering
$(\boldsymbol{\nu}\, u)(P) \mid Q \equiv (\boldsymbol{\nu}\, u)(P \mid Q)$ if $u \notin \mathrm{fc}(Q) \cup \mathrm{fn}(Q)$	Res. moved to outmost
$(\boldsymbol{\nu}\, u)(\mathsf{def}\, D \,\mathsf{in}\, P) \equiv \mathsf{def}\, D \,\mathsf{in}\, (\boldsymbol{\nu}\, u)(P)$ if $u \notin \mathrm{fc}(D) \cup \mathrm{fn}(D)$	
$(\mathsf{def}\, D \,\mathsf{in}\, P) \mid Q \equiv \mathsf{def}\, D \,\mathsf{in}\, (P \mid Q)$ if $\mathrm{fpv}(D) \cap \mathrm{fpv}(Q) = \emptyset$	Combine recur. processes
$\mathsf{def}\, D \,\mathsf{in}\, (\mathsf{def}\, D' \,\mathsf{in}\, P) \equiv \mathsf{def}\, D \text{ and } D' \,\mathsf{in}\, P$ if $\mathrm{fpv}(D) \cap \mathrm{fpv}(D') = \emptyset$	

Figure 2.3.: Session Calculus: Structural Congruence.

2.1.3. Duality and Session Types

The session calculus defines a way to describe structured interactions. However, it does not guarantee if the interactions between two processes are compatible or are free of deadlocks. Sessions are compatible if and only

The operational semantics are given by the reduction ($\longrightarrow$) relation, defined by the following rules. $\downarrow$ is the standard *evaluation relation*.

$$
\begin{array}{cr}
\mathsf{accept}\, a(k)\, \mathtt{in}\, P_1 \mid \mathsf{request}\, a(k)\, \mathtt{in}\, P_2 \longrightarrow (\boldsymbol{\nu}\, k)(P_1 \mid P_2) & [\textsc{Link}] \\
k![\tilde{e}]; P_1 \mid k?(\tilde{x})\, \mathtt{in}\, P_2 \longrightarrow P_1 \mid P_2[\tilde{c}/\tilde{x}] \quad (\tilde{e} \downarrow \tilde{c}) & [\textsc{Com}] \\
k \lhd l_i; P \mid k \rhd \{l_i\colon P_i\}_{i\in\{1..n\}} \longrightarrow P \mid P_i \quad (1 \leq i \leq n) & [\textsc{Lbl}] \\
\mathsf{throw}\, k[k']; P_1 \mid \mathsf{catch}\, k(k')\, \mathtt{in}\, P_2 \longrightarrow P_1 \mid P_2 & [\textsc{Pass}] \\
\mathsf{if}\, e\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2 \longrightarrow P_1 \quad (e \downarrow \mathtt{t}rue) & [\textsc{If1}] \\
\mathsf{if}\, e\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2 \longrightarrow P_2 \quad (e \downarrow \mathtt{f}alse) & [\textsc{If2}] \\
\mathsf{def}\, D\, \mathsf{in}\, (X[\tilde{e}\tilde{k}] \mid Q) \longrightarrow \mathsf{def}\, D\, \mathsf{in}\, (P[\tilde{c}/\tilde{x}] \mid Q) & [\textsc{Def}] \\
\tilde{e} \downarrow \tilde{c}, X(\tilde{x}\tilde{k}) = P \in D & \\
P \longrightarrow P' \Rightarrow (\boldsymbol{\nu}\, u)(P) \longrightarrow (\boldsymbol{\nu}\, u)(P') & [\textsc{Scop}] \\
P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q & [\textsc{Par}] \\
P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q' \equiv Q \Rightarrow P \longrightarrow Q & [\textsc{Str}]
\end{array}
$$

Figure 2.4.: Session Calculus: Reduction Rules.

if the same channel of the two interacting processes are "associated with complementary behaviours" [HVK98, Definition 5.2]. This is an important requirement that provides the theoretical basis for communication safe processes.

Actions	Dual Actions
$\mathsf{accept}\, a(k)\ \mathtt{in}\, P$	$\mathsf{request}\, a(k)\, \mathtt{in}\, P$
$k![e]\ ; P$	$k?(x)\ \mathtt{in}\, P$
$k \rhd \{l_i\colon P_i\}_{i\in\{1..n\}}$	$k \lhd l\ ; P$
$\mathsf{throw}\, k[k']\ ; P$	$\mathsf{catch}\, k(k')\ \mathtt{in}\, P$
inact	inact

Table 2.1.: Dual Actions in the Session Calculus.

In the syntax given in Figure 2.1, complementary (*dual*) actions are listed in Figure 2.1; these pairs of actions, when composed together, will reduce following the reduction rules. To determine if the actions are dual, we inspect the types of the calculus to obtain a high-level type abstraction, viz. **Session Types**.

$$
\begin{array}{rcll}
\text{Sort}S & ::= & \mathbf{nat} \ \mid \ \mathbf{bool} & \\
 & & \mid \langle \alpha, \overline{\alpha} \rangle & \text{complement} \\
\text{Type}\alpha & ::= & \downarrow[\tilde{S}];\alpha & \text{output} \\
 & & \mid \ \uparrow[\tilde{S}];\alpha & \text{input} \\
 & & \mid \ \oplus\{l_i : \alpha_i\}_{i\in\{1..n\}} & \text{selection} \\
 & & \mid \ \&\{l_i : \alpha_i\}_{i\in\{1..n\}} & \text{branching} \\
 & & \mid \ \downarrow[\alpha];\beta & \text{channel output} \\
 & & \mid \ \uparrow[\alpha];\beta & \text{channel input} \\
 & & \mid \mu\,\mathbf{t}.\alpha \ \mid \ \mathbf{t} & \text{recursion} \\
 & & \mid \mathtt{1} & \text{inaction} \\
 & & \mid \ \perp & \text{bottom}
\end{array}
$$

Figure 2.5.: Session Type: Syntax.

The session type syntax is given in Figure 2.5, where t, t', ... are *type variables*; S, S', ... are *sorts*; α, β, ... are *set of types.*

$\langle \alpha, \overline{\alpha} \rangle$ is a sort that represents complementary interaction structures. $\downarrow[\tilde{S}]$ $;\alpha$ represents output of sorts $\tilde{S}$ then does the action of type α; its dual is $\uparrow[\tilde{S}]$ $;\alpha$ which represents input of sorts $\tilde{S}$ followed by action of type α. $\oplus\{l_i : \alpha_i\}_{i\in\{1..n\}}$ represents selection, where one of l_i is selected out of n choices, then perform the behaviour of the corresponding α_i (external choice); $\&\{l_i : \alpha_i\}_{i\in\{1..n\}}$ represents branching behaviour where it waits for a label l_i to be selected, then behaves as the corresponding α_i (internal choice). $\downarrow[\alpha]$ $;\beta$ represents channel output, a higher-order output that sends channel instead of names, and its dual is $\uparrow[\alpha]$ $;\beta$, representing channel input. $\mu\,\mathbf{t}.\alpha$ represents recursion, and when $\mathbf{t}$ is encountered recurs to the behaviour of α. $\mathtt{1}$ represents inaction, and $\perp$ represents no further interactions are possible. Given the session types of interacting processes (i.e. one on each side of the interaction for binary sessions), we use a type system to catch potential communication errors in the interactions. The session type system consists of typing rules that the sessions must conform to, and these rules have a similar notion of duality in the *types*, where $\overline{\alpha}$ is dual type of α, and is defined in full in Table 2.2.

Based on the duality of types, a type discipline, namely, Session Types is

Types	=	Dual Types
$\overline{\downarrow[\tilde{S}];\alpha}$	=	$\uparrow[\tilde{S}];\overline{\alpha}$
$\overline{\&\{l_i : \alpha_i\}_{i\in\{1..n\}}}$	=	$\oplus\{l_i : \overline{\alpha}_i\}_{i\in\{1..n\}}$
$\overline{\downarrow[\alpha];\beta}$	=	$\uparrow[\alpha];\overline{\beta}$
$\overline{1}$	=	1
$\overline{\perp}$	=	$\perp$

Table 2.2.: Duality in Session Types.

presented in [HVK98, Section 5]. Processes that can be derived from the axioms in the typing system without getting stuck following the typing rules from top to bottom are *typable*. Typable processes in the typing system are guaranteed *communication correctness* property, i.e. complementary interaction behaviours in the common channel, and *deadlock freedom* property, i.e. absence of communication deadlocks in the interacting processes, the typing system is also sound; A sound typing system will not cause stuck errors if the implementation is correct. An extension of the type system presented in [HVK98] is given in Chapter 7.

2.1.4. Multiparty Session Types

Session Types introduced in the last section describe an approach for safe communication between two participants. When a communication involves more than two participants, the correctness of interactions between any two participants can be guaranteed by binary session types. However, binary sessions cannot prevent interleaving of sessions in a communication with multiple binary sessions. Interleaving sessions might introduce incorrect communication logic or cause communication deadlocks between sessions and does not eliminate deadlocks in communication in a more general setting.

This will especially be a problem when multiple binary sessions are implemented by different parties, and the participants are not given any global coordination. The end result may be correct interactions in each of the local perspectives but incorrect in the global perspective. Recent works from Deniélou and Yoshida [DY12, DY13] attempt to resolve this from the bottom up – by constructing (or synthesising) a global session type using communicating automata, where the ultimate aim is to infer a global view from binary sessions. Chapter 7 describes a compositional approach with similar aims in mind. A major extension of session types called Multiparty Session Types (MPST) [HYC08] is introduced by Honda, Yoshida and Carbone in 2008 to tackle these scenarios. MPST make no assumption on the order of interactions between multiple participants, and introduces *global type* as an explicit specification for the overall interactions between multiple participants within a *multiparty session*. The global type is then converted automatically into *local types* by a *projection algorithm* for each of the participants. Local types, sometimes called endpoint types, are generalised session types for each of the participants/endpoints, and are similar to ordinary session types, but for more than one interaction opponent. The framework of MPST is shown in Figure 2.6. Note that the implementations do not use the global types directly, but follow the local types as specifications.

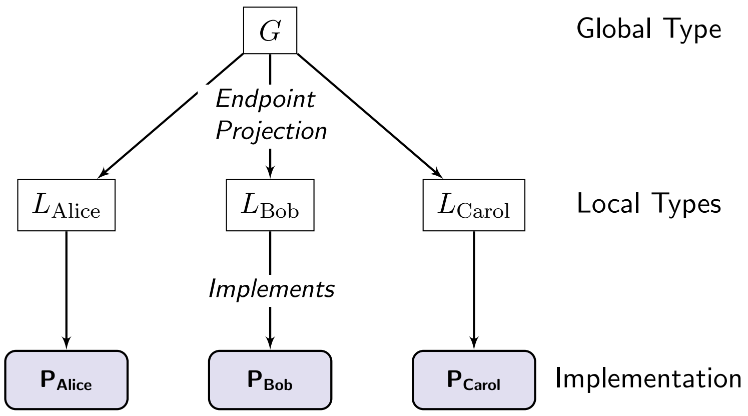


Figure 2.6.: Multiparty Session Types Framework.

Multiparty Session Calculus and Types

Multiparty Session Types are defined in terms of a multiparty session calculus and reduction rules as their operational semantics. We show the multiparty session calculus from [HYC08] describing global interactions in Figure 2.7. The calculus and the reduction rules are very similar to the ones presented in the previous section for session types (actions are prepended with interacting participant): *Multicast session request* is the multiparty version of session request. *Value/session sending/receiving* are prefixed with the session name s. The rest of the multiparty session calculus syntax are same as the ones given in Figure 2.1. *Message queue* are introduced for explicit asynchronous communication. The global and local types syntax in Figure 2.8. The types are governed by typing rules ensuring typable programs are communication safe.

$$
\begin{array}{rll}
P & ::= \overline{a}_{[2..n]}(s).P & \text{multicast session request} \\
 & \mid a_{[p]}(s).P & \text{session acceptance} \\
 & \mid s!\langle e\rangle; P & \text{value sending} \\
 & \mid s?(x); P & \text{value reception} \\
 & \mid s!\langle\!\langle \tilde{s}\rangle\!\rangle; P & \text{session delegation} \\
 & \mid s?((\tilde{s})); P & \text{session delegation} \\
 & \mid s \lhd l; P & \text{label selection} \\
 & \mid s \rhd \{l_i :\ P_i\}_{i\in I} & \text{label branching} \\
 & \mid \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q & \text{conditional branch} \\
 & \mid P \mid Q & \text{parallel composition} \\
 & \mid \mathsf{0} & \text{inaction} \\
 & \mid (\boldsymbol{\nu}\, n)(P) & \text{hiding} \\
 & \mid \mathsf{def}\, D\, \mathsf{in}\, P & \text{recursion} \\
 & \mid X\langle \tilde{e}\tilde{s}\rangle & \text{process call} \\
 & \mid s : \tilde{h} & \text{message queue} \\
e & ::= v \mid e\ \mathsf{and}\ e' \mid \mathsf{not}\ e\ \ldots & \text{expressions} \\
v & ::= a \mid \mathsf{true} \mid \mathsf{false} & \text{values} \\
h & ::= l \mid \tilde{v} \mid \tilde{s} & \text{messages-in-transit} \\
D & ::= \{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I} & \text{declaration for recursion}
\end{array}
$$

Figure 2.7.: Multiparty Session Calculus: Syntax.

2.1.5. Parameterised Multiparty Session Types

Many multiparty interactions, especially those based on parallel algorithms, often contain a number of very similar substructures. The number of such substructures are not known at design time, but are instantiated by parameters supplied at execution time. Examples are interactions between an arbitrary number of buyers and sellers in a financial negotiation or parallel algorithms that can be segmented to a variable number of nodes such as

$$
\begin{array}{rll}
\text{Value } U & ::= \tilde{S} \;\mid\; T@\mathsf{p} & \\
\text{Sort } S & ::= \mathsf{bool} \;\mid\; \mathsf{nat} \;\mid\; \ldots \;\mid\; \langle G \rangle & \\
\text{Global } G & ::= p \to p' \colon k\langle U \rangle.G & \text{values} \\
& \mid p \to p' \colon k\{l_j \colon G_j\}_{j \in J} & \text{branching} \\
& \mid G, G' & \text{branching} \\
& \mid \mu\,\mathbf{t}.G & \text{recursive} \\
& \mid \mathbf{t} & \text{variable} \\
& \mid \mathsf{end} & \text{end} \\
\text{Local } T & ::= k!\langle U \rangle; T & \text{output} \\
& \mid k?\langle U \rangle; T & \text{input} \\
& \mid k \oplus \{l_i \colon T_i\}_{i \in \{1..n\}} & \text{selection} \\
& \mid k \,\&\, \{l_i \colon T_i\}_{i \in \{1..n\}} & \text{branching} \\
& \mid \mu\,\mathbf{t}.T \;\mid\; \mathbf{t} & \text{recursion} \\
& \mid \mathsf{end} & \text{inaction}
\end{array}
$$

Figure 2.8.: Multiparty Session Types: Type Syntax.

implementations of the Fast Fourier Transformation (FFT) algorithm. Parameterised Multiparty Session Types [DYBH12a, DYBH12b] is an extension of standard MPST for this category of interactions.

Parameterised MPST adds integer *indices* to roles of a multiparty session, and extends the global type with *primitive recursion operator* from Gödel's System $\mathcal{T}$ to iterate through the indices. The syntax of primitive recursion is given by the global type

$$\mathbf{R}\,G\,\lambda i\!:\!I.\lambda\mathbf{x}.G'\,\mathtt{i}$$

which takes three parameters, a global type G, a recursion body and an index $\mathtt{i}$. The recursion body has an index variable i with range I (restricted set of natural numbers), a type variable for recursion $\mathbf{x}$ and a recursion body G'.

Figure 2.9 shows the reduction rules for primitive recursion in the global types when applied to an index $\mathtt{i}$. The first rule is the base case where global type G is used when the value of index `i` reaches 0, and the second rule corresponds to repeating the body G' and reducing the index `i` by 1 in each iteration, and the original value of `i` is $n{+}1$.

$$
\begin{array}{lcl}
\mathbf{R}\, G\, \lambda i\!:\!I.\lambda \mathbf{x}.G' \;\; 0 & \longrightarrow & G \\
\mathbf{R}\, G\, \lambda i\!:\!I.\lambda \mathbf{x}.G' \;\; (\mathrm{n}{+}1) & \longrightarrow & G'\{\mathrm{n}/i\}\{\mathbf{R}\, G\, \lambda i\!:\!I.\lambda \mathbf{x}.G'\; \mathrm{n}/\mathbf{x}\}
\end{array}
$$

Figure 2.9.: Global Type Reduction of Parameterised MPST.

For example, suppose G is end, G' is $\mathbf{x}$ and the index is 2. The reduction of the primitive recursion is shown below:

$$
\begin{array}{llcll}
 & \mathbf{R}\,\mathsf{end}\,\lambda i\!:\!I.\lambda \mathbf{x}.\mathbf{x}\;\; 2 & \longrightarrow & \mathbf{x}\;\{1/\mathrm{i}\}\{\mathbf{R}\,\mathsf{end}\,\lambda i\!:\!I.\lambda \mathbf{x}.\mathbf{x}\;\; 1/\mathbf{x}\;\} & \text{Rule 2} \\
\longrightarrow & \mathbf{R}\,\mathsf{end}\,\lambda i\!:\!I.\lambda \mathbf{x}.\mathbf{x}\;\; 1 & \longrightarrow & \mathbf{x}\;\{0/\mathrm{i}\}\{\mathbf{R}\,\mathsf{end}\,\lambda i\!:\!I.\lambda \mathbf{x}.\mathbf{x}\;\; 0/\mathbf{x}\;\} & \text{Rule 2} \\
\longrightarrow & \mathbf{R}\,\mathsf{end}\,\lambda i\!:\!I.\lambda \mathbf{x}.\mathbf{x}\;\; 0 & \longrightarrow & \mathsf{end} & \text{Rule 1}
\end{array}
$$

2.1.6. Subtyping and Asynchronous Subtyping

A *subtyping relation* in a type discipline refers to the relation between type T and subtype S, where all values representable by S are a subset of all values representable by T. For example, in the context of ordinary datatypes, int (natural numbers) is a subtype of real (real numbers), where int is a more refined or specialised type than real, such that any operations applied to real can be applied to int safely without violations.

There are two main branches of subtyping work in session types. The first branch of session subtyping extends session types with *datatype subtyping*, first studied by Gay and Hole [GH05] then by Gay [Gay08] and more recently by Padovani [Pad13]. Session compatibility between two interacting processes is no longer limited to identical message datatypes (e.g. send int and receive int), but also their subtypes. Given the following session types (in the syntax of [GH05]):

$$S = ![\mathsf{int}];\mathsf{end} \qquad\qquad T = ?[\mathsf{real}];\mathsf{end}$$

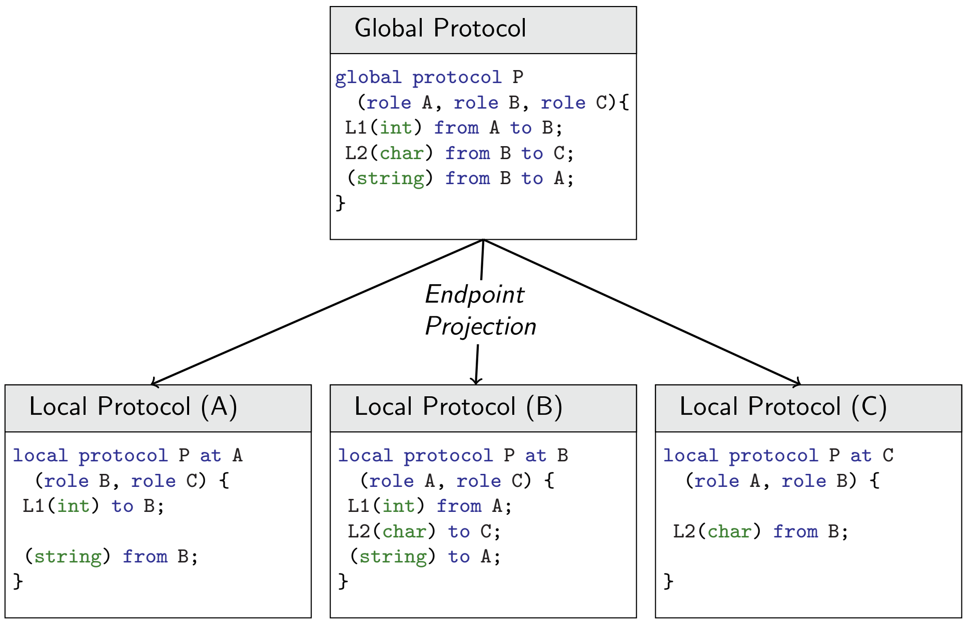
S and T are dual as long as int is a subtype of real ($\mathsf{int} \leqslant \mathsf{real}$), despite the message types int and real are not strictly identical.

The other branch of session subtyping is structural, and is specific to session types. *Asynchronous subtyping* [MYH09, Mos09] by Mostrous, Yoshida and Honda characterises compatibility between classes of permutations of interactions within asynchronous multiparty sessions. The work considers the conditions for a session to remain safe when changing the orders of non-blocking sends and receives are incompatible if modelled in a synchronous session.

There are many scenarios where asynchronous subtyping is applicable in developing practical distributed applications, and the theory is particularly useful in parallel programming. For instance, overlapping communication and computation is amongst the fundamental optimisation techniques involving communication in parallel programming. Implementations initiate non-blocking communication early, so that computation can execute in the foreground while the data being sent or received continue in the background, reducing the stall time for data transmission.

Later in this thesis (Chapter 3) we present an implementation of a type checker, incorporating asynchronous subtyping as a local refinement for safe parallel programming. We will give a more detailed account on how the theory is interpreted in practice.

2.1.7. Scribble Protocol Description Language



```
global protocol P
  (role A, role B, role C){
 L1(int) from A to B;
 L2(char) from B to C;
 (string) from B to A;
}
```

```
local protocol P at A
  (role B, role C) {
 L1(int) to B;

 (string) from B;
}
```

```
local protocol P at B
  (role A, role C) {
 L1(int) from A;
 L2(char) to C;
 (string) to A;
}
```

```
local protocol P at C
  (role A, role B) {

 L2(char) from B;

}
```

Figure 2.10.: The Scribble Protocol Description Language.

Scribble [HMB$^+$11, Scr] is a protocol description language with its roots in Multiparty Session Types. Scribble protocols describe application-level communication protocols, and it follows the framework of MPST. The user specifies the global protocol and local protocols can be derived from the global protocol through endpoint projection. An example is shown in Figure 2.10.

The current version of the Scribble language specification is developed and maintained at Imperial College, with a number of industry stakeholders including RedHat, Ocean Observatories Initiatives (OOI) and Cognizant.

Similar to the session calculus in Section 2.1.1, the core Scribble language also consists of basic primitives for interactions and structuring constructs for conditionals and iterations. We introduce the Scribble language later in Section 3.2.1, which is part of the Session C programming framework we present in Chapter 3.

2.2. Parallel Hardware Architectures and Programming Models

Parallel computations are not just about modelling concurrency and parallelism. In order to perform parallel computations, we need to first formulate problems into algorithmic recipes that can be executed in parallel; Furthermore, we need the machinery that can execute the given recipes in parallel. In this section, we explain different categories of parallel hardware with their corresponding programming models. The aim of this section is to give a general overview of the state of the art of parallel programming, and how our language-based approach to communication safety can relate to these models.

We first look at parallel hardware architectures, since most, if not all, traditional parallel programming models are designed to provide an abstraction of its underlying hardware, rather than custom hardware designed specifically to fit the models.

2.2.1. Superscalar Processors

We begin with parallel hardware architectures of microprocessors. Microprocessors (or simply processors) are the main hardware components of a computer architecture. A processor (or more specifically, the Central Processing Unit) in the Von Neumann architecture takes input instructions stored in memory and carries out the computations specified by the instructions sequentially. In order to get higher performance out of a single processor, microinstructions are pipelined, and additional functional units added to a processor. Multiple instructions can then be issued in a single CPU clock cycle and executed simultaneously. This is called *superscalar execution*, and it exploits instruction-level parallelism in the processor. Figure 2.11 shows the performance improvement in perfect superscalar execution.

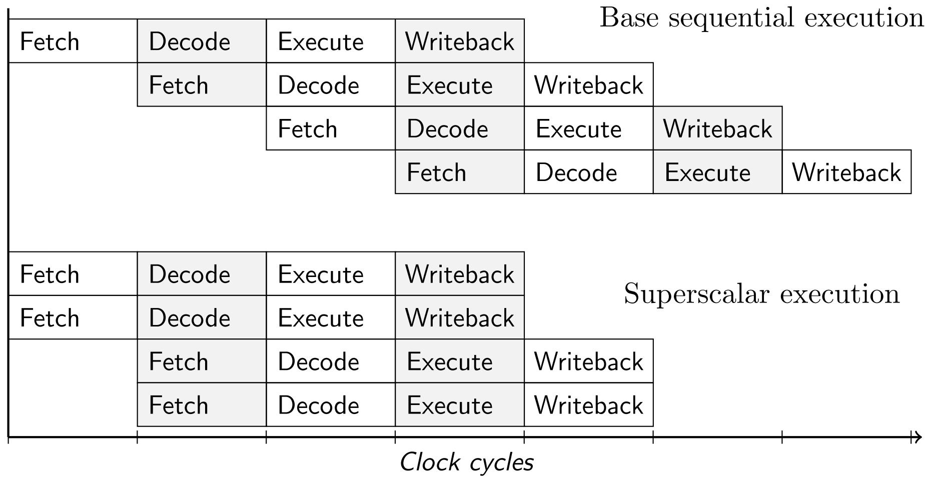


Figure 2.11.: Superscalar execution.

Programming Model: Implicit The parallelism in superscalar execution is implicit, meaning that no special programming models or techniques are needed to convert a sequential program to parallel. Given a stream of sequential instructions, the scheduler in the processor will detect and resolve dependencies in the instructions, so that they can be executed in parallel correctly. Since the sequential programs are not tailored to be executed in parallel, it is hard to predict the performance and the overall improvements are limited compared to parallel programs.

2.2.2. Multicore Processors

As the name suggests, a multi-core processor contains multiple processor *cores*, or independent CPUs, on a single physical chip. In terms of architecture, multicore processors are very similar to multiprocessor systems, where each of the cores can work independent of other cores, and the main difference between the two hardware arrangements is the tighter coupling in a multicore processor. Typical designs of a multicore processor share a local coherent cache memory between the cores, so that there are performance gains if the cores are performing computations that can take advantage of memory locality; whereas multiprocessor systems can only access memory through the system bus, which is much lower level in the computer memory hierarchy and orders of magnitude slower. Figure 2.12 shows the difference in multicore processor and multi-processor architectures.

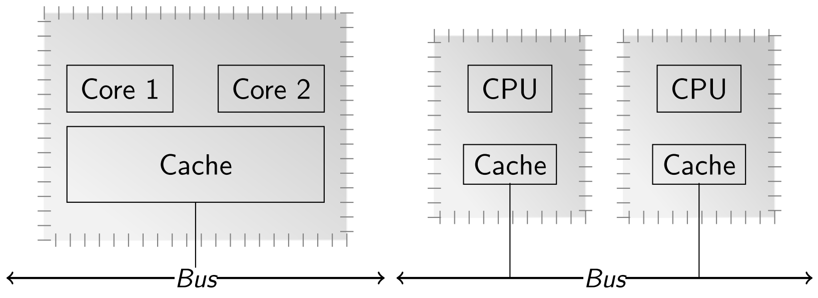


Figure 2.12.: Multicore Processor and Multi-processor Architecture.

Due to the *power wall*, i.e. the reduction of performance due to overheating and high power power consumption, and the physical limits of shrinking single-core processor die size to reduce power consumption, it has become infeasible to further increase performance of single-core processors. Recent trends see the industry has switched to developing processors in a multicore package in order to achieve higher performance with lower power consumption. It is now commonplace to find multicore processors in low power embedded systems such as mobile phones, which shows how necessary multicore processors are.

Programming Model: Shared Memory Despite the advantages from the hardware standpoint, developing software for parallel computer architectures with multicore processors proves to be a huge challenge. Unlike superscalar processors, developing software that can fully utilise the parallel hardware requires clever design. *Shared memory* is one of the main programming models for multicore processors. In the model, each processor core runs an independent lightweight process or *thread* and accesses a shared memory for synchronisation and coordination. To prevent race conditions between the threads, *locks* are used to give threads mutually exclusive access to the shared memory. However, competing for a single lock between multiple threads means that it becomes a scalability bottleneck when the number of threads increases. Moreover, lock-based concurrency is very prone to *deadlocks*, where no progress in the threads are possible. As an alternative to locks, lockless concurrency techniques such as Software Transactional Memory (STM) or Hardware Transactional Memory (HTM) have been the focus of intense research for the shared memory model.

Programming Model: Message-Passing In recent years, research such as Intel's 80-core microprocessor project [MVF08] attempts to investigate multicore processors in a much bigger scale. The hardware design of the 80-core processor lacks the shared coherent cache memory common in conventional multicore processors, and uses Network-on-Chip as the processor's inter-connect architecture. Because of the scalability issues of the shared memory model, *message-passing* was chosen as the programming model for the architecture. Message-passing is a model where messages are sent between independent threads or processes for synchronisation and coordination.

2.2.3. Distributed and Cluster Computing

Distributed computing is not strictly a parallel hardware architecture, but the distributed computing research lays the foundation for cluster computing. A distributed system is composed of multiple networked complete systems. The components of the distributed system interact with each other by passing messages over the network, which could be Ethernet, or even the Internet. We make a distinction between distributed computing and cluster computing for their purposes, computer clusters are purpose-built, tightly-coupled distributed systems for high-performance parallel computation, the components of a cluster are similar or identical high-performance hardware such that it is easier to distribute tasks in massively parallel applications. All of the fastest supercomputers in the world are clusters. Figure 2.13 shows two distributed systems, the switch-connected system has a more uniform connection and is a more likely candidate architecture of a computer cluster.

Programming Model: Message-Passing Message-passing is the most common model for programming most distributed and cluster computing systems. However, novel languages and tools were invented to abstract the underlying architecture further, including the Partitioned Global Address

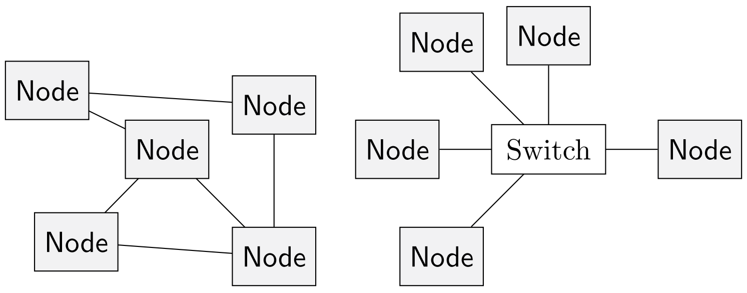


Figure 2.13.: Two examples of distributed systems.

Space (PGAS) parallel programming model, combining the advantages of memory locality in shared memory model and the underlying distributed memory message-passing based model between nodes. We will discuss PGAS in more details in the next section (Section 2.3.3).

Summary Parallel architectures discussed above show that despite the difference in properties and design, the programming models of the parallel hardware are closely related to communication between the components for synchronisation and coordination. It also emerges that message-passing is almost universally applicable to all the architectures. In the next section we look at the concrete state-of-the-art languages and tools for parallel programming.

2.3. State-of-the-art Languages and Tools for Parallel Programming

This section we present some languages and tools for parallel programming, grouped by their programming model. The languages and tools presented are representative of its respective programming model which we compare against session-based programming.

2.3.1. Threads and Locks

The most common shared memory programming tools are threads. Threads are lightweight processes that have a shared address space that can run in parallel. POSIX threads (`Pthreads`) is a standard thread API on UNIX-like systems for threads programming, which gives developers access to very low-level manipulation of threads.

Interleaving of parallel threads that access a same piece of memory can easily lead to race conditions. To prevent race conditions, *locks* are introduced to give processes exclusive access to a single piece of shared state or resource. However, misuse of locks is a source of *deadlocks*, where a group of processes block and wait for each other in a cyclic configuration.

On the other hand, higher level languages and tools exist to simplify threads programming and avoid deadlocks. For example, *Cilk* [FLR98, Ran98] is a research programming language originally developed in MIT as a fork-join model for multi-threaded programming. Cilk also features work-stealing for task scheduling. *Cilk Plus* is a commercial implementation of Cilk maintained by Intel. It uses a combination of pragma-based program transformation and libraries for extracting latent parallelism (shared memory multithreading) from sequential code.

OpenMP is an API that supports multiprocessing programming in C/C++ and Fortran, by extracting latent parallelism in sequential code using compiler directives (`pragmas`) and runtime libraries. The OpenMP runtime libraries manage parallelisation and synchronisation by inspecting pragma annotations in the source code of the sequential application.

These approaches simplify shared memory programming, but have limited flexibility[1] compared to direct thread manipulation.

2.3.2. Message-Passing

Message-passing is a technique for coordinating distributed processes by explicitly sending or receiving messages between them. It is more scalable than shared memory programming models because processes do not compete for a single shared resource, but instead the parallelism is made explicit with messages. Message-Passing Interface (MPI) [The93] is a parallel programming API standard based on message-passing, developed jointly by academia and industry over the past 20 years. It is the de facto standard for parallel programming in scientific computing, and is available for most High Performance Computing systems because of its scalability over large scale parallel systems. MPI is designed to be language independent and portable, and multiple implementations exist, most notably MPICH2 [Gro02] and the

[1]`https://software.intel.com/en-us/articles/choosing-the-right-threading-framework`

open source Open MPI [GFB+04, GSB+06]. The language and hardware independent property of MPI led to research such as [SnPM+10] to program reconfigurable hardware.

However, it is agreed in the HPC community that MPI programming is difficult. MPI encourages a Single Program, Multiple Data (SPMD) programming model, where a single source code defines the behaviour of all the parallel process that it spawns. For a novice MPI user, it is easy to design a parallel application with *communication mismatch*, where messages are sent but never received or vice versa; Moreover, mismatch of communication may lead to *communication deadlocks* where a process blocks indefinitely waiting for a message. These are the top MPI-related errors in parallel applications identified in an Intel survey [DKdS+05], with "send/receive inconsistency" ranked as the top MPI error by MPI users in the survey.

2.3.3. PGAS Languages

Recent research languages for HPC from DARPA's *High Performance Computing Systems* (HPCS) project, such as Chapel [Cha], X10 [CGS+05, LP10, X10] and Fortress [For] share a parallel programming model called Partitioned Global Address Space (PGAS) that share a notion of partitioned globally accessible memory locations, with different access semantics for local and remote partitions. The model applies both to non-uniform memory hierarchies within a node, and to distributed memory locations on clusters. These languages focus on reducing programming complexity for shared memory parallelism through a range of annotations and high-level constructs for coordinating and synchronising thread behaviours. Even though PGAS languages offer a convenient model and primitives for parallel programming, communication deadlocks can still occur as in message-passing programming.

2.3.4. Heterogeneous Computing

Programming with combinations of different types of specialised acceleration hardware such as GPU and FPGA require different programming models. The previous section has covered the programming models for each of the hardware types, i.e. GPU — OpenCL and CUDA, FPGA — VHDL or data-flow programming languages. However, to combine the computation power of the acceleration hardware requires coordination languages such as MPI or session-based languages that can coordinate the PEs to work in unison.

2.3.5. Session-based Languages

Session-based languages are a general category of programming languages and tools that applies the formal typing system of session types as a communication specification.

Session Java (SJ) was introduced in [HYH08] as the first general-purpose session-typed distributed programming language. Another recent extension of SJ added event-based programming primitives [HKP$^+$10], for a different target domain: scalable and type-safe event-driven implementation of applications that feature a large number of concurrent but independent threads (e.g. Web servers). Preliminary experiments with parallel algorithms with SJ were reported in a workshop paper [BHY10]. That early work considered only simple iteration chaining without analysis of deadlock-freedom, and without the general multi-channel primitives required for efficient representation of the complex topologies tackled by *multi-channel SJ* presented in this thesis.

The Bica language [GVR$^+$10] is an extension of Java implementing binary sessions, which allowing session channels to be used as fields in classes. Bica does not support multi-channel primitives and does not guarantee deadlock-freedom across multiple sessions.

[SNZE10] extends SJ-like primitives with multiparty session types and studies type-directed optimisations for the extended language. Their language does not allow sessions to be interleaved within the same process, which is less expressive than that of the formal model originally presented in [HYC08]. Their design is targeted at more loosely-coupled distributed applications than parallel algorithms, where processes are tightly-coupled and typically communicate via high-bandwidth, low-latency media; their optimisations, such as message batching, could increase latency and lower performance.

2.4. Summary

In this chapter we have covered the theoretical basis for communication safe programming – *Session Types*, a type system for modelling and guaranteeing correctness of communication.

We also explored current state of the art of parallel computing, where we looked at programming tools, languages, and verification techniques for parallel programing.

3 The Session C Programming Framework

Overview This chapter presents an efficient programming toolchain for message-passing parallel algorithms which can fully ensure, for any typable programs and for any execution path, deadlock freedom, communication-safety and global progress through static checking. The methodology is embodied as a multiparty session-based programming environment for C and its runtime libraries, which we call Session C. Session C also supports optimisation by asynchronous messaging through session subtyping, which preserves original safety guarantees.

3.1. Introduction

Message-passing applications are difficult to implement correctly. Even if all local calculations are correct, communication deadlocks and communication mismatches may be introduced by the message-passing part of a program. One of the root causes of communication errors is the lack of conformance to an assumed protocol among endpoint programs. Typical examples (written as MPI [MPI] commands) are circular waits such as:

```
Process 1:   MPI_Recv(from=2); MPI_Send(to=3)
Process 2:   MPI_Recv(from=3); MPI_Send(to=1)
Process 3:   MPI_Recv(from=1); MPI_Send(to=2)
```

or communication mismatches such as:

```
Process 1:  MPI_Send(to=2)
Process 2:  MPI_Recv(from=3)
Process 3:  MPI_Send(to=1)
```

To avoid such deadlocks, one might permute the order of messages using asynchronous sending such as `MPI_Isend` followed by `MPI_Recv`, but it is often forgotten to write a required synchronisation (`MPI_Wait`). These are simple errors often illustrated in the textbooks [GLS99, GKKG03], which still appear in many programs including large scale MPI applications, e.g [Par]. Such communication errors are often hard to detect except by runtime analysis. Even if detected, the bugs are hard to locate and fix because they come from distributed processes. Testing in general does not offer full safety assurance as it relies on executing a particular sequence of events and actions.

This chapter proposes a new programming framework for message-passing parallel algorithms centring on explicit, formal description of *global protocols*, and examines its effectiveness through an implementation of a toolchain for C. All validations in the toolchain are done statically and are efficient, with a polynomial-time bound with respect to the size of the program and global protocol. The framework is based on the theory of Multiparty Session Types [HYC08, BCD$^+$08, DY12], and it supports a full guarantee of deadlock-freedom, type-safety, communication-safety and global progress for any statically well-typed programs. Global protocols serve as a guidance for a programmer to write safe programs, representing a type abstraction of expressive communication structures (such as sequencing, choice, broadcast, synchronisation and recursion). The toolchain uses a language Scribble [Scr, HMB$^+$11] for describing the Multiparty Session Types in a Java-like syntax.

Simple

```
protocol Simple(role P1, role P2, role P3) {
  int   from P1 to P2;
  char  from P3 to P1;
  float from P2 to P3;
}
```

A simple example of a protocol in Scribble which corrects the first erroneous MPI program (a wait cycle) is given on the left. For endpoint code development, the programmer uses the *endpoint protocol* generated by the projection algorithm in the toolchain. For example, the above global protocol is projected to P2 to obtain:

```
Simple @ P2
protocol Simple at P2(role P1, role P3) {
  int from P1;
  float to P3;
}
```

which gives a template for developing a safe code for P2 as well as a basis of static verification. Since we start from a correct protocol, if endpoint programs conform to the induced endpoint protocols, it automatically ensures deadlock-free, well-matched interactions.

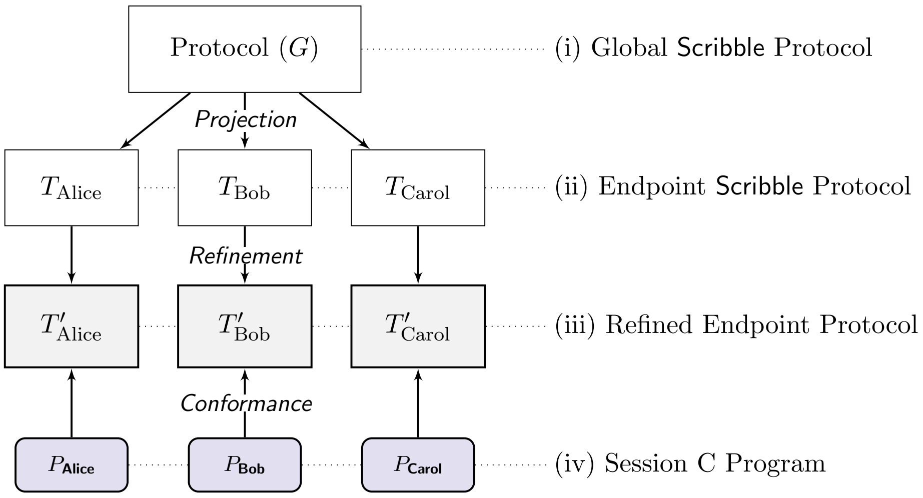


Figure 3.1.: Session C Programming Framework.

Overview of the Toolchain A Session C program is developed in a top-down approach. Figure 3.1 shows the relationships between the four layers (i–iv) that make up a complete Session C program. A Session C programmer first designs a global protocol (i) using Scribble (explained in Section 3.2.1). A Session C program is a collection of individual programs (iv) in which each of the programs implements a participant (called *endpoint*) of the communication. We first extract the endpoint protocol from the global protocol by *projection* (ii). The projection takes the global protocol G and an endpoint (say Alice), and extracts only the interaction that involves Alice (T_{Alice}). Step (iii) describes a key element of our toolchain, the *protocol refinement*. T'_{Alice} is an endpoint protocol refined from the original T_{Alice}.

This allows the programmer to write a more refined program P_{Alice} (which conforms to T'_{Alice}) than a program following the original T_{Alice}. Session C supports the *asynchronous message optimisation* [Mos09, MYH09], the reordering of messages for minimising waiting time as a refinement, Session C implements this optimisation in its subtyping checker (Section 3.3.2). Once P_{Alice} conforms to T'_{Alice} such that $T'_{\text{Alice}} < T_{\text{Alice}}$ (T'_{Alice} is more refined), then P_{Alice} automatically enjoys safety and progress in its interactions with P_{Bob} and P_{Carol}.

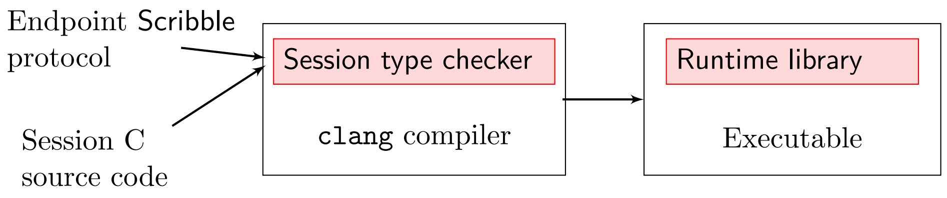


Figure 3.2.: Session C workflow.

Programming environment The programming environment of Session C is made up of two main components, the *session type checker* and the *runtime library* (Section 3.2.2). Figure 3.2 shows the workflow. The session type checker takes an endpoint protocol (T_{Alice}) and a source code P_{Alice} as an input from the user. The endpoint protocol is generated from the global protocol G through the projection algorithm. The session type checker validates the source code against its endpoint protocol. When the program is optimised, it generates T'_{Alice} from P_{Alice} and checks if $T'_{\text{Alice}} < T_{\text{Alice}}$ (Section 3.3.2). The API provides a simple but expressive interface for session-based communications programming.

The contributions of this chapter includes the following:

- A toolchain for developing and executing message-passing parallel algorithms based on a formal and explicit description of interaction protocols (Section 3.2.1), with an automatic safety guarantee. All algorithms used in the toolchain are polynomial-time bounded (Section 3.3.2).

- The first multiparty session-based programming environment for a low-level language, Session C, built from expressive session constructs supporting collective operations (Section 3.2), together with the associated runtime library.

- A session type checker for Session C, which is the first to offer automatic, formal assurance of communication deadlock-freedom (i.e. for any possible *static* control path and interleaving except for limitations outlined in Section 3.2.1) for a large class of message-passing parallel programs (Section 3.3.1), supporting messaging optimisations through the incorporation of the asynchronous subtyping [Mos09, MYH09] (Section 3.3.2).

- The validation of our methodology through the implementations of typical message-passing parallel algorithms, leading to concise and clear programs (Section 3.4). The benchmark results show that representative parallel algorithms in Session C are executed competitively against their counterparts in MPI (the overhead is on average 1%) (Section 3.5).

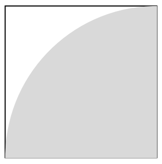
All source code is available on [Ses].

3.2. Protocols and Programming in Session C

3.2.1. The Scribble Protocol Description Language

Our toolchain uses Scribble [HMB+11, Scr], a developer-friendly notation for specifying application-level communication protocols based on the theory of multiparty session types [HYC08, BCD+08, DY12]. Scribble's development tool [Scr, HMB+11] supports parsing, well-formedness checking and endpoint projection, with bindings to multiple programming languages. We briefly introduce its syntax.

MonteCarloPi
```
import int;
protocol MonteCarloPi(role Master, role Worker0, role Worker1) {
  // number of simulations to do in each worker
  int from Master to Others; // broadcast
  rec LOOP {
    from Others to Master { Yes: No: }; // gather
    LOOP;
  }
}
```



The above listing shows a simple Scribble global protocol for Monte Carlo π estimation. The algorithm uses random sampling to estimate the value of π. A Scribble protocol begins with the preamble, in Line 1, consisting of a message type declaration after the keyword `import`. Then the protocol definition is given starting from, in Line 2, the keyword `protocol`, followed by the protocol name, `MonteCarloPi`, and its parameters which are the roles to be played by participants. In addition to the explicitly declared roles, the protocol body can use two special roles – `Others` (for "all other roles") and `All` (for "every role"), both roles are used for collective operations which we will explain below. After the declarations, the protocol body which describes the conversation structure follows.

Line 4 says that the `Master` should send an integer (which specifies the number of tries) to `Others`, i.e. to all other roles than `Master`, i.e. to the workers. Line 5 declares a recursion named `loop`. In Line 6, (after each worker locally generates a random point on a square and tests if the point is in the quarter of a circle, i.e. the shaded area in the right figure above. `Master` is informed by `Others` (workers) whether the test was a hit, by choosing `Yes` or `No`. Regardless, Line 7 recurs.

The description of interaction in Lines 4-8 is generic, catering for any number of workers. Here we use *collective roles* in Scribble, where a single role can denote multiple participants. As explained above, we have two collective roles `Others` and `All` which refer to "all other roles" and "every role" respectively. Using them, we can accurately represent the protocols for MPI collective operations as:

- `MPI_Bcast` (broadcast) from `A`: `from A to Others`.
- `MPI_Reduce` to `A`, a gather operation: `from Others to A`.
- `MPI_Barrier` with `A` as a gather point, for which we use consecutive interactions: `from Others to A`; `from A to Others`.
- `MPI_Alltoall`, a scatter-gather operation: `from All to Others`.

These collective roles can be used as a source and/or a target as far as it is not ambiguous (e.g. `from Others to Others`) and it does not induce a self-circular communication (e.g. `from All to All`). Each `All` is macro-expanded for each endpoint when projecting a global protocol, whereas

`Others` is preserved after projection and is linked to programming constructs, as we shall discuss later.

Global protocol	Endpoint protocol
U `from myrole to role1,..,rolen/Others`	U `to role1,..,rolen/Others`
U `from role1,..,myrole,..,rolen/Others to role`	U `to role`
U `from role1,..,rolen/Others to myrole`	U `from role1,..,rolen/Others`
U `from role to role1,..,myrole,..,rolen/Others`	U `from role`
U `from All to Others`	U `to Others;` U `from Others;`
`from myrole to role {` $l_1 : T_1 \cdots l_n : T_n$ `}`	`to role {` $l_1 : T'_1 \cdots l_n : T'_n$ `}`
`from role to myrole {` $l_1 : T_1 \cdots l_n : T_n$ `}`	`from role {` $l_1 : T'_1 \cdots l_n : T'_n$ `}`
`from All to Others {` $l_1 : T_1 \cdots l_n : T_n$ `}`	`to Others {` $l_1 : T'_1 \cdots l_n : T'_n$ `}` `from Others {` $l_1 : T'_1 \cdots l_n : T'_n$ `}`
`repeat from myrole to role {` T `}`	`repeat to role {` T' `}`
`repeat from role to myrole {` T `}`	`repeat from role {` T' `}`
`rec X {` T `}`	`rec X {` T' `}`

Table 3.1.: Projection from global protocol to endpoint protocol of `myrole`.

We present a summary of the Scribble syntax for global and local protocols in above table, which also shows how the former is projected to the latter. In each line, the left-hand side gives a syntax of a global protocol, while the right-hand side gives its projection onto participant `myrole`. U is a payload type; T and T' are global and endpoint types; and l is a label for branching. T and T' can be empty, denoting termination. Line 1 indicates two cases, U `from myrole to role1 ,.., rolen`, which is a multicast from `myrole` to n other roles; and U `from myrole to Others`, which is a multicast from `myrole` to all others. Similarly for Lines 2-4. The right-hand side views the left-hand global interaction from the viewpoint of `myrole`. In Line 5, `from All to Others` means every role sends to the remaining roles. Hence, for `myrole`, it means (1) it is sending to all others, i.e. broadcast; and then (2) receiving from all others, i.e. reduce.

Limitation: loops

We highlight that Scribble protocols represent static communication patterns, and they capture the *structure* of an interaction. Hence there are limitations to what Scribble protocols can guarantee, as with the underlying MPST type theory (but examples using *primitive recursion* is fine). In particular, if a user decides to use `rec Label { ... }` to represent a loop in the global protocol, but supplies different loop conditions to each endpoint implementation using the loop, then communication safety cannot be guaranteed since the mismatched conditions will introduce a communication mismatch when one of the loop bodies terminates. In order to leverage the full safety guarantee offered by Session C, safe loops should be specified using the `repeat` construct, which synchronises loop conditions between endpoints, instead of `rec`. A legitimate use of `rec` is when a loop involves multiple participants, where each participant is involved in a different number of iterations, e.g.:

```
A: for (i=0; i<10; i++){} | B: for (i=0; i<2; i++){}
                          | C: for (i=0; i<8; i++){}
```

This is correct as long as the user ensures that the loop conditions are compatible, since in the general case this information is only available at runtime and cannot be embedded into static, Scribble protocols. Then the user should ensure that the loop conditions are compatible, since in the general case this information is only available at runtime and cannot be embedded into static, Scribble protocols.

```
1  protocol MonteCarloPi at Master
2      (role Worker0, role Worker1){
3    int to Others;
4    rec LOOP {
5      from Worker0, Worker1 { Yes: No: }
6      LOOP;
7    }
8  }
```

```
1  protocol MonteCarloPi at Worker0
2      (role Master, role Worker1) {
3    int from Master;
4    rec LOOP {
5      to Master { Yes: No: }
6      LOOP;
7    }
8  }
```

As a concrete example of projection acting on the whole protocol, the endpoint protocols resulting from the projection of the Monte Carlo simulation example onto `Master` and `Worker0`, respectively, are given above. Note that this protocol uses the `rec` construct and the user must ensure the compatibility of the loop conditions between the `Master` and the `Workers`. Hence on Line 3 the `Master` sends to each `Worker` process the number of tries (such that the number of iterations in `Master` equals the total number of iterations in all `Workers`).

3.2.2. Session C: Programming and Runtime

Session C offers a high-level interface for safe communications programming based on a small collection of primitives from the session type theory. These primitives are supported by a runtime whose implementation currently uses the ØMQ (ZeroMQ) [ZMQ] socket library, which provides efficient messaging over multiple transports including local in/inter-process communication, TCP and PGM (Pragmatic General Multicast).

A Session C program is a C program that calls the session runtime library. All communication runtime primitives must be called directly from the `main` function as we do not perform inter-procedural analysis. Pointers are allowed in the computation code as long as they do not involve communication or the runtime library, which may introduce communication mismatches outside of Session C runtime. The following code implements `Master` whose endpoint protocol is given in the previous subsection.

```c
1  /* Session C implementation for Master */
2  #include <libsess.h>
3  ...
4  int main(int argc, char *argv[])
5  { // variable declaration ...
6    session *s;
7    join_session(&argc, &argv, &s, "MCPi_Master.spr");
8    const role *Worker0 = s->get_role(s, "Worker0");
9    const role *Worker1 = s->get_role(s, "Worker1");
10
11   int count = 100;
12   msend_int(100, _Others(s));
13
14   while (count-- > 0) {
15     switch(inbranch(Worker0, &rcvd))
16         { case Yes: hits++; break; case No: break; }
17     switch(inbranch(Worker1, &rcvd))
18         { case Yes: hits++; break; case No: break; }
19   }
20   printf("Pi: %.5f\n", (4*hits)/(2*100.0));
21   end_session(s);
22 }
```

In the `main` function, `join_session` (Line 7) indicates the start of a session, whose arguments (from the command line arguments `argc` and `argv`) are a session handle of type `session *` and the location of the endpoint Scribble file. `join_session` establishes connections to other participating processes in the session, according to a connection configuration information such as the host/port for each participant, automatically generated from the global protocol. Next, the lookup function `get_role` returns the participant identifier of type `role *`. Then we have a series of session operations such as `send_`*type* or `recv_`*type* (discussed below). Lines 15-18 expand the choice from `Others` in the protocol into individual choices. Finally an `end_session` cleans up the session. Any session operation before `join_session` or after `end_session` is invalid because they do not belong to any session.

Scribble endpoint	Session C runtime interface
`int to Bob;`	`send_int(role *r, int val);`
`string from Bob`	`recv_int(role *r, char *str);`
`int to role1,..,rolen;`	`msend_int(int val, int roles_count,...);`
`string from role1,..,rolen;`	`mrecv_string(char *str, int roles_count,...);`
`int to Others;`	`msend_int(int val, _Others(sess));`
`string from Others/role1,..,rolen;`	`mrecv_string(char *str, _Others(sess));`
`repeat to Bob { ... }`	`while (outwhile(int cond,int roles_cnt,...))` `{ ... }`
`repeat from Bob { ... }`	`while (inwhile(int roles_cnt, ...))` `{ ... }`
`rec X { ... }`	Ordinary `while`-loop or `for`-loop
`to Bob { LABEL0: ... }`	`outbranch(role *r, const int label);`
`from Bob { LABEL0: ... }`	`inbranch(role *r, int *label);`

Table 3.2.: Session C primitives.

Programming Communications in Session C Table 3.2 lists these primitives as well as control primitives we illustrate next, in correspondence with the Scribble protocol construct introduced in the Section 3.2.1. The first six lines are for message passing. Each function name mentions a type explicitly, as in `send_`*datatype*, following MPI and to ensure type-safety under the lack of strong typing in C. We support `char`, `int`, `float`, `double`, `string` (C-string, contiguous NULL-terminated array of `char`), `int_array` (contiguous array of `int`), `float_array` (contiguous array of `float`), and `double_array` (contiguous array of `double`). These types are sufficient for implementing most parallel algorithms; for composite types that are not in the runtime library, the programmer can choose to combine existing primitives, or augment the library with marshalling and unmarshalling of the composite type, to allow type checking.

In Lines 3/4 of the table, `msend` and `mrecv` specify the number of roles (a roles count) of the targets/sources, respectively. Lines 5/6 show how the programmer can specify `Others` in `msend` and `mrecv`: the roles count and roles list are replaced by a macro `_Others(s)` with the session handle as the argument.

Structuring Message Flows: Branching and Iteration Branching (choice) in Session C is declared explicitly by the use of `outbranch` and `inbranch`. Different branches may have different communication behaviours, and the deciding participant needs to inform the other participant which branch is chosen. The passive participant will then react accordingly.

```c
// Alice
if (i>3) {
  outbranch(Bob, BR_LABEL0);
  send_int(Bob, 42);
} else {
  outbranch(Bob, BR_LABEL1);
  recv_int(Bob, &val);
}
```

```c
// Bob
switch (inbranch(Alice, &rcvd_label)) {
  case BR_LABEL0:
    recv_int(Alice, &val);
    break;
  case BR_LABEL1:
    send_int(Alice, 42);
    break;
}
```

Above, the branching is initiated by a call to `outbranch` in the then-block or else-block of an if-statement. On the receiving side of the branch, the program first calls `inbranch` to receive the branch label. A switch-case statement should then be used to run the segment of code which corresponds to the branching label.

For iteration, two methods are provided: *local* and *communicating iterations*. *Local iteration* is a standard statement such as `while`-statements, with session operations occurring inside. *Communicating iteration* is a distributed version of loop, where, at each iteration, the loop condition is computed by the process calling `outwhile` and is communicated to processes calling `inwhile`. This while loop is designed to support multicast, so that a single `outwhile` can control multiple processes. This is useful in a number of iterative parallel algorithms, in which the loop continues until certain conditions (e.g. convergence) are reached and cannot be determined statically.

```c
// Master process (Alice)
while (outwhile(i++<3, 1, Bob))
  recv_int(Bob, &value);
```

```c
// Slave process (Bob)
while (inwhile(1/*no. of roles*/,
     Alice))
  send_int(Alice, 42);
```

Above, Alice issues an `outwhile` with condition `i++<3` which will be evaluated in each iteration. `outwhile` then sends the result of the evaluation (i.e. `1` or `0`) to Bob and also uses that as the local `while` loop condition. Then Bob receives the result of the condition evaluation from Alice by the `inwhile` call, and uses as the local `while` loop condition. Both processes execute the body of the loop, where Bob sends an integer to Alice. This repeats until `i++<3` evaluates to `0`, then both processes exit the while loop.

3.3. Type Checking and Message Optimisation

3.3.1. Session Type Checker

The session type checker for an endpoint program is implemented as a `clang` C compiler plugin. The `clang` compiler is the full-featured C/C++/Objective-C compiler frontend of the Low-Level Virtual Machine (LLVM) project [LA04]. LLVM is a collection of modular and reusable individual libraries for building compiler toolchains. The modular approach of the project allows easy mix-and-match of individual components of a compiler to build source code analysis and transformation tools. Our session type checker is built as such a tool, utilising the parser and various AST-related frontend modules from the `clang` compiler.

Endpoint Type Checking verifies that the source code conforms to the corresponding endpoint protocol in Scribble. The type checker operates by ensuring that the linear usage of the communication primitives conforms to a given Scribble protocol, based on the correspondence between Scribble and Session C constructs given in the table in Section 3.2.2. The following example shows how Scribble statements are matched against Session C communication primitives.

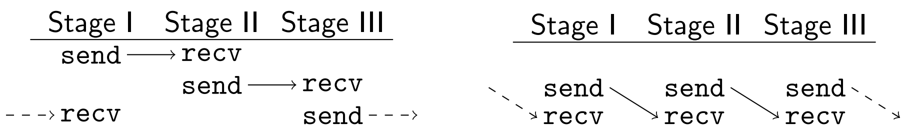
We quickly outline how the type checker works. First, the Scribble endpoint protocol is parsed into an internal tree representation. For brevity, hereafter we refer to it as *session tree*. Except for recursion (which itself is *not* a communication), each node of a session tree consists of (1) the target role, (2) the type of the node (e.g. send, receive, choice, etc.) and (3) the datatype, if relevant (e.g. `int`, `string`, etc.). For example, a Scribble endpoint type statement "`int to Worker;`" becomes a node {`role: Worker, type: send, datatype: int`}.

The type checking is done by *inferring* the session typing of each program and matching the resulting session tree against the one from the endpoint protocol. The type inference is efficiently done by extracting session communication operations from the top-level source code. Body of function calls are not inspected and function pointers are disallowed for role variables. Since C allows unrestricted type conversion by casting, we use the datatype explicitly mentioned in communication functions as the type of an argument rather than the type of its expression. For example, `send_int(Bob, 3.14)` says that sending 3.14 as `int` is the intention of the programmer, which is safe if the receiver is intended to receive an integer. A session tree is then constructed from this session typing. For example, a runtime function call, `send_int(Worker, result)` will be represented by a node {`role: Worker, type: send, datatype: int`}.

We can now move to the final process of session type checking in Session C. After their construction, the session trees from both Scribble endpoint protocol and the program are *normalised*, removing unused dummy nodes, branches without session operations and iteration nodes without children, thus compacting the trees to a canonical form. We then compare these two normalised session trees, and verify that they satisfy the asynchronous subtyping relation (illustrated in Section 3.3.2) up to minimisation, if the Session C implementation conforms to the given Scribble endpoint protocol in the presence of asynchronous message optimisations.

3.3.2. Asynchronous Message Optimisation

This subsection illustrates one of the key contributions of our toolchain, the type checking in the presence of *asynchronous message optimisation.* Parallel programs often make use of parallel pipelines to overlap computation and communication. The overlapping can reduce stall time due to blocking wait in the asynchronous communication model, as far as the overlapping does not interfere with data dependencies.



Above (left) shows a native but immediately safe ring pipeline and (right) an efficient parallel pipeline, which needs only two steps to complete instead of three, since Stage I does not need to wait for data from Stage III. However, this parallel pipeline is hard to type check against a naturally specified global type (which would be based on the left figure where interactions take place one by one), because of the permuted communication operations – we cannot match the `send` against the `recv`, because they criss-cross. But these two figures are equivalent under the asynchronous communication model with non-blocking send and blocking receive.

Naive Stage II

```
while (i++ < N) {
  recv_int(StageI, &rcvd);
  send_int(StageIII, result);
  compute(result);
  result = rcvd;
}
```

Optimised Stage II

```
while (i++ < N) {
  send_int(StageIII, result);
  compute(result);
  recv_int(StageI, &rcvd);
  result = rcvd;
}
```

To see this point concretely, the above listing juxtaposes an unoptimised and optimised implementation of the Stage II. Both programs communicate values correctly despite the different order of communication statements assuming the lack of data dependencies in the computation and communication. Note `compute` is positioned after a send, so that `compute` can be carried out while the data is being sent in the background, taking advantage of non-blocking sends.

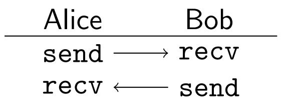
The use of parallel pipelines is omnipresent in message-passing parallel algorithms. To type-check them, we apply the asynchronous subtyping theory [Mos09, MYH09], which allows the following deadlock-free permutations, where send is non-blocking (asynchronous) and receive is blocking (here *channel* refers to the connection between two endpoints):

1. Permuting Receive-Send to Send-Receive in the same or different channels;

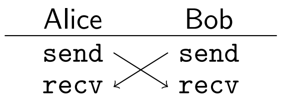
2. Permuting order of Send-Send if they are in different channels;

3. Permuting order of Receive-Receive if they are in different channels

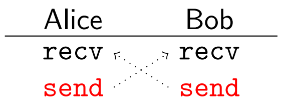
Note that if we permute in the different direction from (1) (i.e. Send-Receive to Receive-Send), it causes a deadlock. e.g. in the efficient pipeline described above, if `send-recv` is permuted to `recv-send` in the Stage I, it causes a deadlock between the Stage I and II. This is because in our asynchronous communication semantics, send is non-blocking and receive is blocking. Below is an example of permutation that will lead to a deadlock:



Original interaction



Permuted Send-Recv



Permuted Recv-Send (deadlock)

The left figure shows the original version of the interaction. Alice and Bob are exchanging data in the same communication channel. If we permute the order of message exchange in Bob to Send-Recv, as in the middle figure, both Alice and Bob can receive after the non-blocking send is issued. However, if we permute the order of message exchange of Alice to Recv-Send as shown in the right figure, both Alice and Bob cannot progress past receive because receive is blocking but neither Alice nor Bob has started sending, thus we have a deadlock.

We give the subtyping rules against Scribble endpoint protocols below, where T is an endpoint type. The type context C defined as:

$$\frac{-}{T < T}\lfloor\textsc{Id}\rfloor$$

$$\frac{T_1 < C[T_2] \quad U' \texttt{ from/to role} \notin C}{U \texttt{ to role;} T_1 < \ C[U \texttt{ to role;} T_2]}\lfloor\textsc{Send}\rfloor$$

$$\frac{T_1 < C[T_2] \quad U' \texttt{ to role} \notin C \quad \forall \texttt{role}'.\, U' \texttt{ from role}' \notin C}{U \texttt{ from role;} T_1 < \ C[U \texttt{ from role;} T_2]}\lfloor\textsc{Recv}\rfloor$$

$$\frac{T_1 < T_2}{\texttt{rec X}\,\{T_1\} < \texttt{rec X}\,\{T_2\}}\lfloor\textsc{Rec}\rfloor$$

$$\frac{T_1 < T_2}{\texttt{repeat from/to role}\,\{T_1\} < \texttt{repeat from/to role}\,\{T_2\}}\lfloor\textsc{Repeat}\rfloor$$

$$C \ ::= \ [] \ \mid \ U \texttt{ from role;} C \ \mid \ U \texttt{ to role;} C$$

The subtyping algorithm in Session C conforms to the rules listed above (which come from [Mos09]) and is their practical refinement, which we describe below:

1. ($\lfloor\textsc{Id}\rfloor$) An endpoint type is a permutation of itself.

2. ($\lfloor\textsc{Recv}\rfloor$) For each receive statement, search for a matching receive for

the same channel in the source code until a receive statement is found or search failed. Send and other statements in different channels can be skipped over.

3. (⌊Send⌋) For each send statement, search for a matching send for the same channel in the source code until a receive statement is found or search failed. Sends can only be permuted between statements in different channels, so overtaking a receive operation is disallowed.

4. (⌊Rec⌋ ⌊Repeat⌋) We apply the permutation described above on consecutive statements within `rec` and `repeat` blocks so the search for matching send/receive is bounded.

Finally, we check that all nodes in the source code and protocol session type trees have been visited.

The algorithm for the subtype checking is outlined below, `check_loop_node` is only executed on loop (`rec` and `repeat`) nodes:

```python
def check_loop_node(protocol, implementation):
  for c in protocol.children:
    allMatching |= findMatching(c, implementation.children):
  return allMatching # Type checking succeeds if matching pairs are found

def findMatching(protocolNode, implNodes):
  found = False
  for implNode in implNodes:
    if implNode.visited: continue

    # No permutation allowed for non send/recv nodes
    if not implNode.isSend and not implNode.isRecv: return False

    if matches(implNode, protocolNode): # Check if node matches specification
      implNode.visited = True
      return True # Node found, done

    # If both nodes are the same 'channel' (i.e. same receiver or same sender)
    if protocolNode.role == implNode.role:

      if protocolNode.isRecv and implNode.isSend:
        continue # OK for Send node to overtake Receive node

      if protocolNode.isSend and implNode.isRecv:
        return False # Instant fail if Receive node overtakes Send node

        return found # Gets to the end without finding a match (default: False)
```

Listing 3.1: Subtype checker algorithm

We end this section by identifying the time-complexity of the present toolchain. It uses well-formedness checking of a global protocol and its projection, which are both polynomial-time bound with respect to the size of the global type [DY10, DY12]. The asynchronous subtype-checker as given above is polynomial against the size of a local type based on the arguments from [MYH09, DY10, Mos09]. Type inference for session typed processes are polynomial [HYC08, MYH09, DY12]. We conclude that the complexity of the whole toolchain is polynomial time-bounded against the size of a global type and a program.

The type inference process goes through the source code in one pass, extracting the endpoint protocol of the implementation. The type checking process compares the endpoint protocol of the implementation and the specification, and performs linearly outside of loops. Within loops, where asynchronous subtyping applies, the subtype checking process steps through each line of the specification protocol, and looks for a matching line in the inferred endpoint protocol in the body of the loop until either a matching line is found or a line is found and does not satisfy the asynchronous subtyping rules above. The subtyping check is bound by $O(M^2)$ where M is proportional to the number of interaction statements in loop bodies. Hence session type checking in Session C is polynomial time-bounded.

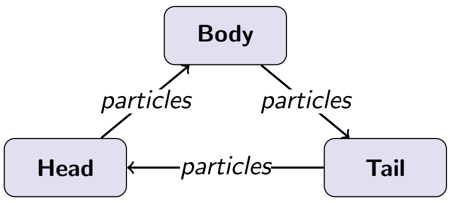
Thus the toolchain is in principle efficient. Furthermore, a careful examination of each algorithm suggests they tend to perform linearly with a small factor in normal cases (e.g. unless deeply nested permutations are carried out for optimisations). Our usage experience confirms this observation.

3.4. Parallel Algorithms in Session C

In this section we demonstrate the effectiveness of Session C for clear, structured and safe message-passing parallel programming, through two algorithms which exemplify complex optimisations and communication topologies. Other representative parallel algorithms [AWW$^+$09, GLS99, Lei91] can be implemented in Session C.

3.4.1. N-body Simulation: Optimised Ring topology

The parallel N-body algorithm forms a circular pipeline. Such a ring topology [N-b] is used in many parallel algorithms such as LU matrix decomposition [CLR08]. The N-body problem involves finding the motion, according to classical mechanics, of a system of particles given their masses, initial positions and velocities. Parallelisation is achieved by partitioning the particle set among a set of m worker processes. Each worker is responsible for a partition of all particles.



Nbody

```
protocol Nbody (role Head, role Body, role Tail) {
  rec NrOfSteps {
    rec SubCompute {
      particles from Head to Body;
      particles from Body to Tail;
      particles from Tail to Head;
      SubCompute;
    }
    NrOfSteps;
  }
}
```

Figure 3.3.: Ring topology of the 3-role `Nbody` protocol.

Figure 3.3 shows the global protocol of N-body simulation with 3 workers, `Head`, `Body` and `Tail`. The simulation is repeated for a number of steps (`rec NrOfSteps`). In each step, the resultant forces of particles held by a worker are computed against all particles held by others. We arrange our workers in a ring pipeline and perform a series of sub-computations (`rec SubCompute`) to propagate the particles to all workers, each involving receiving particles from a neighbouring worker and sending particles received in the previous sub-computation to the next worker.

Nbody @ Body

```
protocol Nbody at Body(role Head, role Tail) {
  rec NrOfIters {
    rec SubCompute {
      particles from Head;
      particles to Tail;
      SubCompute;
    }
    NrOfIters;
  }
}
```

```
while (iterations++ < NR_OF_ITERATIONS) { /* Body implementation */
  while (rounds++ < NR_OF_NODES) {
    send_particles(Tail, tmp_parts);//permuted
    // Update veclocities
    compute_forces(particles, tmp_parts,...);
    recv_particles(Head, &tmp_parts);
  } // Update positions by reeceived velocities
  compute_positions(particles, pvs, ... );
}
```

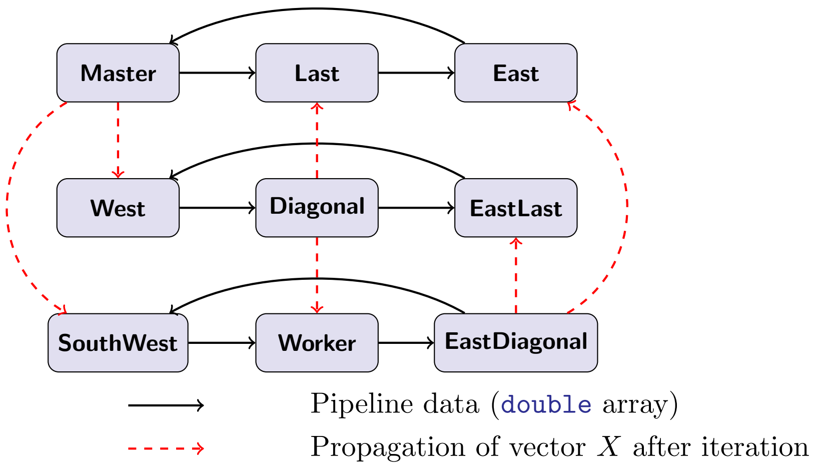
All of the endpoint protocols inherit the two nested `rec` blocks from the global protocol. In the body block of `rec SubCompute`, the order of send and receive are different in `Head` and `Body`. As discussed in Section 3.3.2, Session C allows permuting the order of send and receive for optimisations under the asynchronous subtyping, so that we can type-check this program. Using the endpoint protocols as specification, we can implement the workers. The code above implements the `Body` worker which is typable by our session type checker, despite the difference in order of send and receive from its endpoint Scribble.

3.4.2. Linear Equation Solver: Wraparound Mesh Topology

The aim of the linear equation algorithm is finding an x such that $Ax = b$, where A is an $n \times n$ matrix and x and b are vectors of length n. We use the parallel Jacobi algorithm [Jac], which decomposes A into a diagonal component D and a remainder R, $A = D + R$. The algorithm iterates until the normalised difference between successive iterations is less than a predefined error.

Solver

```
protocol Solver (role Master, ...) {
  rec Iter {
    rec Pipe {
      double_array from Master to Last;
      double_array from Last to East;
      double_array from East to Master;
```


Master
Last
East
West
Diagonal
EastLast
SouthWest
Worker
EastDiagonal
Pipeline data (double array)
Propagation of vector X after iteration

```
   // Other communication in pipeline
   Pipe;
  }
  // Distribute X vector from diagonal
  double_array from Master to SouthWest;
  double_array from Master to West;
  // Distribution of other columns
  Iter;
 }
}
```

Our parallel implementation of this algorithm uses p^2 processors in a $p \times p$ wraparound mesh topology to solve an $n \times n$ system matrix. The matrix is partitioned into submatrix blocks of size n^2/p^2, assigned to each of the processors.

Above shows the global protocol and the dataflow of the linear equation solver implementation with 9 workers.

An endpoint protocol of the `Diagonal` role is listed below.

Solver @ Diagonal

```
protocol Solver at Diagonal
  (role West, role EastLast, role Last, role Worker) {
 rec Iter {
  rec Pipe {
   double_array from West;

   double_array to EastLast;
   Pipe;
  }
  double_array to Last, Worker;
```

```
    Iter;
  }
}
```

The overall iteration of the algorithm is controlled by a `rec Iter` block. In each iteration, the computed values are put into a horizontal pipeline to compute the sums. The resultant X vector is then calculated by the diagonal node to other workers in the mesh for the next iteration.

The corresponding Session C code is given below.

```
1  while (!iter_completed)) {
2    computeProducts(partsum, blkA, newXVec, ...);
3    computeSums(sum, partsum, ...);
4    pipe = 0;
5    while (pipe++ < columns) {
6      send_double_array(EastLast, partsum, blkDim);
7      computeSums(sum, partsum, blkDim);
8      recv_double_array(West, partsum, &length);
9    }
10   // calculate X vector
11   copyXVector(newXVec, oldXVec, ...);
12   computeDivisions(newXVector, sum, ...);
13   msend_double_array(newXVec, Last, Worker, ...);
14 }
```

The asynchronous message optimisation is again applied to the horizontal pipeline (Line 6–8) in order to overlap communications and computations.

3.5. Performance Evaluation

This section presents performance results for the four algorithms which feature different topologies and communication structures.

The purpose of the evaluation is to demonstrate the overhead of implementations using the session primitives, which are needed for session type checking, in the Session C runtime when compared to standard MPI. For all of the algorithms, the computation code is shared between Session C and MPI implementations, so the differences are attributed to communication and overhead of session primitives; Through the benchmarks, we aim to establish that session programming with Session C, which has a smaller and simpler runtime API and guarantees lack of communication errors, can achieve performance comparable to MPI. The runtime differences between the implementations are expected to be constant and represent the overhead of using Session C runtime library.

Additionally, we argue that despite asynchronous optimisation is not suitable for all parallel algorithms (e.g. due to data-dependencies between communication and computation), the optimisation is not just favourable but essential to writing practical parallel programs that are guaranteed communication correct. All four benchmarks are implemented with asynchronous optimisation. In our final benchmark, which implements a Jacobi Solution for Discrete Poisson Equation, we also present a result for an implementation without the optimisation to show that there is a real impact of applying the asynchronous optimisation for suitable algorithms.

The first three benchmarks were taken on workstations with Intel Core i7-2600 processors with 8GB RAM running Ubuntu Linux 11.04; the Jacobi solution benchmarks were taken on a high performance cluster with nodes containing AMD PhenomX49650 processor with 8GB RAM running CentOS 5.6, connected by a dedicated Gigabit Ethernet switch. Each benchmark was run 5 times and the reported runtime is the average of all 5 runs. For the MPI versions, Open MPI 1.4.3 were used. Both use gcc 4.4.3 to compile with the optimisation level `-O3`.

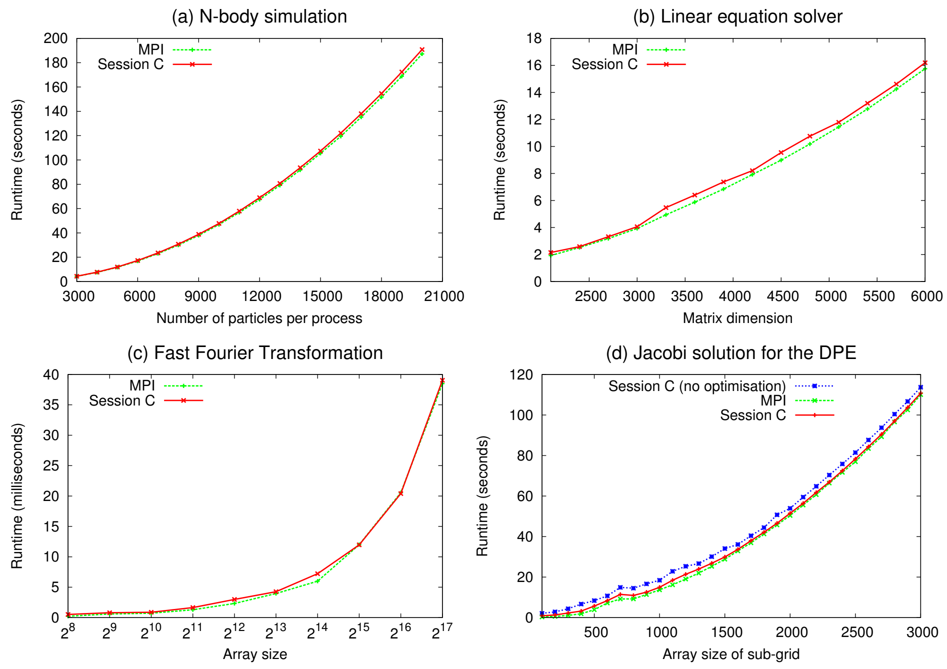


Figure 3.4.: Benchmark results.

N-body simulation Our results are compared against MPI. Both versions use a ring pipeline to propagate the particles, and the two implementations share the same computational component by linking the same compiled object code for the `compute` functions. Our implementations were benchmarked with 3 workers and 1000 iterations on a set of input particles in the two-dimensional space. The results in Figure 3.4(a) show that Session C's execution time is within 3% of the MPI implementation.

Linear equation solver Figure 3.4(b) shows that the MPI linear equation solver is faster than Session C implementation by 1–3%, with the ratio decreasing as the matrix size increases, suggesting the communication overhead is low, if any. The MPI implementation uses `MPI_Bcast` to broadcast the results of each iteration to all nodes in the column, while Session C explicitly distributes the results.

Fast Fourier Transformation (FFT) in a Butterfly Exchange Topology We use a 8 node FFT butterfly (Figure 3.5 shows a butterfly pattern, and Figure 3.6 shows the overall interaction pattern). As seen from Figure 3.4(c), Session C demonstrates a competitive performance compared to FFT implementation, again with the difference in ratio decreasing as the array size gets larger. The algorithm takes advantage of asynchronous optimisation for the butterfly message exchanges.

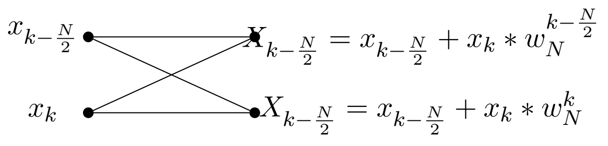

$x_{k-\frac{N}{2}}$
x_k
$X_{k-\frac{N}{2}} = x_{k-\frac{N}{2}} + x_k * w_N^{k-\frac{N}{2}}$
$X_{k-\frac{N}{2}} = x_{k-\frac{N}{2}} + x_k * w_N^{k}$

Figure 3.5.: Butterfly Pattern.

Jacobi solution for the discrete Poisson equation Figure 3.4(d) shows the benchmark results of the implementation of Jacobi solution. The result of the benchmark is very close (within 1%) to that of our reference implementation in MPI. Since the implementation uses a 2D mesh topology, the asynchronous optimisation can be applied to 4 directions of data exchanges between processes. Comparing the results with an unoptimised implementation (i.e. implemented using synchronous communication without

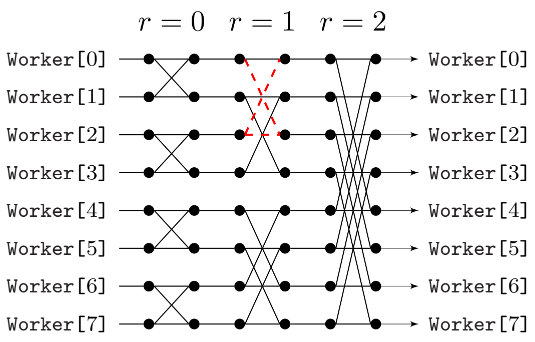


Figure 3.6.: FFT butterfly pattern with 8 processes.

communication-communication overlapping), the performance was improved by up to 8%. In the worse case scenario where there are data dependencies between communication, and it is not possible to use asynchronous optimisation for an implementation, the parallel algorithm will be implemented following the Scribble protocol exactly. Hence we argue that asynchronous optimisation do not degrade the performance of an implementation in Session C.

3.6. Summary and Discussion

We presented Session C, a programming workflow for communication-safe and deadlock-free parallel programming in C. Session C follows the framework of Multiparty Session Types (MPST), where we use Scribble protocol description language to write a communication protocol that governs the global behaviour of the distributed program (global type in MPST), which in turn generates localised protocols called local protocols through a projection algorithm. Session C ensures conformance between endpoint implementations, written with a Session C communications runtime API, and its local protocols by static type checking. Conformance of the local protocols ensures correct communication patterns in the Session C program, and hence guarantees communication safety and deadlock free by the underlying MPST theory. Our type checker also supports asynchronous message optimisation, which allows safe permutation of message send and receive ordering using the asynchronous messaging model. Our results show that optimised parallel applications perform significantly better than naive implementations of the local protocols, and our type checker guarantees correctness of communication in the optimised applications.

A limitation of our static, type checking approach is when the communication pattern is not known statically. We are not able to give the same strong guarantee in the presence of recursion (i.e. loops with loop conditions not captured in Scribble), but in general can be avoided by using synchronised looping primitive (e.g. `repeat`) in Session C.

However, we observed that in order to scale up or add more parallel participants to a Session C program, we need to first write a new Scribble protocol, since Scribble protocols only work for a fixed number of participants; and for each projected local protocol, we need to implement a separate endpoint code. This is cumbersome for large parallel programs where there could be hundreds or thousands of endpoints. In the next chapter, we introduce a new protocol language to address this issue.

4 Pabble: Parameterised Scribble

Overview This chapter presents a parameterised protocol description language, Pabble, which can guarantee safety and progress in a large class of practical, complex parameterised interaction patterns. Pabble can describe an overall interaction topology, using a concise and expressive notation, designed for a variable number of participants arranged in multiple dimensions. These parameterised protocols in turn automatically generate local protocols for type checking parameterised MPI programs for communication safety and deadlock freedom.

4.1. Introduction

In the previous chapter, we introduced Session C, a session-based programming framework which uses a language Scribble [HMB$^+$11, Scr] for describing the multiparty session types in a Java-like syntax. A simple example of a protocol in Scribble which represents a ring topology between four workers is given below:

```
global protocol Ring(role Worker1, role Worker2, role Worker3, role Worker4){
  rec LOOP {
    Data(int) from Worker1 to Worker2;
    Data(int) from Worker2 to Worker3;
    Data(int) from Worker3 to Worker4;
    Data(int) from Worker4 to Worker1;
    continue LOOP;
  }
}
```

This Ring protocol describes a series of communications in which Worker1 passes a message of type Data(int) to Worker4 by forwarding through Worker2 and Worker3 in that order, and receives a message from Worker4. It is easy to notice that explicitly describing all interactions among distinct roles is verbose and inflexible: for example, when extending the protocol with an additional role Worker5, we must rewrite the whole protocol. On the other hand, we observe that these worker roles have identical communication patterns which can be logically grouped together: $\texttt{Worker}_{i+1}$ receives a message from $\texttt{Worker}_i$ and the last Worker sends a message to $\texttt{Worker}_1$. In order to capture these replicable patterns, we introduce an extension of Scribble with dependent types called *Parameterised Scribble* (Pabble). In Pabble, multiple participants can be grouped in the same role and indexed. This greatly enhances the expressive power and modularity of the protocols. Here 'parameterised' refers to the number of participants in a role that can be changed by parameters.

The following shows our ring example in the syntax of Pabble.

```
global protocol Ring(role Worker[1..N]) {
  rec LOOP {
    Data(int) from Worker[i:1..N-1] to Worker[i+1];
    Data(int) from Worker[N] to Worker[1];
    continue LOOP;
  }
}
```

role Worker[1..N] declares workers from 1 to an arbitrary integer N. The Worker roles can be identified individually by their indices, for example, Worker[1] refers to the first and Worker[N] refers to the last. In the body of the protocol, the sender, Worker[i:1..N-1], declares multiple Workers, bound by the bound variable i, and iterates from 1 to N-1. The receivers, Worker[i+1], are calculated on their indices for each instance of the bound variable i. The second line is a message sent back from Worker[N] to Worker[1].

```
local protocol Ring at Worker[1..N](role Worker[1..N]) {
  rec LOOP {
    if Worker[i:2..N]   Data(int) from Worker[i-1];
    if Worker[i:1..N-1] Data(int) to Worker[i+1];
    if Worker[1]        Data(int) from Worker[N];
    if Worker[N]        Data(int) to  Worker[1];
    continue LOOP;
  }
}
```

The above code shows the local protocol of `Ring`, projected with respect to the parameterised `Worker` role. The projection for a parameterised role, such as `Worker[1..N]`, will give a parameterised local protocol. It represents multiple endpoints in the same logical grouping.

Challenges The main technical challenge for the design and implementation of parameterised session types is to develop a method to automatically project a parameterised global protocol to a parameterised local protocol ensuring termination and correctness of the algorithm.

Unfortunately, as in the indexed dependent type theory in the λ-calculus [AH05, XP98], the underlying parameterised session type theory [DYBH12a] has shown that the projection and type checking with general indices are undecidable. Hence there is a tension between termination and expressiveness to enable concise specifications for complex parameterised protocols.

Our main approach to overcome these challenges is to make the theory more practical by extending Scribble with index notation originating from a widely used text book for modelling concurrent Java [MK06]. For example, notations `Worker[i:1..N-1]` and `Worker[j+i]` in the Ring protocol are from [MK06]. Interestingly, this compact notation is not only expressive enough to represent representative topologies ranging from parallel algorithms to distributed web services, but also offers a solution to cope with the undecidability of parameterised multiparty session types.

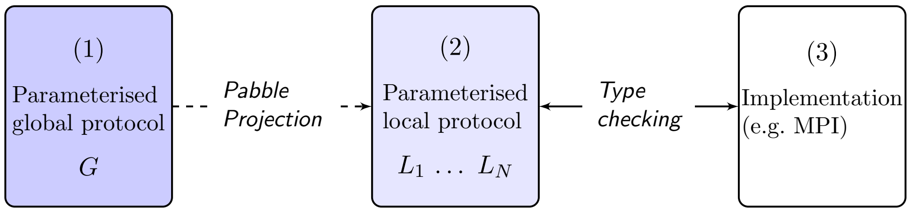

(1)
Parameterised global protocol
G
Pabble Projection
(2)
Parameterised local protocol
$L_1 \ldots L_N$
Type checking
(3)
Implementation (e.g. MPI)

Figure 4.1.: Pabble programming workflow.

4.1.1. Overview

Figure 4.1 shows the relationships between the three layers: global protocols, local protocols and implementations. (1) A programmer first designs a global protocol using Pabble. (2) Then our Pabble tool automatically projects the global protocol into its local protocols. (3) The programmer then either implement the parallel application using the local protocol as specification, or type-check existing parallel applications against the local protocol. If the communication interaction patterns in the implementations follow the local protocols generated from the global protocol, this method automatically ensures deadlock-free and type-safe communication in the implementation. In this work we focus on the design and implementation of the language for describing parallel message-passing based interaction as global and local protocols in (1) and (2).

The contributions of this chapter include:

- The first design and implementation of Parameterised Multiparty Session Types in a global protocol language (Pabble) (Section 4.3.1). The protocols can represent complex topologies with arbitrary number of participants, enhancing expressiveness and modularity for practical message-passing parallel programs.
- The projection algorithm for Pabble to check the well-formedness of parameterised global protocols (Section 4.3.2 and 4.3.3) and to generate parameterised local protocols from well-formed parameterised global protocols (Section 4.3.4). A correctness and termination proof of the projection algorithm is also presented (Section 4.3.6).
- A number of Pabble use cases in parallel programming and web services in Section 4.4.

Additional use cases of Pabble such as common interaction patterns for high performance computing described in Dwarfs [AWW$^+$09] can be found on the project web page [Pab].

4.2. Design aims

There are two main aims of the Pabble design.

- To close the gap between industry-standard MPI and Scribble, which covers most, if not all, use cases of parallel programming; and
- To be able to express scalable protocols in Scribble (number of participants unknown at design time)

In short, we want a middle-ground between high-level protocol description and practical implementation.

MPI uses integers to identify processes (process ids) within communicators, which is a way to easily scale up – in the sense that process ids can be calculated by arithmetic expressions. In parallel programming and MPI, *Worker* processes are processes that share a very similar communication structure that performs calculations. In Pabble, we take the idea of using integer to index processes that are *Worker*s having similar communication structure to replace process names, e.g. Alice, Bob, etc., used in Scribble. This makes the Worker processes easier to scale up to, say 100 processes, where we could end up having different process names referring to a related group of processes.

```
global protocol P(role Worker[1..10]) {
  Data(int) from Worker[i:1..9] to Worker[i+1];
}
```

Which says "each Worker send to its next neighbour". It is reasonable to ask why we do not explicitly iterate through the list of processes, similar to a for-loop in ordinary programming languages, e.g.

```
global protocol P(role Worker[1..10]){
  foreach(i:1..9) {
    Data(int) from Worker[i] to Worker[i+1];
  }
}
```

This has the same semantic meaning as the previous example. However, the difference is when implementing in a Single Program, Multiple Data language/API such as MPI, the two examples are implemented differently, as shown below,

```
// MPI convention
// if Worker[i:2..10] Data(int) from Worker[i-1];
// if Worker[i:1..9] Data(int) to Worker[i+1];

// (rank == i && 2 <= i && rank <= 10)
if (2 <= rank && rank <= 10)
  MPI_Recv(buf, cnt, MPI_INT, rank-1, Data, MPI_COMM_WORLD, &status);
if (1 <= rank && rank <= 9)
    MPI_Send(buf, cnt, MPI_INT, rank+1, Data, MPI_COMM_WORLD);

// foreach (i:1..9) {
//   if Worker[i+1] Data(int) from Worker[i];
//   if Worker[i]  Data(int) to  Worker[i+1];
// }
```

```
for (int i=1; i<=9; i++) {
  if (rank == i+1)
    MPI_Recv(buf, cnt, MPI_INT, i, Data, MPI_COMM_WORLD, &status);
  if (rank == i)
    MPI_Send(buf, cnt, MPI_INT, i+1, Data, MPI_COMM_WORLD);
}
```

The former is the convention of MPI and is more efficient, where each process only evaluates the two statements once, compared to evaluating the statement multiple times for each process.

Above is the reason why a range is used in the condition of the endpoint. Pabble's approach to representing parallel processes come in two key points:

Grouping: related processes are grouped together as a single ”role” where the differences between each individual participant of a conversation are because they are processes at the beginning or end of the index range

Indexing: each process inside a group of processes is identified by its indices

The example of sending from every odd-indexed process to even-indexed process can be expressed as follows:

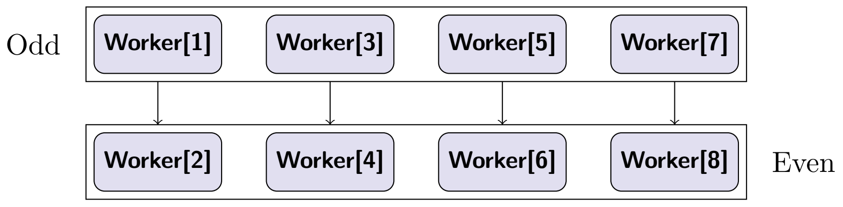
```
global protocol P(role Worker[1..10]) {
  foreach(i:0..4) {
    Data(int) from Worker[i*2+1] to Worker[i*2+2];
  }
}
```

This Pabble protocol makes use of a `foreach` statement which calculates the pairs of processes by explicitly identifying the sender (index =i*2+1) and (index = i*2+2).

In MPI, the natural way of implementing the above may be:

```
if (rank%2 == 0)
  MPI_Recv(buf, cnt, MPI_INT, rank-1, Data, MPI_COMM_WORLD, &status);
if (rank%2 == 1)
  MPI_Send(buf, cnt, MPI_INT, rank+1, Data, MPI_COMM_WORLD);
```

Which splits the process into behaviour when process id (rank) is odd and when rank is even. It is not possible to have a global protocol for this because processes with consecutive indices alternate between sender and receiver, but going back to the idea of “grouping” related processes, since the odd processes and the even processes have different behaviour (one group only sends and another group only receives), the odd-even interaction can be expressed as two groups of communicating indexed processes, i.e.



Since we do not have strict mapping between process orders, we can segment the group as Odd $= 1, 3, 5, 7$ and Even $= 2, 4, 6, 8$, so the protocol may instead be written as:

```
global protocol P(role Odd[1..4], role Even[1..4]) {
  Data(int) from Odd[i:1..4] to Even[i];
}
```

This is the preferred way of writing complex protocols with Pabble, because it leverages the high-level understanding of an interaction pattern (grouping) without taking away the advantage of using indices as process identifiers. The correspondence with MPI is still simple because both groups are indexed.

4.3. Pabble: Parameterised Scribble

Scribble [HMB+11, Scr] is a developer friendly notation for specifying application-level protocols based on the theory of multiparty session types [BCD+08, HYC08]. This section introduces an evolution of Scribble with parameterised multiparty session types (Pabble), defines its endpoint projection and proves its correctness.

4.3.1. Syntax of Pabble

Global Protocols

Figure 4.2 lists the core syntax of Pabble, which consists of two protocol declarations, global and local. A global protocol is declared with the protocol name (*str* denotes a string) with role and group parameters followed by the body G. Role R is a name with argument expressions. The argument expressions are ranges or arithmetic expressions h, and the number of arguments corresponds to the dimension of the array of roles: for example,

Global Pabble			
		$\texttt{global protocol}\ str\texttt{(}para\texttt{) \{ }G\texttt{ \}}$	
Parameter			
$para$	$::=$	$\texttt{role}\ R_d\texttt{, ...,}$	Role declaration
		$\texttt{group}\ str\ \texttt{= \{}R_d\texttt{, ...\}, ...}$	Group declaration
Global protocol body			
G	$::=$	$l(T)\ \texttt{from}\ R\ \texttt{to}\ R\texttt{;}$	Interaction
	$\mid$	$\texttt{choice at}\ R\ \texttt{\{ }G_1\texttt{ \} or ... or \{ }G_N\texttt{ \}}$	Choice
	$\mid$	$\texttt{foreach (}b\texttt{) \{ }G\texttt{ \}}$	Foreach
	$\mid$	$\texttt{allreduce}\ op_c\texttt{(}T\texttt{);}$	Reduction
	$\mid$	$\texttt{rec}\ l\ \texttt{\{ }G\texttt{ \}}$	Recursion
	$\mid$	$\texttt{continue}\ l\texttt{;}$	Continue
	$\mid$	$G\ G$	Sequential composition
Payload type			
T	$::=$	$\texttt{int} \mid \texttt{float} \mid \ldots$	Data types
Expression			
e	$::=$	$e\ op\ e$	Binary expressions
	$\mid$	num	Integers
	$\mid$	$i, j, k, \ldots \mid \texttt{N}$	Variables, constants
op	$::=$	$op_c \mid \texttt{-} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{log} \mid \ldots$	Binary operations
op_c	$::=$	$\texttt{+} \mid \texttt{*} \mid \ldots$	Commutative operations
Role			
R_d	$::=$	str	Role declaration
	$\mid$	$str\texttt{[}e..e\texttt{]}\ldots\texttt{[}e..e\texttt{]}$	Param. role declaration
R	$::=$	str	Roles
	$\mid$	$str\texttt{[}h\texttt{]}\ldots\texttt{[}h\texttt{]}$	Param. roles
	$\mid$	$\texttt{All}$	*All* group role
h	$::=$	$b \mid e$	Role parameter
b	$::=$	$i : e..e$	Binding range
Local Pabble			
		$\texttt{local protocol}\ str\ \texttt{at}\ R_d\texttt{(}para\texttt{) \{ }L\texttt{ \}}$	
Local protocol body			
L	$::=$	$[\texttt{if}\ R]\ l(T)\ \texttt{from}\ R\texttt{;}$	(Conditional) Receive
	$\mid$	$[\texttt{if}\ R]\ l(T)\ \texttt{to}\ R\texttt{;}$	(Conditional) Send
	$\mid$	$\texttt{choice at}\ R\ \texttt{\{ }L_1\texttt{ \} or ... or \{ }L_N\texttt{ \}}$	Choice
	$\mid$	$\texttt{foreach (}b\texttt{) \{ }L\texttt{ \}}$	Foreach
	$\mid$	$\texttt{allreduce}\ op_c\texttt{(}T\texttt{);}$	Reduction
	$\mid$	$\texttt{rec}\ l\ \texttt{\{ }L\texttt{ \}}$	Recursion
	$\mid$	$\texttt{continue}\ l\texttt{;}$	Continue
	$\mid$	$L\ L$	Sequential composition

Figure 4.2.: Pabble syntax.

`Worker[1..4][1..2]` denotes a 2-D array with size 4 and 2 in the two dimensions respectively, forming a 4-by-2 array of roles.

Declared roles can be grouped by specifying a named group using the keyword `group`, followed by the group name and the set of roles. For example,

```
group EvenWorker={Worker[2][2], Worker[4][2]}
```

creates a group which consists of two `Workers`. A special built-in group, `All`, is defined as *all processes in a session*. We can encode collective operators such as many-to-many and many-to-one communication with `All`, which will be explained later.

Apart from specifying ranges by constants, ranges can also be specified using expressions. Expression e consists of operators for numbers, logarithm, left and right logical shifts (`<<`, `>>`), numbers, variables (i, j, k), and constants (`M`, `N`). Constants are either *bound* outside the protocol declaration or are left *free* (unbound) to represent an arbitrary number. As in [MK06], when the constants are bound, they are declared by numbers outside the protocol, e.g. `const N = 100` or lower and upper bounds, e.g. `const N = 1..1000`. We also allow leaving the declaration *free* (unbound), e.g. `const N`, as a shorthand to represent an arbitrary constant with lower and upper bounds `0` and `max` respectively, i.e. `const N = 0..max`, where `max` is a special value representing the maximum possible value or practically unbounded. Binding range expression b takes the form of $i : e_1..e_n$ which means i is ranged from e_1 to e_n. Binding variables always bind to a range expression and not individual values. We shall explain the use of binding range expressions later in more details.

In a global protocol G, $l(T)$ `from` R_1 `to` R_2 is called an *interaction statement*, which represents passing a message with label l and type T from one role R_1 to another role R_2. R_1 is a *sender role* and R_2 is a *receiver role*. `choice at` R `{` G_1 `}` `or ... or` `{` G_N `}` means the role R will select one of the global types G_1,. . . ,G_N. `rec` l `{` G `}` is recursion with the label l which declares a label for `continue` l statement. `foreach (`b`) {`G`}` denotes a for-loop whose iteration is specified by b. For example, `foreach (i:1..n ){` G `}` represents the iteration from 1 to n of G where G is parameterised by i.

Finally, `allreduce` op_c`(`T`)` means all processes perform a distributed reduction of value with type T with the operator op_c (like `MPI_Allreduce` in MPI). It takes a mandatory predefined operator op_c where op_c must be a commutative and associative arithmetic operation so they can correspond to MPI reduction operations which have the same requirements. Pabble currently supports sum and product.

We allow using simple expressions (e.g. `Worker[i:0..2*N-1]`) to parameterise ranges. In addition, indices can also be calculated by expressions on bound variables (e.g. `Worker[i+1]`) to refer to relative positions of roles.

These restrictions on indices such as bound variables and relative indices calculations ensure termination of the projection algorithm and type checking. The binding conditions are discussed in the next subsection.

Local Protocols

Local protocol L consists of the same syntax of the global type except the input from R (receive) and the output to R (send). The main declaration `local protocol` *str* `at` R_e `(...)` `{` L `}` means the protocol is located at role R_e. We call R_e *the endpoint role.* In Pabble, multiple local protocol instances can reside in the same parameterised local protocol. This is because each local protocol is a local specification for a participant of the interaction. Where there are multiple participants with a similar interaction structure that fulfil the same *role* in the protocol, such as the `Workers` from our `Ring` example from the introduction, the participants are grouped together as a single parameterised role. The local protocol for a collection of participants can be specified in a single parameterised local protocol, using *conditional statements* on the role indices to capture corner cases. For example, in a general case of a pipeline interaction, all participants receive from one neighbour and send to another neighbour, except the first participant which initiates the pipeline and is only a sender and the last participant which ends the pipeline and does not send. In these cases we use conditional statements to guard the input or output statements. To express conditional statements in local protocols, `if` R may be prepended to an input or output statement. `if` R input/output statement will be ignored if the local role does not match R. More complicated matches can be performed with a parameterised role, where the role parameter range of the condition is matched against the parameter of the local role. For example, `if Worker[1..3]` will match `Worker[2]` but not `Worker[4]`. It is also possible to bind a variable to the range in the condition, e.g. `if Worker[i:1..3]`, and `i` can be used in the same statement.

4.3.2. Well-formedness Conditions: Index Binding

As Pabble protocols include expressions in parameters, a valid Pabble protocol is subject to a few well-formedness conditions. Below we show the conditions which ensure indices used in roles are correctly bound. We use fv/bv to denote the set of free/bound variables defined as $\mathsf{fv}(i) = \{i\}$, $\mathsf{fv}(\texttt{N}) = \mathsf{fv}(\mathit{num}) = \emptyset$ and $\mathsf{fv}(i : e_1 \ldots e_n) = \cup\mathsf{fv}(e_j)$ and $\mathsf{fv}(\texttt{foreach}(b)\{G\}) = (\mathsf{fv}(b) \cup \mathsf{fv}(G)) \setminus \mathsf{bv}(b)$ and $\mathsf{bv}(i : e_1 \ldots e_n) = \{i\}$. Others are inductively defined.

1. In a global protocol role declaration, `global protocol`, indices outside of declared range are "invalid", for example, a role `Worker[0]` is invalid if the role is declared `role Worker[1..3]`.

2. Let $\texttt{foreach}(b_1)\{\ \texttt{foreach}(b_2)\{\ \ldots \texttt{foreach}(b_n)\{G\}\}\}$ with $n \geq 0$:

   a) Suppose an interaction statement $l(T)$ `from` R_1 `to` R_2`;` appears in G. Let $R_1 = \mathit{Role}_1[h_1] \ldots [h_n]$ and $R_2 = \mathit{Role}_2[e'_1] \ldots [e'_m]$ (we assume $n = 0$ (resp. $m = 0$) if R_1 (resp. R_2) is either a single participant or group).

      (1) $n = m$ (i.e. the dimensions of the parameters are the same)

      (2) $\mathsf{fv}(h_j) \subseteq \cup\mathsf{bv}(b_i)$ (i.e. the free variables in the sender roles are bound by the for-loops).

      (3) $\mathsf{fv}(e'_j) \subseteq (\cup\mathsf{bv}(b_i)) \cup \mathsf{bv}(h_j)$ (i.e. the free variables in the receiver roles are bound by either the for-loops or sender roles);

   b) Suppose a choice statement `choice at` R `{` G_1 `}` `or` `{` G_2 `}` appears in G. Then R is a single participant, i.e. either Role or $\mathit{Role}[e]$ with $\mathsf{fv}(e) \subseteq (\cup\mathsf{bv}(b_i))$.

Condition 2(a)(1) ensures the number of sender parameters matches the number of receiver parameters. For example, the following is invalid:

$l(T)$ `from R[i:1..N-1][j:1..N] to R[i+1];`

Condition 2(a)(2) ensures variables used by a sender are declared by the enclosing for-loops.

Condition 2(a)(3) makes sure the receiver parameter at the j-th position is bound by the for-loops or the sender parameter at the j-th position (and not binders at other positions). For example, the following is valid:

$l(T)$ `from R[i:1..N-1][j:1..N] to R[i+1][j];`

But with the index swapped, it becomes invalid:

$l(T)$ `from R[i:1..N-1][j:1..N] to R[j][i+1];`

Condition 2(b) is similar for the case of `choice` statements where R should be a single participant to satisfy the unique sender condition in [CDCP12, DY12].

4.3.3. Well-formedness Conditions: Constants

In Pabble protocols, constants can be defined by

(1) A single numeric value (`const N=4`); or

(2) Lower and upper bound constraints not involving the `max` keyword (e.g. `const N=1..1000`).

Lower and upper bound constraints are designed for runtime constants, e.g. the number of processes spawned in a scalable protocol, which is unknown at design time and will be defined and immutable once the execution begins. To ensure Pabble protocols are communication-safe in all possible values of constants, we must ensure that all parametrised role indices stay within their declared range. Such conditions prevent sending or receiving from an invalid (non-existent) role which will lead to communication mismatch at runtime.

In case (1), the check is trivial. In case (2), we require a general algorithm to check the validity between multiple constraints appear in the regions. First, we formulate the constraints of the values of the constants as a series of linear inequalities. We then combine the linear inequalities and determine the feasible region using integer linear programming. The feasible region represents the pool of possible values in any combination of the constraints. The following explains how to determine whether the protocol will be valid for all combinations of constants:

```
const M = 1..3;
const N = 2..5;
```

```
3 global protocol P(role R[1..N]) {
4   T from R[i:1..M] to R[i+1];
5 }
```

The basic constraints from the constants are:

$$1 \leq \texttt{M},\ \texttt{M} \leq 3,\ 2 \leq \texttt{N} \text{ and } \texttt{N} \leq 5$$

We then calculate the range of `R[i+1]` as `R[2..M+1]`. Since the objective is to ensure that the role parameters in the protocol body (i.e. `1..M` and `2..M+1`) stay within the bounds of `1..N`, we define a constraint set to be:

$$1 \leq 1 \text{ \& } \texttt{M} \leq \texttt{N} \text{ and } 1 \leq 2 \text{ \& } \texttt{M} + 1 \leq \texttt{N}$$

which are lower and upper bound inequalities of the two ranges. From them, we obtain this inequality as a result:

$$\texttt{M} + 1 \leq \texttt{N}$$

By comparing this against the basic constraints on the constants, we can check that not all outcomes belong to the regions and thus this is not a communication-safe protocol (an example of a unsafe case is $\texttt{M} = 3$ and $\texttt{N} = 2$). On the other hand, if we alter Line 4 to `T from R[i:1..N-1] to R[i+1];`, the constraints are unconditionally true and so we can guarantee all combinations of constants `M` and `N` will not cause communication errors.

Arbitrary Constants In addition to constant values and lower and upper bound constants, we also consider the use cases when the value of a constant can be any arbitrary value in the set of natural numbers. This is an extension of case (2) with the `max` keyword, where we write `const N = 0..max` to represent a range without upper bound.

In order to check that role indices are valid with unbounded ranges, we enforce two simple restrictions. First, only one constant can be defined with `max` in one global protocol[1]. Secondly, when the index is unbounded, its range calculation only uses addition or subtraction on integers (e.g. `i+1`).

A protocol with an invalid use of arbitrary constants is shown below:

```
const N = 1..max;
global protocol Invalid(role R[1..N]) {
  T from R[i:1..N-1] to R[i+1];
  T from R[j:1..N] to R[j+1];
}
```

[1]Or all arbitrary constants in the global protocol can be expressed in terms of a single arbitrary constant

If `N` is instantiated to `1`, then the role is declared to be `R[1..1]`. In the first interaction statement, `R[i:1..1-1]` is invalid, as `R[0]` is not in the range of `R[1..0]`. In the second statement `R[j+1]` is also invalid, as it evaluates to `R[N+1]` and is out of range `R[1..N]`.

On the other hand, the following protocol is valid since the indices always stay between 0 and `N`.

```
const N = 1..max;
global protocol Valid(role R[0..N]) {
  T from R[i:0..N-1] to R[i+1];
  T from R[j:1..N] to R[j-1];
}
```

Most representative topologies with an arbitrary number of participants can be represented under these conditions, we show some examples in the next chapter.

4.3.4. Endpoint Projection

In the next step, a Pabble protocol should be *projected* to a local protocol, which is a simplified Pabble protocol as viewed from the perspective of a given endpoint. The projection algorithm is described below. To begin with, the header of the global protocol

$$\texttt{global protocol}\ \mathit{name(param)}\ \{\ G\ \}$$

is projected onto

$$\texttt{local protocol}\ \mathit{name}\ \texttt{at}\ R_e\mathit{(param)}\ \{\ L\ \}$$

where the protocol name *name* and parameters *param* are preserved and the endpoint role R_e is declared.

Table 4.1 shows the projection of the body of global protocol G onto $\boldsymbol{R}$ at endpoint role R_e. The projection rules will be applied from top to bottom in the table, if a global protocol matches multiple rules, then there will be more than one line of projected protocol for a single global protocol. In Rules 1–4, we show the rule for the single argument as the same rule is applied to n-arguments. Each rule is applied if $\boldsymbol{R}$ meets the condition in the second column under the constraints given by the constant declarations. Rules 1 and 2 show the projection of the interaction statement when $\boldsymbol{R}$ appears in the receiver and the sender position respectively. Since $\boldsymbol{R}$ is a single

		Condition	Global Protocol	Projected Local Protocol
1	Receive	$\boldsymbol{R} = R_e$	U `from` R' `to` $\boldsymbol{R}$`;`	U `from` R'`;`
2	Send	$\boldsymbol{R} = R_e$	U `from` $\boldsymbol{R}$ `to` R'`;`	U `to` R'`;`
3	Param. Receive	$\boldsymbol{R} \in R_e$	U `from` R' `to` $\boldsymbol{R}$`;`	`if` $\boldsymbol{R}$ U `to` R'`;`
4	Param. Send	$\boldsymbol{R} \in R_e$	U `from` $\boldsymbol{R}$ `to` R'`;`	`if` $\boldsymbol{R}$ U `from` R'`;`
5	All to All		U `from All to All;`	U `to All;` U `from All;`
6	Group	$\boldsymbol{R} \subseteq R_e$	U `from` R' `to` $\boldsymbol{R}$`;`	`if` $\boldsymbol{R}$ U `from` $\boldsymbol{R'}$`;`
7	Group	$\boldsymbol{R} \subseteq R_e$	U `from` $\boldsymbol{R}$ `to` R'`;`	`if` $\boldsymbol{R}$ U `to` R'`;`
8	Relative Role	$\boldsymbol{R}[e] \subseteq R_e$	U `from` R'`[`b`]` `to` $\boldsymbol{R}$`[`e`];`	`if` $\boldsymbol{R}$`[apply(`b`,` e`)]` U `from` R'`[inv(e)];`
9	Relative Role	$\boldsymbol{R}[b] \subseteq R_e$	U `from` $\boldsymbol{R}$`[`b`]` `to` R'`[`e`];`	`if` $\boldsymbol{R}$`[`b`]` U `to` R'`[`e`];`

10	Choice Sender	$\boldsymbol{R} = R_e$
		 or $\boldsymbol{R} \in R_e$
			`choice at` $\boldsymbol{R}$ `{` G_1 `}`
			 `or ... or {` G_N `}`
				`choice at` $\boldsymbol{R}$ `{` L_1 `}`
				 `or ... or {` L_N `}`

11	Choice Receiver		`choice at` R' `{` G_1 `}`
			 `or ... or {` G_N `}`
				`choice at` R' `{` L_1 `}`
				 `or ... or {` L_N `}`

12	Recursion		`rec` l `{` G `}`	`rec` l `{` L `}`
13	Continue		`continue` l`;`	`continue` l`;`
14	Foreach		`foreach (`b`) {` G `}`	`foreach (`b`) {` L `}`
15	All reduce		`allreduce` op_c`(`T`);`	`allreduce` op_c`(`T`);`

Table 4.1.: Projection of G onto $\boldsymbol{R}$ at the end-point role R_e.
L and L_i correspond to the projection of G and G_i onto $\boldsymbol{R}$.

participant, it should satisfy $\boldsymbol{R} = R_e$ (i.e. the role is the endpoint role). The projection simply removes the reference to role $\boldsymbol{R}$ from the original interaction statement.

Rules 3 and 4 show the projection of an interaction statement if role $\boldsymbol{R}$ is a parameterised single participant where $\boldsymbol{R}$ is an element of the endpoint role R_e. For example, if $R_e =$ `Worker[1..3]`, $\boldsymbol{R}$ can be either `Worker[1]`, `Worker[2]` or `Worker[3]`. In addition to removing the reference of role $\boldsymbol{R}$ in the receive and send statements, we also prepend the conditions which the role applies. The order of which the projection rules are applied ensure that an interaction statement will be localised to receive then send. In general, both receive-send or send-receive in the projected local protocol is correct, as long as the projection algorithm is consistent and the well-formedness conditions of the global protocol are satisfied. The global protocol will ensure, by session typing, that a send will have a matching receive at the same stage of the protocol.

Rule 5 is for All-to-All communication. Any role $\boldsymbol{R}$ will send a message with type U to all other participants and will receive some value with type U from all other participants. Since all participants start by first sending a message to all, no participant will block waiting to receive in the first phase, so no deadlock occurs.

Rules 6 and 7 are the projection rules for the case that we project onto a group. We need to check that a group is a subset of the endpoint role R_e with respect to the group declarations in the global protocol. Then the rules can be understood as Rules 3 and 4.

Range (b)	Expression (e)	`apply`(b, e)	`inv`(e)
`i:1..N`	`i+1`	`i:2..N+1`	`i-1`
`i:1..3`	`i*2`	`i:2,4,6`	`i/2`
`i:1..3`	`i`	`i:1..3`	`i`
`i:0..3`	`1<<i`	`i:1,2,4,8`	`log(i, 2)`
`i:1..3`	`i%2`	`i:1,0,1`	Invalid

Table 4.2.: Examples of `apply()` and `inv()`.

Rules 8 and 9 show the projection of interaction statements with parameterised roles using relative indexing (we show only one argument: the algorithm can be extended easily to multiple arguments using the same methods). Rule 8 uses two auxiliary transformations of expressions, `apply` and `inv`. Table 4.2 lists their examples. `apply` takes two arguments, a range with binding variable (b) and an expression using the binding variable (e). The expression is *applied* to both ends of the range to transform the relative expression into a well defined range. `inv` calculates the inverse of a given expression, for example, the inverse of `i+1` is `i-1` and the inverse of `i*2+1` is `(i-1)/2`. In cases when an inverse expression cannot be derived, such as `i%2`, the expression will be calculated by expanding to all values in the range and instantiating every value bound by its binding variable (e.g. `i`).

A concrete example is given as follows, to project the statement

```
U from W[i:1..3] to W[(i+1)%2];
```

the statement will be expanded to

```
U from W[1] to W[0];
U from W[2] to W[1];
```

```
U from W[3] to W[0];
```

before applying the projection rules. In order to perform the range expansion above, the beginning and the end of the range must be known at projection time. For this reason, the projection algorithm returns failure if a statement uses parameterised roles with such expressions and the range of the expressions is defined with arbitrary constants (see Section 4.3.3). Otherwise, the expressions might expand infinitely and not terminate. This is the only situation where projection may fail, given a well-formed global protocol. The condition $\boldsymbol{R}[b] \subseteq R_e$ of Rule 9 means the range of b is within the range of the endpoint role R_e. For example, `W[i:1..2]` $\subseteq$ `W[1..3]`.

If a projection role matches the choice role ($\boldsymbol{R}$ in `choice at` $\boldsymbol{R}$) (Rule 10), then it means a selection statement, whose action is selecting a branching by sending a label. The child or-blocks ($L_1 \ldots L_N$) are recursively projected; whereas if a projection role does not match the choice role (Rule 11), then the choice statement represents a branch statement, which is the dual of the selection. For recursion (Rule 12), continue (Rule 13) and foreach (Rule 14) statements are just kept in the projected endpoint protocol.

4.3.5. Collective Operations

In addition to point-to-point message-passing, collective operations can also be concisely represented by Pabble. Endpoint message-passing statements are interpreted differently depending on the declarations (i.e. parameters) in the global type. Figure 4.3–4.6 lists the four basic messaging patterns and the interpretations of their projections: point-to-point, scatter (distribution), gather (collection) and all-to-all (symmetric distribution and collection). As shown in the Figures, the combination of projected local statements and the type (i.e. single participant or group role) of the local role being projected are unique and can identify the communication pattern in the global protocol.

4.3.6. Termination and Correctness of the Projection

The parameterised session theory which Pabble is based on [DYBH12a] has shown that, in the general case, projection and type checking are undecidable. Our first challenge for Pabble's design is to ensure the termination of well-formed checking and projection, without sacrificing the expressiveness. The theorems and proofs can be found in this section.

Point-to-Point

Pabble role declarations:

`role A[1..M], role B[1..N]`

$$\begin{tikzcd} A_1 \arrow[r] & B_1 \\ A_2 \arrow[r] & B_2 \\ A_3 \arrow[r] & B_3 \end{tikzcd}$$

Pabble statement	Projection of A	Projection of B
`U from A to B;`	`U to B;`	`U from A;`
`U from A[i] to B[j];`	`if A[i] U to B[j];`	`if B[j] U from A[i];`
`U from A[i:1..N] to B[i+1];`	`if A[i:1..N] U to B[i+1];`	`if B[i:2..N+1] U from A[i-1];`

Figure 4.3.: Point-to-point communication and Pabble representation.

Scatter pattern

Pabble role declarations:

`role A, role B[1..N], group C`

$$\begin{tikzcd} & C_1 \\ \begin{matrix} A \\ B[i] \end{matrix} \arrow[ur] \arrow[r] \arrow[dr] & C_2 \\ & C_3 \end{tikzcd}$$

Pabble statement	Projection of A/B	Projection of C
`U from A to C;`	`U to C;`	`if C U from A;`
`U from B[i] to C;`	`if B[i] U to C;`	`if C U from B[i];`

Figure 4.4.: Scatter pattern and Pabble representation.

Gather pattern

Pabble role declarations:

`group A, role B, role C[1..N]`

$$\begin{tikzcd} A_1 \arrow[dr] & \\ A_2 \arrow[r] & \begin{matrix} B \\ C[i] \end{matrix} \\ A_3 \arrow[ur] & \end{tikzcd}$$

Pabble statement	Projection of A	Projection of B/C
`U from A to B;`	`if A U to B;`	`U from A;`
`U from A to C[i];`	`if A U to C[i];`	`if C[i] U from A;`

Figure 4.5.: Gather pattern and Pabble representation.

Theorem 4.1 (Termination) *Given global protocol G, the well-formed checking terminates; and given a well-formed global type G and an endpoint role R_e, projection G on R_e always terminates.*

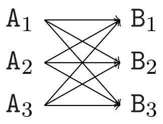
Pooof. Given a well-formed global type G and an endpoint role R_e, by case analysis on each projection rule in Table 4.1:

Case 1. Receive. Trivial because $\boldsymbol{R} = R_e$ checking terminates.

All-to-all pattern

Pabble role declarations:

`group A, group B`



Pabble statement	Projection of A	Projection of B
`U from A to B;`	`if A U to B;`	`if B U from A;`
`U from All to All;`	`U to All; U from All;`	`U to All; U from All;`

Figure 4.6.: All-to-all pattern and Pabble representation.

Case 2. Send. Trivial because $\boldsymbol{R} = R_e$ checking terminates.

Case 3. Param. Receive. Trivial because $\boldsymbol{R} \in R_e$ checking terminates.

Case 4. Param. Send. Trivial because $\boldsymbol{R} \in R_e$ checking terminates.

Case 5. All to All. Trivial.

Case 6. Group. There are two sub cases:

1. if R_e is an unbounded role, and $\boldsymbol{R}$ is an unbounded role, Given a well-formed global type, there can only be a single unbounded role, so $\boldsymbol{R} \subseteq R_e$, otherwise $\boldsymbol{R} \subsetneq R_e$ then terminate. If $\boldsymbol{R}$ is a bounded role, $\boldsymbol{R} \subseteq R_e$ terminates after iterating through all its members.

2. if R_e is a bounded role, $\boldsymbol{R} \subseteq R_e$ terminates after iterating through all members.

Case 7. Group. Terminates as Case 6.

Case 8. Relative Role. There are two sub cases in the condition:

1. if R_e is an unbounded role, and $\boldsymbol{R}[\boldsymbol{e}]$ is an unbounded role, Given a well-formed global type, there can only be a single unbounded role, so $\boldsymbol{R}[\boldsymbol{e}] \subseteq R_e$, otherwise $\boldsymbol{R}[\boldsymbol{e}] \subsetneq R_e$ then terminate. If $\boldsymbol{R}[\boldsymbol{e}]$ is a bounded role, $\boldsymbol{R}[\boldsymbol{e}] \subseteq R_e$ terminates after iterating through all its members.

2. if R_e is a bounded role, $\boldsymbol{R}[\boldsymbol{e}] \subseteq R_e$ terminates after iterating through all members.

The termination of this rule is then ensured by the termination of `apply`(b, e) and `inv`(e). If `inv`(e) is not defined, we first check e has a finite range and use Rule 3 and 4 by expanding the interaction statements to all values in the range (as explained in Section 4.3.4).

Case 9. Relative Role. There are two sub cases in the condition:

1. if R_e is an unbounded role, and $\boldsymbol{R}[\boldsymbol{b}]$ is an unbounded role, Given a well-formed global type, there can only be a single unbounded role, so $\boldsymbol{R}[\boldsymbol{b}] \subseteq R_e$, otherwise $\boldsymbol{R}[\boldsymbol{b}] \subsetneq R_e$ then terminate. If $\boldsymbol{R}[\boldsymbol{b}]$ is a bounded role, $\boldsymbol{R}[\boldsymbol{b}] \subseteq R_e$ terminates after iterating through all its members.

2. if R_e is a bounded role, $\boldsymbol{R}[\boldsymbol{b}] \subseteq R_e$ terminates after iterating through all members.

Case 10. Choice Sender. If R_e is a single participant, $\boldsymbol{R} = R_e$ checking terminates. If R_e is a parameterised role, $\boldsymbol{R} \in R_e$ checking terminates. Given a well-formed protocol, the projection of $G_{i\in\{1..N\}}$ to $L_{i\in\{1..N\}}$ terminates, hence this rule terminates.

Case 11. Choice Receiver. Given a well-formed protocol, the projection of $G_{i\in\{1..N\}}$ to $L_{i\in\{1..N\}}$ terminates, hence this rule terminates.

Case 12. Recursion. Given a well-formed protocol, the projection of G to L terminates, hence this rule terminates.

Case 13. Continue. Trivial.

Case 14. Foreach. Given a well-formed protocol, the projection of G to L terminates, hence this rule terminates.

Case 15. Allreduce. Trivial.

By the definition of the well-formedness conditions in Section 4.3.3 and 4.3.2, if a free variable appears in the range position, it is bound by either for-loops or the sender role in the interaction statement. In the case of the for-loop, we can apply the same reduction rules of the for-loop of the global types and apply the equality rules in [DYBH12a, Figure 15]. Hence one can check, given R_e and $\boldsymbol{R}$, all of the conditions (in the second column) in Table 4.1 are decidable. For the projection, the only non-trivial projection rule is Rule 8. Hence the projection algorithm always terminates. □

Note that the above theorem implies the termination of type checking (see Theorem 4.4 in [DYBH12a]).

Theorem 4.2 (Range) *The indices of roles appearing in a local protocol body do not exceed the lower and upper bounds stated in the global protocol ProtocolName(para) in `global protocol ProtocolName(para){ G }` or the constant declarations (`const N = n..m;`).*

Pooof.

- If the range of indices of roles relies on case (1) of Section 4.3.3 (i.e. single numeric value), the constant will be within bounds in the local protocol as it appears in the global protocol, ensured by each condition in the projection algorithm in Table 4.1 which checks whether the roles conform to the bounds in the global protocol.
- If the range of indices of roles relies on case (2) Section 4.3.3 (i.e. lower and upper bounds not involving the `max` keyword), the correctness is ensured by the constraints of integer linear programming from the well-formedness conditions.

□

4.3.7. Correctness of projection

The theorems above states the termination of the projection algorithm and the correctness of ranges. We discuss the correctness of the projection algorithm below, by analysing each rule in Table 4.1. We either correspond each of our rules with existing theory which is proven correct or give an argument as to why the projection is correct.

Case 1. Receive. This corresponds to the standard receive in the MPST.

Case 2. Send. This corresponds to the standard send in the MPST.

Case 3. Param. Receive. Parameterised participants are treated as ordinary participants, so this corresponds to the standard receive in the MPST.

Case 4. Param. Send. Parameterised participants are treated as ordinary participants, so this corresponds to the standard send in the MPST.

Case 5. All to All. Suppose there is only 1 participant, i.e. `All` = { `P1` }. U `from All to All;` is equivalent to U `from P1 to P1;` This is a special case of the first case of the first rule in [DYBH12a, Figure 10], where the sender is also the receiver. Hence it is projected as U `to P1;` U `from P1;`. Since all participants (i.e. `P1` only) follow the above endpoint type, it is equivalent to U `to All;` U `from All;`). Suppose there are more than 1 participants, i.e. `All` = { `P1`, ..., `P`n }. U `from All to All;` is equivalent to U `from P1 to P1;` U `from P1 to P2;` ... U `from P1 to P`n`;` ...U `from P`n `to P1;` U `from P`n `to P2;` ... U `from P`n `to P`n `;`. Following the projection rules above and project for the endpoint role `All` (i.e. `P1`, ..., `P`n), we obtain for each endpoint:

P1	U `to P1;` U `from P1;` U `to P2;` ... U `to P`n`;` U `from P2;` ...
Pi	...
Pn	U `from P1;` U `from P2;` ... U `to P`n`;` U `from P`n`;`

We apply the asynchronous subtyping rules in [MYH09] to reorder the interactions for each endpoint such that all endpoints interact with the same participant. For example,

P1	U `to P1;` U `to P2;` ... U `to P`n`;` U `from P1;` ... U `from P`n`;`
Pi	...
Pn	U `to P1;` U `to P2;` ... U `to P`n`;` U `from P1;` ... U `from P`n`;`

Since the ordering of self-interaction (e.g. U `to P1;` U `from P1;`) cannot be permuted, the only ordering we can obtain for all participants is U `to P1;` U `to P2;` ... U `to P`n`;` U `from P1;` U `from P2;` ... U `from P`n`;`, which is equivalent to U `to All;` U `from All;`. Similar reasoning extends to protocols with unbounded roles, hence the projection rule is correct.

Case 6. Group. The correctness of projection is ensured by the projection rule 5, 6 (projection of a quantified global type) of [DY11, Figure 5], but instead of applying on a single role, apply on all members of group $\boldsymbol{R}$.

Case 7. Group. The correctness of projection is ensured by the projection rule 5, 6 (projection of a quantified global type) of [DY11, Figure 5], but instead of applying on a single role, apply on all members of group $\boldsymbol{R}$.

Case 8. Relative Role. A relative role statement U `from` R'`[`b`]` `to` $\boldsymbol{R}$`[`e`];` can be expanded to multiple interaction statements. Suppose b is defined to be the range from i to j and e is defined to be a function $f()$. The statement is expanded to U `from` R'`[`i`]` `to` $\boldsymbol{R}$`[`$f(i)$`];` U `from` R'`[`$i+1$`]` `to` $\boldsymbol{R}$`[`$f(i+1)$`];` $\ldots U$ `from` R'`[`j`]` `to` $\boldsymbol{R}$`[`$f(j)$`];`. By applying Rule 3, 4 with the definition of `inv()` (which defines the inverse of f)[2], correctness is ensured by Rule 3 and 4.

Case 9. Relative Role. Similar to Case 8, a relative role statement is expanded and the correctness is ensured by Rule 3, 4, but more straightforward since `inv()` is not involved in the projection.

Case 10. Choice Sender. The correctness of this projection rule is ensured by rule 2, case 1 (projection of branching global type to selection endpoint type) in [DYBH12a, Figure 10].

Case 11. Choice Receiver. The correctness of this projection rule is ensured by rule 2, case 2 (projection of branching global type to branching endpoint type) in [DYBH12a, Figure 10].

Case 12. Recursion. The correctness of this projection rule is ensured by rule 4 (projection of recursion) in [DYBH12a, Figure 10].

Case 13. Continue. The correctness of this projection rule is ensured by rule 5 (projection of type variable) in [DYBH12a, Figure 10].

Case 14. Foreach. To show the correctness of `foreach` we can apply same reduction rules of the for-loop of the global types and apply the equality rules in [DYBH12a, Figure 15].

[2]As explained in Section 4.3.4, projection will fail if the inverse cannot be derived.

Case 15. All reduce. The `allreduce` primitive encapsulates the pattern of simultaneous all to all and reduction. The collective reduction involves all roles, but cannot be decomposed to individual send and receive as with Rule 5 (All to All) due to the reduction computation. Hence the local protocol of `allreduce` is defined as the same as its global protocol as an atomic operation. Its correctness is ensured by the definition.

4.4. Pabble Examples

In Section 4.3.4 we describe how to obtain a local Pabble protocol by projection from a Pabble protocol. The local protocol can then be used as a blueprint to implement parallel programs. In this section we run through two examples of local protocol projection, using a `Ring` protocol in Section 4.4.1 and a `ScatterGather` protocol in Section 4.4.2, showing projection of protocols involving point-to-point and multicast collective applications respectively.

Then we present Pabble use cases in Web services in Section 4.4.3 and Remote Procedure Call (RPC) composition in Section 4.4.4, showing the capabilities of Pabble as a general-purpose parameterised protocol description language.

Finally we show an implementation of a parallel linear equation solver (Section 4.4.5) in MPI following a wraparound mesh protocol designed in Pabble, demonstrating how Pabble can be used in practical programming. Additional Pabble examples from the Dwarfs [AWW$^+$09] are presented in the next chapter.

4.4.1. Projection Example: Ring Protocol

We now run through the projection of the `Ring` protocol in Section 4.1 as an example. Local protocols are generated from the global protocols. From the perspective of a projection tool, to write a protocol for an endpoint, we start with `local protocol` followed by the name of the protocol and the endpoint role it is projected for. Since the only role of the `Ring` protocol is `Worker` which is a parameterised role, we use the full definition of the parameterised role, `Worker[1..N]`. Then we list the roles used in the protocol inside a pair of parentheses, similar to function arguments in a function definition in C. Note that if the projection role is in the list, we exclude it because the local protocol itself is in the perspective of that role; however, since parameterised roles can be used on multiple endpoint roles, we allow parameterised roles to appear in the list of roles in the protocol. The first line of the projected protocol is thus given as follows:

```
local protocol Ring at Worker[1..N](role Worker[1..N])
```

We then copy the recursion statement to the local protocol, which will be present in all projected protocols.

```
  rec LOOP {
```

Next, we take the first interaction statement from `Ring` protocol and project it with respect to `Worker`, applying the rules listed in Table 4.1. As the first statement involves a parameterised destination role, we apply Rule 7 to extract the receive portion of the interaction statement. The `apply()` function is applied to `i:1..N-1` and the relative expression `i+1` to obtain `2..N` for the role condition. The `inv()` of relative expression `i+1` is `i-1`, which will form the index of the sender role.

```
  if Worker[i:2..N] Data(int) from Worker[i-1] ;
```

Since `Worker` also matches the source parameterised role, Rule 8 is applied to get the send portion of the interaction statement.

```
  if Worker[i:1..N-1] Data(int) to Worker[i+1];
```

Then we move on to the second statement of the global protocol, which is `Data (int)from Worker[N] to Worker[1];`. Similar to the previous statement, we apply Rule 3 and Rule 4 to obtain the respective receive and send statements in the local protocol.

```
  if Worker[1] Data(int) from Worker[N];
  if Worker[N] Data(int) to Worker[1];
```

Finally we apply Rule 13 to trivially copy the `continue` statement to the local protocol.

```
  continue LOOP; }
```

The resulting local protocol is shown in Section 4.1.

4.4.2. Projection Example: ScatterGather Protocol

The following example shows another parameterised protocol, which represents the scatter-gather pattern of work distribution and reduction. This example uses a common parallel programming idiom, collective operations. In contrast to the previous example, there is more than one declared role in the protocol, and one of the roles is an ordinary non-parameterised role.

```
1  global protocol ScatterGather(role Master, role Worker[1..N], group Workers={
       Worker[1..N]}){
2    Scatter(int) from Master to Workers;
3    Gather(int) from Workers to Master;
4  }
```

Listing 4.1: ScatterGather global protocol.

In this protocol, the statements involve two roles, one of which is an ordinary role `Master` (in the sense that it is non-parameterised), and the other is a parameterised role `Worker[i:1..N]`. The `Worker` parameterised role represents a group of related roles, but does not expand to multiple explicit message-passing statements. We further declare a group role `Workers` which include all the `Worker` roles as members. The statement in Line 2 is a *scatter* operation by which the `Master` distributes a message of type `Scatter(int)` to each of the named endpoints in `Workers` group, `Worker[1]` to `Worker[N]`. The statement in Line 3 is a *gather* operation, the reverse of the scatter, which the `Master` role collects messages of type `Gather(int)` from the members of the `Workers` group. Figure 4.7 depicts the interactions in the protocol.

Listing 4.2 shows the local protocol of `ScatterGather` at the `Master` role. Since `Master` is a non-parametric participant, Rule 2 and 1 are applied to get Line 2 and 3 respectively. This results in a protocol body without conditional interactions.

```
1  local protocol ScatterGather at Master(role Master, role Worker[1..N], group
       Workers={Worker[1..N]}) {
2    Scatter(int) to Workers;
3    Gather(int) from Workers;
4  }
```

Listing 4.2: Master endpoint from ScatterGather protocol.

The local protocol of `Worker` for `ScatterGather` is similarly derived by applying the projection rules. Since `Workers` is a group role and a subset of `Worker[1..N]`, Rule 6 and 7 are applied to get Line 2 and 3.

```
1  local protocol ScatterGather at Worker[1..N](role Master, role Worker[1..N],
       group Workers={Worker[1..N]}) {
2    if Workers Scatter(int) from Master;
3    if Workers Gather(int) to Master;
}
```

Listing 4.3: Worker endpoint from ScatterGather protocol.

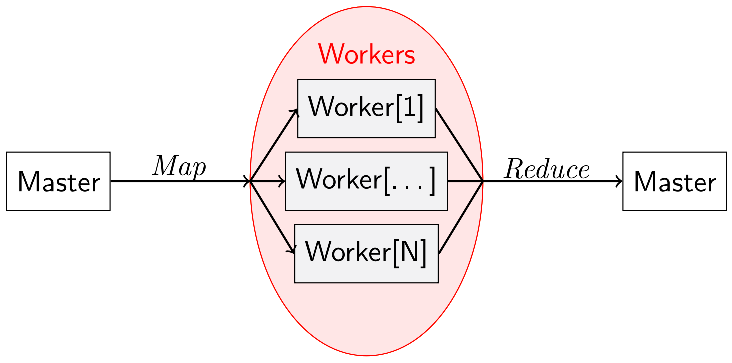


Figure 4.7.: Topology of the ScatterGather protocol.

4.4.3. Use Case: Web Services

Pabble is inspired by applications in the domain of parallel programming, but the parametric nature of Pabble as a protocol language allows us to express interactions with more flexibility while keeping the protocols succinct.

Quote-Request protocol specification (C-UC-002) is the most complex use case in [WSC] published by W3C Web Services Choreography Working Group [CDL].

```
global protocol WebService (role Buyer, role Supplier[1..S], role Manufacturer
    [1..M]) {
  Quote() from Buyer to Supplier[1..S];
  rec RENEGOTIATE_MANUFACTURER {
    foreach (j:1..M) {
      Item() from Supplier[i:1..S] to Manufacturer[j];
      Quote() from Manufacturer[j] to Supplier[1..S];
    }
    // Gather
    Quote() from Supplier[1..S] to Buyer;
    foreach (i:1..S) { // (3)
      rec RETRY_NEGOTIATION {
        choice at Buyer {
          // Buyer accepts quote and place orders (4a)
          ok() from Buyer to Supplier[i];
        } or {
          // Buyer modifies quotes and send back to supplier (4b)
          modify(Quote) from Buyer to Supplier[i];
          choice at Supplier[i] {
            // Supplier agrees
            // to modified quote (5a)
            ok() from Supplier[i] to Buyer;
          } or {
            // Supplier modifies quote again (5b)
            retry(Quote) from Supplier[i] to Buyer;
            // Retry Supplier[i]-Buyer negotiation
            continue RETRY_NEGOTIATION;
          } or {
            // Reject (5c)
            reject() from Supplier[i] to Buyer;
          } or {
            // Supplier renegotiate with Manufacturers for quote (5d)
            renegotiate() from Supplier[i] to Buyer;
            continue RENEGOTIATE_MANUFACTURER;
          }
        }
      } // Try NEXTSUPPLIER
    }
} }
```

Listing 4.4: Pabble Example: Web Services Use Case

```
local protocol WebService at Buyer (role Supplier[1..S], role Manufacturer[1..M
    ]) {
  Quote() to Supplier[1..S];
  rec RENEGOTIATE_MANUFACTURER {
    Quote() from Supplier[1..S];
    foreach (i:1..S) {
      rec RETRY_NEGOTIATION {
        choice at Buyer {
          ok() to Supplier[i];
        } or {
          modify(quoteType) to Supplier[i];
          choice at Supplier[i] {
            ok() from Supplier[i];
          } or {
            retry(quoteType) from Supplier[i];
            continue RETRY_NEGOTIATION;
          } or {
            reject() from Supplier[i];
          } or {
            renegotiate() from Supplier[i];
            continue RENEGOTIATE_MANUFACTURER;
          }
        } // choice at Buyer
      }
    }
} }
```

Listing 4.5: Pabble Example: Buyer Endpoint of Web Service Use Case.

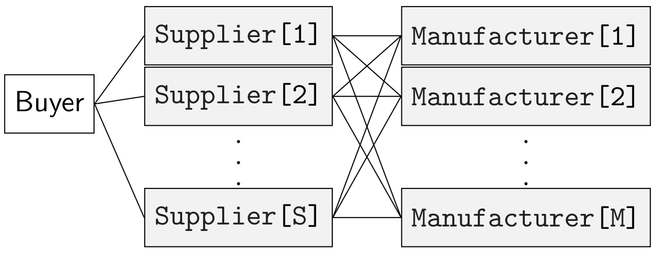


Figure 4.8.: Web Services Quote-Request interaction.

It describes the interaction between a buyer who interacts with multiple suppliers who in turn interact with multiple manufacturers in order to get a quote for some goods or services.

The basic steps of the interaction is as follows:

1. A buyer requests a quote from a set of suppliers
2. All suppliers forward the quote request of the items to their manufacturers
3. The suppliers interact with their manufacturers to build the quotes for the buyer, which is then sent back to the buyer
4. a) Either the buyer agrees with the quotes and place the orders
   b) Or the buyer modify the quote and send back to the suppliers
5. In the case the supplier received an updated quote request (4b)
   a) Either the supplier responds to updated quote request by agreeing to it and sending a confirmation message back to buyer
   b) Or the supplier responds to the updated quote request by modifying it and sending back to buyer and the buyer goes back to step 4
   c) Or the supplier responds to the updated quote request by rejecting it
   d) Or the supplier renegotiates with the manufacturers, in which case we return to step 3

Figure 4.8 shows the interactions between different components in the Quote-Request use case. We set the generic number `S` for Suppliers and `M` for

Manufacturers. The interactions are described as a Pabble global protocol in Listing 4.4.

In the protocol, we omitted the implicit `requestIdType` from the payload type in all of the messages which keeps track of states of each role in the stateless web transport. The `Buyer` initiates the quote request on Line 2, when it broadcasts a `Quote()` message to all `Suppliers`. Then on Line 4–7 each of the `Supps` forward the quote requests to their respective `Manufacturers`, and get a reply from each of them by a series of gather and scatter interactions. Next, the `Suppliers` reply to the `Buyer` on Line 9, and the `Buyer` then decides between accepting the offer straight away (Line 14, outcome 4a), or sending a modified quote request (Line 17, outcome 4b). If a `Supp` received a modified quote, it decides between accepting the modified quote (Line 21, outcome 5a), rejecting the modified quote straight away (Line 29, outcome 5c) or modifying the quote and renegotiating with `Buyer` (Line 24, outcome 5b). It is also possible that the `Supplier` renegotiates with its `Manufacturers` again, so it notifies the `Buyer` and returns back to the initial negotiation phase (Line 32, outcome 5d). The projected endpoint protocol for `Buyer` is Listing 4.5.

4.4.4. Use Case: RPC Composition

We present a use case from the Ocean Observatories Initiative project [OOI]. The use case describes a high-level Remote Procedure Call (RPC) request/response protocol between layers of proxy services. An application sends a request to a high-level service, and the service is expected to reply to the application with a result. If the service does not provide the requested service, then this high-level service will issue a request to a lower level service which can process the request. This request-response protocol is chained between services in each level until a low-level service is reached.

Figure 4.9 describes the chaining of RPC-style request/response protocol. A request is routed to the most relevant service provider through multiple proxy services, hidden from higher level services. The request routes through a multi-hop path from the requester to the resources. The reply is routed in reverse through the same participant proxy services back to the requester.

We represent this series of interactions using a Pabble protocol outlined below. The set of participants, `Service[1..N]`, represents a proxy service

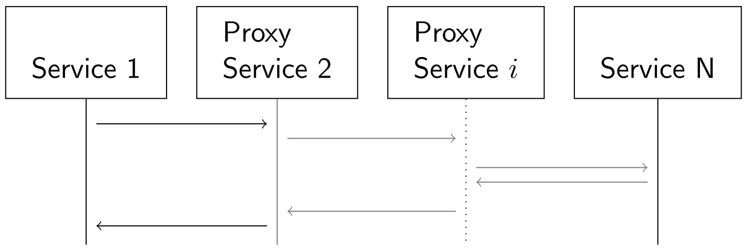


Figure 4.9.: RPC request/response chaining.

in each of the levels. `Service[1]` is the requester and `Service[N]` is the actual service provider. A `Request()` message is sent from a `Service` to the `Service` in the level directly below, until it reached `Service[N]` which will process the request and reply to the higher level service with a `Response()`. Using a `foreach` loop with decrementing indices, the `Response()` is cascaded to the originating service, `Service[1]`. The Pabble protocol is shown in Listing 4.6.

```
global protocol RPCChaining(role Service[1..N]) {
  foreach (i:1..N-1) {
    Request() from Service[i] to Service[i+1];
  }
  // Request() processed by Service[N] to give Response()
  foreach (i:N..2) {
    Response() from Service[i] to Service[i-1];
  }
}
```

Listing 4.6: Pabble Example: RPC request/response chaining

As the request and response phase are symmetric and involve the same participants, we are able to compact the multi-layer protocol to only using two `foreach` loops, each with one parameterised interaction statement. `N` can be an arbitrary constant to allow maximum flexibility in the protocol. This simple and concise representation of thep complex RPC chaining protocol is possible because of the index notation in Pabble.

4.4.5. Implementation Example: Linear Equation Solver

Listing 4.9 shows an example implementation outline for a linear equation solver using a wraparound mesh, which follows the Pabble protocol in Listing 4.7. The topology is illustrated in Figure 4.10, and is similar to the non-parametric version presented in Section 3.4.2 The example is given in Message-Passing Interface (MPI), the standardised API for developing message-passing applications in parallel computing.

```
1  global protocol Solver(role W[1..N][1..N], group Col={W[1..N][1]}) {
2    rec CONVERGE {
3      Ring(double) from W[i:1..N][j:1..N-1] to W[i][j+1];
4      Ring(double) from W[i:1..N][N] to W[i][1];
5
6      // Vertical propagation - Group-to-Group
7      (double) from Col to Col;
8      continue CONVERGE;
9    }
10 }
```

Listing 4.7: Pabble Example: Linear Equation Solver.

The protocol above describes a wraparound mesh that performs a ring propagation between `W` (for worker) in the same row (Line 3–4), and the result of each `W` row is distributed to all `W`s in the first column (i.e. `W[*][1]`) using a group-to-group distribution on Line 7. The global protocol is then automatically projected into its local protocol shown in Listing 4.8 below. Developers can then implement the application using its local protocol as a guide.

```
local protocol Solver at W(role W[1..N][1..N], group Col={ W[1..N][1] }) {
  rec CONVERGE {
    if W[i:1..N][j:2..N] Ring(double) from W[i][j-1];
    if W[i:1..N][j:1..N-1] Ring(double) to W[i][j+1];
    if W[i:1..N][1] Ring(double) from W[i][N];
    if W[i:1..N][N] Ring(double) to W[i][1];

    // Vertical propagation - Group-to-Group
    if Col (double) from Col;
    if Col (double) to Col;
    continue CONVERGE;
  }
}
```

Listing 4.8: Pabble Example: Linear Equation Solver local protocol.

Note the similarity of the local protocol and the structure of the MPI implementation in Listing 4.9. In particular, the conditional send and receive in MPI can directly correspond to the role conditions in the local protocol which was derived from the global protocol by projection.

```
MPI_Init(&argc, &argv); // Start of protocol
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Process ID
MPI_Comm_size(MPI_COMM_WORLD, &size); // # of Process
MPI_Comm Col; int N = (int)sqrt(size);
```

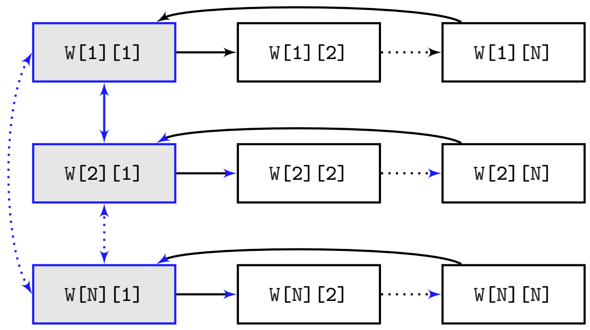


Figure 4.10.: N^2-node wraparound mesh topology.

```
...
/* Calculate condition for W[i:1..N][j:2..N] */
if (2 <= rank%N+1 && rank%N+1 <= N)
 MPI_Recv(buf, cnt, MPI_DOUBLE, rank-1/*W[i][j-1]*/, Ring, MPI_COMM_WORLD);
/* Calculate condition for W[i:1..N][j:1..N-1] */
if (1 <= rank%N+1 && rank%N+1 <= N-1)
 MPI_Send(buf, cnt, MPI_DOUBLE, rank+1/*W[i][j+1]*/, Ring, MPI_COMM_WORLD);
/* Calculate condition for W[i:2..N][j:1..N] */
if (2 <= rank/N+1 && rank/N+1 <= N)
 MPI_Send(buf, cnt, MPI_DOUBLE, rank-N*1/*W[i-1][j]*/, Ring, MPI_COMM_WORLD);
/* Calculate condition for W[i:1..N-1][j:1..N] */
if (1 <= rank/N+1 && rank/N+1 <= N-1)
 MPI_Send(buf, cnt, MPI_DOUBLE, rank+N*1/*W[i+1][j]*/, Ring, MPI_COMM_WORLD);

/* Distribute vertically: Group-to-Group on 'Col' group communicator */
if (rank%N+1 == 1)
 MPI_Allgather(buf_col, cnt_col, MPI_DOUBLE,
                buf_col, cnt_col, MPI_DOUBLE, Col);
...
MPI_Finalize(); // End of protocol
```

Listing 4.9: MPI implementation for Solver protocol

4.5. Summary and Discussion

In this chapter we presented Pabble, a parameterised protocol description language based on Scribble and parameterised MPST. We show that the language is able to capture communication patterns that can scale over the number of participants in a compact and concise syntax. Despite the undecidability in its underlying parameterised MPST theory, Pabble overcomes the shortcoming of the theory by using integer indices which also add practicality to the language. We developed theory and a tool to ensure that Pabble protocols are safe even when they scale up to an unbounded number of processes. We conclude the chapter by demonstrating the expressiveness of Pabble through a number of examples which would be difficult to recreate in Scribble, and that Pabble is a suitable protocol abstraction for practical and scalable parallel applications.

As Pabble is derived from the static typing system of parameterised MPST, there are certain limitations as to what Pabble protocols can express. As with Scribble used in Session C, Pabble cannot express data-defined communication structures – communication structures that are defined by data available at runtime. Another pattern that Pabble cannot effectively express is data-dependent structures. The most common example is loops written with `rec`, where the loop conditions are dependent on input data, and do not guarantee that loops across endpoints defined in the same `rec` in the global protocol terminate together. However, `rec` can represent the *structure* of communication (i.e. iteration), which is important for specifying structured protocols.

Recall that in the previous chapter we mentioned in Session C, for each endpoint protocol, a separate endpoint has to be implemented. With Pabble, these endpoint protocols are grouped into a single parameterised endpoint, and we need to associate these parameterised endpoints with implementation. In the next chapter, we discuss how we can tackle this with case studies to show how our proposed Pabble-based approaches work.

5 Scalable High-performance Session Programming

Overview This chapter presents a number of case studies of using Pabble for describing scalable communication protocols for parallel programming. The case studies show that Pabble protocols are sufficiently expressive to represent core patterns of parallel programming described as Dwarfs [AWW$^+$09]. We compare implementations of the scalable Pabble protocols based on Session C and MPI, results show that using MPI as a target runtime for scalable application performs and scales better than the Session C runtime, and hence motivates the use of MPI as standard session communication primitives for parallel programming in the next chapter.

5.1. Introduction

This thesis focuses on applying the theory of Session Types, which can automatically ensure deadlock-freedom and communication-safety, i.e. matching communication pairs, for message-passing parallel programs. There are two general approaches to applying Session Types to parallel programming: type checking and code generation.

In Chapter 3 we have presented a session-based programming framework Session C which uses the type checking approach. However, since Session C uses Scribble to specify protocols, Session C applications are difficult to scale. In Chapter 4 we presented a parametric extension of the Scribble language, called Pabble, which addresses this limitation. We have a choice of type checking parameterised parallel programs with Pabble, or generate safe parallel programs from Pabble. In both cases, given Pabble, as a new way of modelling *scalable* parallel programs, we need a runtime or a set of programming primitives that are more suitable to be use in conjunction with Pabble.

In this chapter, we present a number of case studies representing core patterns of parallel programming from Dwarfs [AWW+09]. We aim to demonstrate how our protocols can help abstract core communication patterns of parallel programming, and how they can be implemented. We also present an analysis of runtime, comparing Session C runtime and MPI, and discuss which is more suitable for scalable session-based parallel programming.

5.2. Case Studies

This section presents case studies of using Scribble protocols in parallel programming. All of these examples are representative patterns from common parallel patterns known as Dwarfs [AWW+09]. The Dwarf evaluation metric was proposed as a collection of high-level, core parallel communication patterns from important scientific and engineering methods. Each of these patterns is called a *Dwarf*, which represents one category of patterns and covers a broad range of concrete algorithm implementations. Dwarfs are used to evaluate our session-based protocol language and our programming methodology because they are not language or optimisation specific, being able to express the Dwarfs confirms that our approach is general enough to be extended to more practical use cases.

We have chosen N-body simulation, an example of particle methods dwarf, dense matrix-vector multiplication, a dense linear algebra dwarf, and sparse matrix-vector multiplication, a sparse linear algebra dwarf, to show how Scribble and MPI can be used together for parallel programming from either of our two session-based approaches.

5.2.1. N-body Simulation

We implemented a 2-dimension N-body simulation using a ring topology, using the parameterised `Ring` protocol in Section 4.1.

Using the local `Ring` protocol projected with respect to `Worker`, we may implement the parallel program in MPI as shown in Listing 5.1.

```
// Protocol defines Worker[1..N]
// MPI ranks are 0-based and maps to 0..N-1
```

```
while (i++<n) {
  if (1<=rank && rank<=N-1) MPI_Recv(rbuf, count, rank-1, MPI_INT, ..);
// (Sub-compute) Send received data to process ..
  if (0<=rank && rank<=N-2) MPI_Send(sbuf, count, rank+1, MPI_INT, ..);
  if (rank==0) MPI_Recv(rbuf, count, N-1, MPI_INT, ..);
// (Sub-compute) Send received data to process ..
  if (rank==N-1) MPI_Send(sbuf, count, 0, MPI_INT, ..); }
// Perform global update after round
```

Listing 5.1: MPI implementation of Worker endpoint.

In MPI, all processes share the same source code and compiled program file, and they are only distinguished at runtime by their assigned process id. The process id is stored in the `rank` variable, and is available throughout the program to calculate participants addresses. In the above MPI code, `MPI_Send` and `MPI_Recv` are the primitives in the MPI library to send and receive data, and all the lines are guarded by a rank check. The variables `sbuf` and `rbuf` stand for send buffer and receive buffer respectively, and `count` is the number of elements to send/receive (i.e. array size); `MPI_INT` is an MPI defined macro to indicate the data being sent/received is of type `int`.

The ring topology above is a simple yet powerful topology to distribute data between multiple participants in small chunks. This allows more sub-computation and will potentially allow more overlapping between communication and computation.

A Scribble protocol contains the interaction patterns (i.e. the session typing) for a set of participants. It contains sufficient information to generate the MPI code shown above.

5.2.2. K-means clustering

K-means clustering is an algorithm for grouping a set of objects into k clusters. Initially, k centres of clusters are chosen randomly. Each object will be assigned to a cluster based on their proximity to the nearest centre. After each iteration of the assignment, the centre of the clusters will be recalculated by taking the mean of all objects belonging to that cluster. The whole process will be repeated until the clusters stabilise or reach a pre-determined number of iteration steps. In our implementation, the assignment is parallelised and computed in parallel, and the resulting clusters are distributed between all participants, so that the centres of the clusters can be calculated on each of the participants for the next iteration.

Below is the protocol specification of one of our participants, `Master`, of our K-means clustering implementation.

```
protocol Kmeans at Master(role Worker[1..3], group Workers={Workers[1..3]}) {
  rec STEP {
    // Multicast to Worker[1], Worker[2], Worker[3]
    distribute(int) to Workers;
    // Multi-receive from Worker[1] Worker[2] Worker[3]
    gather(int) from Workers;
    continue STEP;
  }
}
```

Listing 5.2: Protocol of the K-means clustering

The protocol above is a modified `ScatterGather` protocol in Section 4.4.2, where `N` is 3 and `Kmeans` protocol has an extra outer loop.

The implementation of the algorithm in MPI and Session C is listed below:

```
// Master role.
for (int i=0; i<STEPS; i++) {
  kmeans_compute(range_start, range_end);
  // Multicast to Worker1 Worker2 Worker3
  msend_int(centres_master, chunk_sz, 3, Worker1, Worker2, Worker3);

  // ... Update centres with local results

  // Multi-receive from Worker1, Worker2, Worker3
  mrecv_int(centres_wkr, &sz, 3, Worker1, Worker2, Worker3);

  // ... Update centres with remote results
}
```

Listing 5.3: Implementation of K-means clustering with Session C.

In the code above, `msend_int` and `mrecv_int` are the *variadic* primitives for multicast send and multi-receive respectively from the Session C runtime library. The first parameter is the pointer to the data to be sent, followed by the size of the data and the total number of participants to be sent to or received from. `Worker1`, `Worker2`, `Worker3` are the participant identifiers which this participant is communicating with. The variable `chunk_sz` holds the size of the partition to be `msend` to each participant; and `sz` will contain the total number of bytes received by `mrecv`.

The scatter-gather pattern utilised by the above distributes the local results of partitioned computation to other `Workers`, then receives the results from them. At the end of the loop, all participants in the computation will have the complete set of centres from all other participants.

```c
#define MASTER 0 // Master has MPI rank 0
for (int i=0; i<STEPS; i++) {
  kmeans_compute_fpga(range_start, range_end);
  MPI_Scatter(centres_master, chunk_sz, MPI_INT, centres, chunk_sz, MASTER,
       MPI_COMM_WORLD);
  MPI_Gather(centres, chunk_sz, MPI_INT, centres_master, chunk_sz, MASTER,
       MPI_COMM_WORLD);
}
```

Listing 5.4: Implementation of K-means clustering with MPI.

The MPI version of the code (Listing 5.4) makes use of the MPI collective operation primitives `MPI_Scatter` and `MPI_Gather` which are for scatter and gather operations respectively. On the other hand, the asynchronous communication primitives of Session C allows a partial overlap of communications with the process of updating centres to reduce the execution time. This fine grained control is backed by the session type checking process, ensuring the communications with a partial overlap are deadlock-free.

5.2.3. Dense matrix-vector multiplication

Dense matrix-vector multiplication takes a $M \times N$ matrix and multiply it by a N dimensional vector to get a N dimensional vector result. The multiplication can be parallelised by partitioning the input matrix to N segments by row-wise block striping shown in Figure 5.1 and distributed to N processes. Each process gets a copy of the vector, and each element in the vector can be calculated by the processes in parallel.

Listing 5.5 shows a protocol for our dense matrix-vector multiplication. The `Worker[0]` is the coordinator which distributes the partitions to each `Worker`. The primitive `foreach (i:1..N){ }` is a foreach-loop, which iterates from `1` to `N` using the index variable `i`. Inside the `foreach`, `Worker[0]` sends the offset and length of the partitions to each `Worker` (Line 4 and 5) respectively, followed by the actual matrix elements (Line 6). Vector B, which is of size N, is broadcasted to all processes by the coordinator on Line 9. Finally, the results of each `Worker` are gathered by the coordinator and combined to get the result of the matrix multiplication (Line 14).

```
1  global protocol DenseMatVec(role Worker[0..N]){
2    // Scatter Matrix A
```

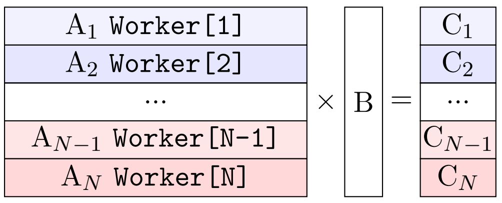


Figure 5.1.: Partitioning of input matrix.

```
  foreach (i:1..N) {
    LBound(int) from Worker[0] to Worker[i];
    UBound(int) from Worker[0] to Worker[i];
    Data(double) from Worker[0] to Worker[i];
  }
  // Scatter Vector B
  (double) from Worker[0] to Worker[1..N];

  // --- Perform calculation ---

  // Gather data
  (double) from Worker[1..N] to Worker[0];
}
```

Listing 5.5: Global protocol of dense matrix-vector multiplication.

An MPI implementation following the above protocol has the code structure shown below. In the initial phase of the calculation, the coordinator, the process of rank 0 (Line 5–17), uses a `for` loop to iterate through the worker process ids (processes with ranks above 0, up to the total number of processes `size`) and calculates the `lbound` and `ubound` for each of the participants, where `lbound` is the first row of the partition, and `ubound` is the last. The partition is then sent to the corresponding `Worker[i]`. Other `Worker` processes receive the values and store locally.

This is followed by a broadcast on Line 25 using an `MPI_Bcast` with root `Worker[0]` for the workers to receive the input vector. A partial result, `C`, is then calculated on each worker, and the result collected by the coordinator using `MPI_Gather`. `MPI_Gather` collects the partial results, then combines them in the `Result` N dimensional array.

The implementation shows how our session protocol descriptions can also correspond to collective operations, such as `(double) from Worker[0] to Worker[1..N]` and `MPI_Bcast`, or `(double) from Worker[1..N] to Worker[0]` and `MPI_Gather`.

```
double A[A_ROWS][A_COLS]; // Matrix A
double B[B_COLS]; // Vector B
double C[B_COLS]; // Partial result
...
if (rank == 0) {
  for (i = 1; i < size; i++) { // Calculate then send to each Worker
    // Calculate LowerBound and UpperBound for each Worker
    lbound = (i - 1) * partition_size;
    ubound = lbound + partion_size;

    MPI_Send(&lbound, 1, MPI_INT, Worker[i], LBound, ...);
    MPI_Send(&ubound, 1, MPI_INT, Worker[i], UBound, ...);

    // Send partition of matrix A
    MPI_Send(&A[lbound][0], (ubound-lbound) * A_COLS, MPI_DOUBLE, Worker[i],
        Data, ...);
  }
} else if (rank > 0) { // Workers, receiving work
  MPI_Recv(&lbound, 1, MPI_INT, Worker[0], LBound, ...);
  MPI_Recv(&ubound, 1, MPI_INT, Worker[0], UBound, ...);

  MPI_Recv(&A[lbound][0], (ubound-lbound) * A_COLS, MPI_DOUBLE, Worker[0], Data,
        ...);
}

// All Workers receive the vector B
MPI_Bcast(&B, B_ROWS, MPI_DOUBLE, Worker[0], ...);
...
// Calculate matrix multiplication
mat_vec_mul(A, B, lbound, ubound, C);
...
// ... Gather results to Worker[0] ...
MPI_Gather(C, 1, MPI_DOUBLE, Result, 1, MPI_DOUBLE, Worker[0], ...);
```

Listing 5.6: MPI implementation of dense matrix-vector multiplication.

5.2.4. Sparse matrix-vector multiplication

Finally we show an implementation of a direct sparse matrix-vector multiplication. Sparse matrices are often used for data representation that are too large to fit in memory as an array, but the content is sparse and can be efficiently compressed to a more compact format. Our implementation uses a $M \times N$ sparse matrix input stored in a compressed sparse row (CSR) format, where the data are represented by three arrays.

- `vals`: a contiguous array containing all values of the sparse matrix in a left-to-right, top-to-bottom order. This compact storage of the matrix skips all empty (or zero) cells in the matrix and only contains cells with a value.

- `row_ptr`: an array containing indices for the `vals` array, each element contains the accumulated total of elements in each row. For example, `[1, 3, 4, 8]` means that row 0 has 1 element, row 1 has 2 elements, row 2 has 1 element and row 3 has 4 elements. This array has the same size as the total number of rows.

- `col_ind`: the column indices for each of the values in `vals`. This array has the same size as `vals`.

The three arrays combined are sufficient to represent a sparse matrix, or a partition of the sparse matrix.

The protocol to perform a sparse matrix-vector multiplication is shown in Listing 5.7. In the protocol, the partitioned matrix rows in CSR format are sent to each worker as separate row, col and values arrays (Line 3, 4 and 5). The N dimensional vector is then sent to all workers. The results of the calculation by each `Worker` are sent back to `Worker[0]` (Line 8).

```
1  global protocol SparseMatVec(role PE[0..N]) {
2    /* Distribute data */
3    (int) from W[0] to W[1..N]; // row_ptr
4    (int) from W[0] to W[1..N]; // col_ind
5    (double) from W[0] to W[1..N]; // vals
6    (double) from W[0] to W[1..N]; // vector
7    /* Output vector */
8    (double) from W[1..N] to W[0];
9  }
```

Listing 5.7: Global protocol of sparse matrix-vector multiplication.

A corresponding implementation for the above protocol may look like the MPI code below:

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
int nr_of_rows = MATRIX_ROWS/size;
...
MPI_Scatter(row_ptr, nr_of_rows, MPI_INT, ..);
...
// calculate number of indices for each process
...
MPI_Scatterv(col_ind, nr_of_elems, MPI_INT, ...);
MPI_Scatterv(vals, nr_of_elems, MPI_DOUBLE, ...);
...
MPI_Bcast(vector, MATRIX_ROWS, MPI_DOUBLE, Worker[0], ...); // Distribute vector
```

```
12 ...
13 // Calculate matrix multiplication
14 mat_vec_mul(row_ptr, col_ind, vals, vector, C);
15 ...
16 MPI_Gather(C, 1, MPI_DOUBLE, Result, 1, MPI_DOUBLE, Worker[0], ...);
```

Listing 5.8: MPI implementation of sparse matrix-vector multiplication.

Each process starts by calculating the expected number of rows it will be owner of, and we assume that the number of rows for each process is the same and the total number of rows can divide exactly by the total number of processes. Next we use `MPI_Scatter` to distribute segments of the `row_ptr` array to each worker process, which sends segments of a given input memory to other processes based on their rank and the segment position in the memory (Line 4).

`nr_of_elems` is an array containing the number of elements to be sent to each worker. Since in a sparse matrix the number of elements in each row is not fixed, the `nr_of_elements` array contains the number of matrix elements each worker receives. The indices of the array correspond to the MPI rank of the workers and the column index `col_ind` is distributed to each worker process by `MPI_Scatterv` (Line 8), a variant of the `MPI_Scatter`, where the `v` stands for variable size as opposed to fixed size in `MPI_Scatter`. Similarly, the actual matrix element values are distributed to all workers by a call to `MPI_Scatterv` on Line 9, using the same `nr_of_elems` to specify the number of elements for each worker.

Once the workers have received the matrix partitions, the coordinator distributes the N dimensional vector by `MPI_Bcast` to all workers to perform the matrix-vector calculation for the rows of the sparse matrix each processor has.

Finally, as in the dense matrix-vector multiplication example, the results are collected by the root worker `Worker[0]` using a `MPI_Gather`. In this implementation, we use exclusively collective operations to distribute and collect results as it is more efficient with the CSR data format. Notice that the protocol does not distinguish between different modes of `MPI_Scatter`, in particular, the Scribble statement `(int) Worker[0] to Worker[1..N ];` corresponds to both `MPI_Scatter` and `MPI_Scatterv`. Hence a single protocol statement can map to multiple implementations, and without external information about the implementation, a code generation tool cannot choose a suitable implementation, and this use case is more suitable for our type checking approach.

5.3. Runtime Analysis

The following experiment compares the performance of Session C against MPI, as we scale up the number of processes to determine which runtime is more suitable to be used with Pabble for developing a parallel program. We expect the result to be similar to those presented in Section 3.5 where we evaluated the performance of the Session C runtime and MPI but in a fixed topology.

As a contrast to the experiments presented in Section 3.5, we compare the performance of Session C and MPI given scalable, parameterised Pabble protocols. Since the type checker in Session C does not work with parameterised protocols, for this evaluation we use a modified type checker which parses and accepts Pabble protocols as specifications. With this modification, the same protocol can be use to type check Session C endpoint code for different number of spawned processes, allowing a limited amount of parameterisation. For the purpose of the experiment, the type checking in this experiment is only needed for completeness; we are focusing on the runtime performance rather than the safety guarantees, and we use the same protocol to ensure that both implementations follow the same communication specification to have a fair comparison.

Environment. The experiments were taken on a cluster with multiple nodes with AMD PhenomX4 9650 Quad-Core @ 2.30GHz CPUs and 8GB DDR2 RAM each, connected by a dedicated Gigabyte Ethernet switch. All implementations were compiled with gcc 4.4.3 with the optimisation level `-O3`. For the MPI versions, Open MPI 1.4.3 were used.

We evaluate the scalability of our implementations by the performance of Session C and MPI with the N-body simulation and the K-means clustering algorithm and the parallel which uses a ring and a scatter-gather topology respectively, described in Section 5.2 earlier in this chapter. The reported runtime for 1 node is the serial execution of the implementations which is identical in both Session C and MPI version since they share the same code for the main computation. The results in Figure 5.2 shows that the

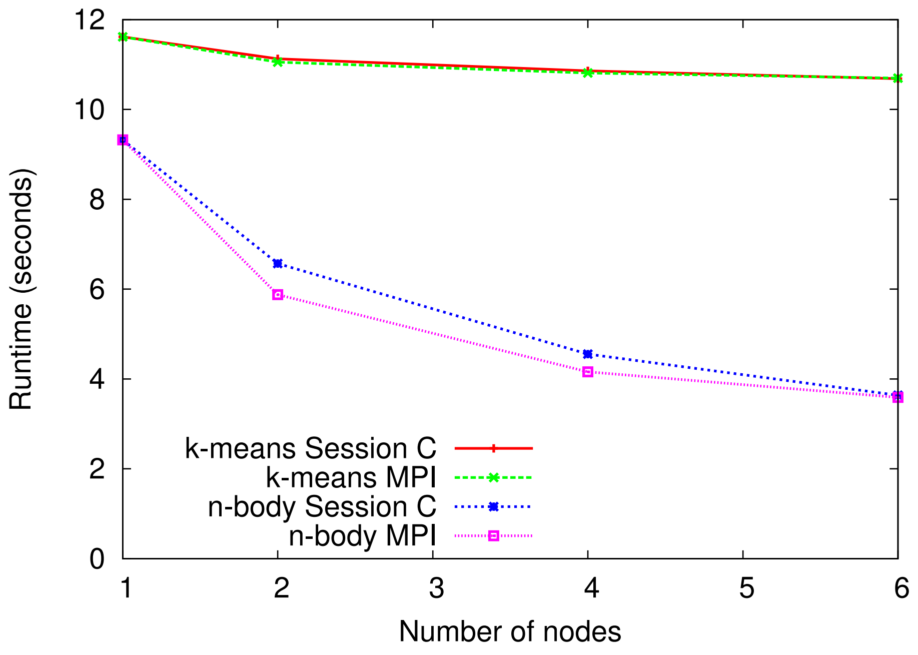


Figure 5.2.: Comparing Session C and MPI performance for different number of spawned processes from the same protocol, testing how the two runtimes scale .

implementations in MPI are marginally faster than the Session C implementations, which agrees with our previous evaluation of the Session C runtime in Section 3.4. From this result we deduce that MPI is has a better performance overall than Session C runtime.

Challenges A valid question to ask is, why is MPI not chosen as the runtime library for Session C in Chapter 3? Session C's type checker is designed for Scribble, a non-parameterised protocol language, and the Session C runtime is a simple session programming API. We showed that the type checking approach is effective, and based on the approach we are able to give strong guarantees about the properties of the program (communication safety, deadlock freedom) because we have control over the usage of the API. For example, `outbranch` and `inbranch` is reliant on using the primitives in a specific pattern in a C program, which we cannot implement in MPI without working around the well-defined primitives. We also have the prospect of outperforming MPI in some cases if we built the Session C API on top of a different library, instead of building on MPI directly. However, we face a number of challenges when building a type checker using the same methodology for Pabble, which is a dependent protocol language and MPI, which is a standard parameterised implementation API. The Pabble language with its well-formedness checks reduces the undecidability issues in the role representation by using integer instead of general indices. The type checking process will compare the protocol against a simplified, canonical local protocol extracted from the implementation, which still posts a challenge in the process of protocol extraction. In particular, inferring source and destination processes from parametric source code is non-trivial. MPI uses process IDs (or ranks) to identify processes, and it is valid to perform numeric operations on the ranks to efficiently calculate target processes. This allows ways of exploiting the C language features while remaining a valid program. For example, instead of using a conventional conditional statement, an MPI function call of this form may be used:

```
MPI_Send(buf, cnt, MPI_INT, rank%2? rank+1: rank-1, ...)
```

where the process ID, `rank`, is being used as a boolean, thus a straightforward analysis of `rank` usages would not be sufficient. In order to correctly calculate target processes of the interactions, it will be necessary to simulate rank calculations by techniques such as symbolic execution or combinations of runtime techniques.

5.4. Summary and Discussion

In this chapter we reviewed a number of Pabble protocols and their possible implementations following the protocol specifications. Furthermore, we compared how the Session C runtime and MPI perform when the same program is scaled up to multiple processes. The results show that MPI is more suitable to be used with Pabble because of the structure of MPI programs. We discussed why type checking MPI against Pabble is a difficult task, and in the next chapter we introduce a code generation framework which is an alternative method of applying Pabble on parallel programming in form of a *code generation* framework to ensure communication safety and deadlock freedom.

6 Safe MPI Code Generation with Pabble

Overview This chapter presents a code generation workflow for type-safe and deadlock-free Message-Passing Interface (MPI) programs. The generation starts from designing a global topology with a protocol specification language based on Parameterised Multiparty Session Types (MPST). An MPI parallel program with backbone is then automatically generated by projecting a global specification. This MPI backbone is merged with sequential computations written in C by aspect-oriented compilation, resulting in a complete MPI program.

6.1. Introduction

Parallel programming with the MPI library is a well-documented difficult task, and communication mismatches are the most common MPI errors by users [DKdS+05]. Reasoning about interactions between distributed processes is difficult at scale, and despite the advances in novel techniques and models such as Partitioned Global Address Space (PGAS) for simplifying parallel programming, MPI is still widely used by the scientific community and will be here to stay in the foreseeable future. In this chapter we follow up on previous chapter's findings, and approach session-based parallel programming using MPI code generation. We generate MPI applications, combining Pabble, a language-independent interaction protocol and sequential

This is a collaboration work with José Gabriel de Figueiredo Coutinho, who implemented the LARA AOP directives and Word Counter and AdPredictor benchmarks using the workflow.

code kernels. We target MPI because it is an efficient runtime, and is a established standard, which is being used as a common interface for different kinds of programming models, including FPGAs [SnPM^{+}10] or stream programming [MMP10].

We also observe that most parallel applications follow a certain structural communication pattern. For example, to implement parallel solutions for iterative numerical methods, a stencil pattern may be used. The stencil pattern is a general underlying communication pattern which does not change in regards to the numerical methods being applied, and from the perspective of a parallel interaction designer, different numerical methods are simply sets of parameters that change the behaviour of the computation between communication. Recent work by Wilkinson and Ferner et al. [WVF13, FWH13] uses *pattern programming* as an approach to teach parallel programming, where they introduced Paraguin, a set of pragmas to annotate C code and transform the code into distributed parallel applications built on top of SUIF compiler system. They also use the Seeds framework, which is an environment for developing in a Java programming environment by extending given pattern classes. The work evaluates the effectiveness of the pattern programming approach for parallel programming by interviewing undergraduate students and shows that pattern programming simplifies parallel programming for novice developers. The approach is also known as algorithmic skeleton frameworks, [GVL10] surveys a number of existing tools and frameworks using the technique for high-level structured parallel programming.

Our work aims to tackle the task of pattern programming with added flexibility, practicality and safety by combining the formally founded communication safety guarantees with session types in Pabble, and MPI as the target runtime library. A Pabble protocol can capture the parallel control flow of an application, and guide the development of the application as a structured template.

Overview This chapter presents a parallel programming workflow based on Pabble. Pabble describes scalable interaction protocols, which captures the overall generic interaction pattern of the parallel application, and is used to generate an annotated MPI application backbone, specifying the interactions between parallel processes. Based on the Pabble protocol, sequential computation kernels are written in C99, using queues to pass data

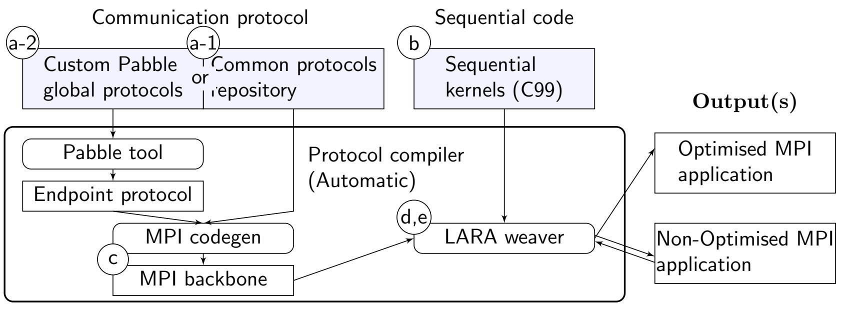


Figure 6.1.: Pabble-based code generation workflow. Shaded boxes indicate user inputs.

locally between kernels. The kernels are inserted into the MPI backbone by LARA [CCC$^+$12], an aspect-oriented compilation tool, which results in a complete MPI application. As part of the merge, LARA also performs pragma-directed optimisations on the source code to overlap communication and computation, improving the runtime performance.

Figure 6.1 shows the overview of our approach. Our approach starts from a Pabble protocol, as an abstract representation of the communication topology, or *parallel communication patterns* of a parallel application. We consider every application a coupling between sequential, computation code that defines functional behaviours of processes in the application, and a communication topology that connects the processes together as a coherent application. Hence, to build a parallel application, we first define the communication protocol, written in Pabble. A valid Pabble protocol is guaranteed free of interactions and patterns that introduce communication errors and deadlocks, moreover, Pabble protocol is designed such that it can represent protocols that scale on the number of processes at runtime similar to MPI.

Contributions Below we list the contributions of this chapter.

- A complete parallel programming workflow by capturing parallel design patterns, based on the Pabble protocol description language and sequential computation kernels (Section 6.2.2) by generating scalable MPI applications. The workflow guarantees communication safety and deadlock freedom by the formal basis of the Pabble protocol language.
- The workflow includes communication-computation merging and optimisation techniques by an aspect-oriented compilation framework

(Section 6.4.2). We show that the optimisation does not violate the communication ordering in Pabble and preserves the safety guarantees of the workflow.

- A number of case studies and performance evaluation of our framework showing the flexibility and productivity improvements of our workflow.

6.2. Application Development Workflow

6.2.1. Interaction protocols with Pabble

Our framework uses Pabble introduced in Chapter 4 to describe communication patterns. Pabble protocols provide a guarantee of communication safety and deadlock freedom between participants in the protocol; this guarantee also extends to scalable protocols, where the number of participants are not known statically, and well-formed conditions ensure that the indexing of participants does not go beyond specified bounds. A Pabble protocol describes (1) the structured message interaction patterns of the application, and (2) the control-flow elements, excluding the logic related to actual computation, so that a Pabble protocol defining a parallel design pattern can be reused for different applications (see Section 6.5.1).

Stencil Protocol

```
const N = 1..max;
global protocol Stencil(role P[1..N][1..N]) {
 rec Steps {
  LeftToRight(T) from P[r:1..N][c:1..N-1] to P[r][c+1];
  RightToLeft(T) from P[r:1..N][c:2..N] to P[r][c-1];
  UpToDown(T)   from P[r:1..N-1][c:1..N] to P[r+1][c];
  DownToUp(T)   from P[r:2..N][c:1..N] to P[r-1][c];
  continue Steps;
 }
}
```

Listing 6.1: Pabble protocol for 5-point stencil.

The full syntax of Pabble is explained in Section 4.3.1 Listing 6.1 presents an example of a Pabble protocol which defines a 5-point stencil design pattern, where $N \times N$ processes are arranged in a 2-dimensional grid, and each participant exchanges messages with its 4 neighbours (except for edge participants). This will be our running example of Pabble-based safe MPI code generation.

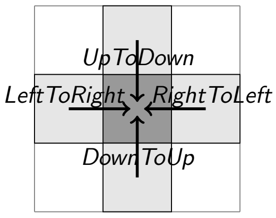


Figure 6.2.: Messages received by a process in a stencil protocol.

Our protocol body starts with a `rec` block, which stands for recursion, and is assigned with the label `Steps`. The recursion block does not specify the loop condition because a Pabble protocol only describes the interaction *structure* while implementation details are abstracted away. In the body of the recursion, we have 4 lines of interaction statements (Line 4-7), one for each direction. Interaction statements describe the sending of a message from one participant to another. For example, in Line 4 a message with label `LeftToRight` and with a generic payload type `T` is sent from `P[r:1..N][c:1..N-1]` to `P[r][c+1]`. The index expression `r:1..N` means that `r` is bound and iterated through the list of values in the range `1..N`, so the line encapsulates $N \times (N-1)$ individual interaction statements. The other interaction statements in Listing 6.1 can be similarly interpreted. Figure 6.2 shows the messages received from neighbours for participant `P[2][2]` in a 3×3 grid, which is defined in the protocol as `role P[1..3][1..3]`.

Protocol repository. To simplify application development in our framework, we provide a repository of common Pabble protocols describing common interaction patterns used by parallel applications. The `Stencil` protocol in Listing 6.1 is one example, and the other patterns in the repository include *ring pipeline*, *scatter-gather*, *master-worker* and *all-to-all.*

6.2.2. Computation kernels

Computation kernels are C functions that describe the algorithmic behaviour of the application. Each message interaction defined in Pabble (e.g. `Label (T)from Sender to Receiver`) can be associated to a kernel by its label (e.g. `Label`).

Figure 6.3 shows how kernels are invoked in a message-passing statement

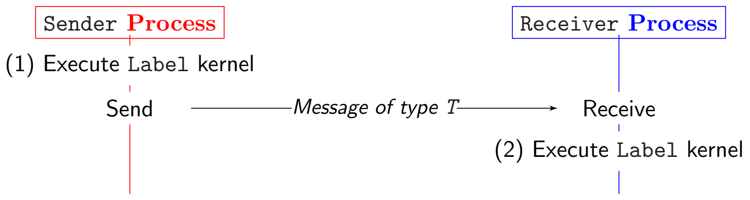


Figure 6.3.: Global view of `Label(T)from Sender to Receiver;`.

between two processes named `Sender` and `Receiver` respectively. Since a message interaction statement involves two participants (e.g. `Sender` and `Receiver`), the kernel serves two purposes: (1) produce a message for sending and (2) consume a message after it has been received. The two parts of the kernel are defined in the same function, but runs on the sending process and the receiving process respectively. The kernels are top-level functions and do not send or receive messages directly through MPI calls. Instead, messages are passed between kernels and the MPI backbone (derived from the Pabble protocol) via a queue API: in order to send a message, the producer kernel (e.g. (1)) of the sending process enqueues the message to its send queue; and a received message can be accessed by a consumer kernel (e.g. (2)), dequeuing from its receive queue. This allows the decoupling between computation (as defined by the kernels) and communication (as described in the MPI backbone).

Writing a kernel. We now explain how a user writes a kernel file, which contains the set of kernel functions related to a Pabble protocol for an application. A minimal kernel file must define a variable `meta` of `meta_t` type, which contains the process id (i.e. `meta.pid`), total number of spawned processes (i.e. `meta.nprocs`) and a callback function that takes one parameter (message label) and returns the send/receive size of message payload (i.e. `unsigned int meta.bufsize(int label)`). The `meta.buflen` function returns the buffer size for the MPI primitives based on the label given, as a lookup table to manage the buffer sizes centrally. Process id and total number of spawned processes will be populated automatically by the backbone code generated. The kernel file includes the definitions of the kernel functions, annotated with pragmas, associating the kernels with message labels. The kernels can use file (i.e. `static`) scope variables for local data storage. Our stencil kernel file starts with the following declarations for local data and meta:

Kernel header
```c
typedef struct {
  double* values; int rows; int cols;
} local_data_t;
static local_data_t *local;

unsigned int buflen(int label) { return local->rows - 2; } // local rows - halo
     rows/cols

meta_t meta = {/*pid*/0, /*nprocs*/1, MPI_COMM_NULL, &buflen};
```

Initialisation. Most parallel applications require explicit partitioning of input data. In these cases, the programmer writes a kernel function for partitioning, such that each participant has a subset of the input data. Input data are usually partitioned with a layout similar to the layout of the participants. In our stencil example where processes are organised in a 2D grid, we partition the input data in a 2D-grid of sub-matrices. The sub-matrices are calculated for each of the process using the `meta.pid` and `meta.nprocs` which are known at runtime when the kernel functions are called. Below is an example of the main part of the initialisation function.

Kernel: Init
```c
#pragma pabble kernel Init
void init(int id, const char *filename)
{ FILE *fp = fopen(filename, "r");
  local = (local_data_t *)malloc(sizeof(local_data_t));
  local->rows = 0; local->cols = 0; local->values = NULL;
  ...
  int proc_per_row = sqrt(meta.nprocs); // Participant per row
  int proc_per_col = sqrt(meta.nprocs); // Participant per column
  int row_offset = (meta.pid / proc_per_row) * row_size; // Start row of data
  int col_offset = (meta.pid % proc_per_col) * col_size; // Start column of data
  ...
  if (within_range) { fscanf(fp, "%f", &local->values[i]); } // Copy data to
       local
  ...
  fclose(fp); }
```

Computation and queues. The kernels are `void` functions with at least one parameter, which is the label of the kernel. Inside the kernel, no MPI primitive should be used to perform message passing. Data received from another participant or data that need to be sent to another participant can be accessed using a receive queue and send queue. Consider the following kernel for the label LeftToRight in the stencil example:

Kernel: LeftToRight

```c
#pragma pabble kernel LeftToRight
void accumulate_LeftToRight(int id)
{ // Sender sends right col of submatrix and Recver receives left col.
  if (!pabble_recvq_isempty() && pabble_recvq_top_id() == id) {
    tmp[HALO_LEFT] = (double *)pabble_recvq_dequeue(); // Get received value.
  } else { tmp[HALO_RIGHT] = (double *)calloc(meta.buflen(id), sizeof(double));
    /* populate tmp[HALO_RIGHT] */
    pabble_sendq_enqueue(id, tmp[HALO_RIGHT]); // Put buffer to be sent
  }
}
```

Each kernel has access to a send and receive queue local to the whole process, which holds pointers to the buffer to be sent and the buffer containing the received messages, respectively. The queues are the only mechanism for kernels to interface the MPI backbone. The simplest kernel is one that forwards incoming messages from the receive queue directly to the send queue. In the above function, when the kernel function is called, it either consumes a message from the receive queue if it is not empty (i.e. after a receive), or produce a message for the send queue (i.e. before a send).

Kernels can have extra parameters. For example, in the init function above, filename is a parameter that is not specified by the protocol (i.e. Init ()). When such functions are called, all extra parameters are supplied by command-line arguments in the final generated MPI application.

In the next two sections we describe: (1) the compilation process to generate the MPI backbone and (2) the merging process in which we combine the MPI backbone and the kernels.

6.3. Compilation Step 1: Protocol to MPI backbone

This section describes the MPI backbone code generation from Pabble protocols. First the generated MPI backbone code of the running example is shown, then the translation rules from Pabble statements to MPI code are explained along with details of how to map Pabble participants into MPI processes.

Generated MPI Backbone

```
1  int main(int argc, char *argv[])
2  { MPI_Init(&argc, &argv);
3    MPI_Comm_rank(MPI_COMM_WORLD, &meta.pid);
4    MPI_Comm_size(MPI_COMM_WORLD, &meta.nprocs);
5  #pragma pabble type T
6    typedef void T; ⇒ typedef double T;
7    MPI_Datatype MPI_T; ⇒ MPI_Datatype MPI_T = MPI_DOUBLE;
8
9    T *bufLeftToRight_r, *bufLeftToRight_s;
10   /** Other buffer declarations **/
11   /** Definitions of cond0, cond1, ... **/
12 #pragma pabble predicate Steps
13   while (1) { ⇒ while(iter())
14     if (cond0) { /*if P[i:0..(N-1)][j:1..(N-1)]*/
15      bufLeftToRight_r = (T *)calloc(meta.buflen(LeftToRight), sizeof(T));
16      MPI_Irecv(bufLeftToRight_r, meta.buflen(LeftToRight), MPI_T, /*P[i][(j-1)]
            */...);
17      MPI_Wait(&req[0], &stat[0]);
18      pabble_recvq_enqueue(LeftToRight, bufLeftToRight_r);
19 #pragma pabble kernel LeftToRight ⇒ accumulate_LeftToRight(LeftToRight);
20    }
21     if (cond1) { /*if P[i:0..(N-1)][j:0..(N-2)]*/
22 #pragma pabble kernel LeftToRight  ⇒ accumulate_LeftToRight(LeftToRight);
23      bufLeftToRight = pabble_sendq_dequeue();
24      MPI_Isend(bufLeftToRight, meta.buflen(LeftToRight), MPI_T, /*P[i][(j+1)]*/
            ...);
25      MPI_Wait(&req[1], &stat[1]);
26      free(bufLeftToRight);
27     }
28     /** similarly for RightToLeft, UpToDown and DownToUp **/
29     MPI_Finalize();
30   }
31   return EXIT_SUCCESS; }
```

Listing 6.2: MPI backbone generated from the `Stencil` protocol.

6.3.1. MPI backbone generation from `Stencil` protocol

Based on the Pabble protocol (e.g. Listing 6.1), our code generation framework generates an *MPI backbone* code (e.g. Listing 6.2). First it automatically generates *endpoint protocols* from a global protocol as an intermediate step to make MPI code generation more straightforward.

An MPI backbone is a C99 program with boilerplate code for initialising and finalising the MPI environment of a typical MPI application (Line 2-4 and 29 respectively), and MPI primitive calls for message passing (e.g. `MPI_Isend` /`MPI_Irecv`). Therefore the MPI backbone realises the interaction between participants as specified in the Pabble protocol, without supporting any specific application functionality. The backbone has three kinds of `#pragma` annotations as placeholders for kernel functions, types and program logic.

The annotations are explained in Section 6.4. The boxed code in Listing 6.2 represents how the backbone are converted to code that calls the kernel functions in the MPI program.

On Lines 5 and 6, *generic type* `T` and `MPI_T` are defined datatypes for C and MPI respectively. `T` and `MPI_T` are refined later when an exact type (e.g. `int` or composite `struct` type) is known with the kernels.

Following the type declarations, are other variable declarations including the buffers (Line 9), and their allocation and deallocation are managed by the backbone. They are generated as guarded blocks of code, which come directly from the endpoint protocol. Line 14-20 shows a guarded receive that correspond to `if P[i:0..(N-1)][j:1..(N-1)] LeftToRight(T)from P[i][j-1]` in the protocol and Line 21-27 for `if P[i:0..(N-1)][j:0..(N-2)] LeftToRight (T)to P[i][j+1]`.

6.3.2. MPI backbone generation from Pabble

Below we explain how Pabble statements are translated into MPI blocks through Table 6.1–6.4.

(1) Interaction. An interaction statement in a Pabble protocol is projected in the endpoint protocol as two parts: receive and send.

The first line of the endpoint protocol shows a receive statement, written in Pabble as `if P[dstId] from P[srcId]`. The statement is translated to a block of MPI code in 3 parts. First, memory is dynamically allocated for the receive buffer (Line 2), the buffer is of `Type` and its size fetched from the function `meta.bufsize(Label)`. The function is defined in the kernels and returns the size of message for the given message label. Next, the program calls `MPI_Recv` to receive a message (Line 3) from participant `P[srcRole]` in Pabble. `role_P(srcIdx)` is a lookup macro from the generated backbone to return the process id of the sender. Finally, the received message, stored in the receive buffer `buf`, is enqueued into a global receive queue with `pabble_recvq_enqueue()` (Line 4), followed by the pragma indicating a kernel of label `Label` should be inserted. The block of receive code is guarded by an if-condition, which executes the above block of MPI code only if the current process id matches the receiver process id.

The next line in the endpoint protocol is a send statement, converse of

(1) Interaction

Global Protocol

```
Label(Type) from P[srcIdx] to P[dstIdx];
```

Projected Endpoint Protocol

```
if P[dstIdx] Label(Type) from P[srcIdx];
if P[srcIdx] Label(Type) to P[dstIdx];
```

Generated MPI Backbone

```
if (meta.pid == role_P(dstIdx)) {
  buf = (Type *)calloc(meta.bufsize(Label), sizeof(Type));
  MPI_Recv(buf, meta.bufsize(Label), MPI_Type, role_P(srcIdx), Label, ...);
  pabble_recvq_enqueue(Label, buf);
  #pragma pabble kernel Label
}
if (meta.pid == role_P(srcIdx)) {
  #pragma pabble kernel Label
  buf = pabble_recvq_dequeue();
  MPI_Send(buf, meta.bufsize(Label), MPI_Type, dstIdx, Label, ...); free(buf);
}
```

(2) Parallel interaction

Global Protocol

```
Label(Type) from P[i:1..N-1] to P[i+1];
```

Projected Endpoint Protocol

```
if P[i:2..N] Label(Type) from P[i-1];
if P[i:1..N-1] Label(Type) to P[i+1];
```

Generated MPI Backbone

```
if (role_P(2)<=meta.pid&&meta.pid<=role_P(N)) {
  buf = (Type *)calloc(meta.bufsize(Label), sizeof(Type));
  MPI_Recv(..., prevRank = meta.pid-1, Label, ...);
  pabble_recvq_enqueue(Label, buf);
#pragma pabble kernel Label
}

if (role_P(1)<=meta.pid&&meta.pid<=role_P(N-1)) {
#pragma pabble kernel Label
  buf = pabble_sendq_dequeue();
  MPI_Send(..., nextRank = meta.pid+1, Label, ...); free(buf);
}
```

(3) Internal interaction

Global/Endpoint Protocol

```
Internal() from __self to __self;
```

Generated MPI Backbone

```

#pragma pabble Internal
```

Table 6.1.: Pabble interaction statements and their corresponding code.

the receive statement, written as `if P[srcIdx] Label(Type)to P[dstIdx]`. The MPI code begins with the pragma annotation, then dequeuing the global send queue with `pabble_sendq_dequeue()` and sends the dequeued buffer with `MPI_Send`. After this, the send buffer, which is no longer needed, is deallocated. The block of send code is similarly guarded by an if-condition to ensure it is only executed by the sender. By allocating memory before receive and deallocating memory after send, the backbone manages memory for the user systematically.

(2) Parallel interaction. A Pabble parallel interaction statement is written as `Label(Type)from P[i:1..N-1] to P[i+1]`, meaning all processes with indices from `1` to `N-1` send a message to its next neighbour. `P[1]` initiates sending to `P[2]`, and `P[2]` receives from `P[1]` then sends a message to `P[3]`, and so on. As shown in the endpoint protocol which encapsulates the behaviour of all `P[1..N]` processes, the statement is realised in the endpoint as conditional receive followed by a conditional send, similar to ordinary interaction. The difference is the use of a range of process ids in the condition, and *relative* indices in the sender/receiver indices. The generated MPI code makes use of expression with `meta.pid` (current process id) to calculate the relative index.

(3) Internal interaction. When role with name `__self` is used in a protocol, it means that both the sending and receiving endpoints are internal to the processes, and there is no interaction with external processes. This statement applies to all processes, and is not to be confused with self-messaging, e.g. `Label() from P[1] to P[1]`, which would lead to deadlock. The statement does not use any MPI primitives. The purpose of using this special role is to create optional insertion point for the MPI backbone, which may be used for optional kernels such as initialisation or finalisation, hence it generates a pragma in the MPI backbone.

(4) Iteration and (5) For-loop. `rec` and `foreach` are iteration statements. Specifically `rec` is recursion, where the iteration conditions are not specified explicitly in the protocol, and translates to `while`-loops. The loop condition is the same in all processes, otherwise be known as *collective loops*. The loop generated by `rec` has a `#pragma pabble predicate` annotation, so that the loop condition can be later replaced by a kernel (see Section 6.4).

The `foreach` construct, on the other hand, specifies a counting loop, iterating over the integer values in the range specified in the protocol from

(4) Iteration

Global/Endpoint Protocol

```
rec LoopName { ... continue LoopName; }
```

Generated MPI Backbone

```

#pragma pabble predicate LoopName
while (1) {
  ... }
```

(5) For-loop

Global/Endpoint Protocol

```
foreach (i:0..N-1) { ... }
```

Generated MPI Backbone

```

for (int i=0; i<=N-1; i++) {
  ...
}
```

Table 6.2.: Pabble control-flow statements and their corresponding code.

the lower bound (e.g. `0`) to the upper bound value (e.g. `N-1`). This construct can be naturally translated into a C `for`-loop.

(6) Choice

Global Protocol

```
choice at P[master] {
  Branch0(Type) from P[master]
                to P[worker];
  ...
} or { ... }
```

Projected Endpoint Protocol

```
choice at P[master] {
  if P[worker] Branch0(Type) from P[master];
  if P[master] Branch0(Type) to P[worker];
  ...
} or { ... }
```

Generated MPI Backbone

```
if (rank==role_P(master)) { // Choice sender
#pragma pabble predicate Branch0
  if (1) {
    // Block of send.
    MPI_Send(..., MPI_Type, role_P(worker), Branch0, ...);
  } else
#pragma pabble predicate Branch1
  if (1) { ... }
} else { // Choice receiver
  MPI_Probe(role_P(master), MPI_ANY_TAG, comm, &status); switch (status.MPI_TAG) {
    case Branch0:
    // Ordinary block of recv.
    if (rank==role_P(worker)) {
      MPI_Recv(..., MPI_Type, role_P(master), Branch0, ...);
      pabble_recvq_enqueue(Branch0, buf); }
      ... break;
#pragma pabble Branch1
    case Branch1: ...
  }
}
```

Table 6.3.: Pabble choice and their corresponding code.

(6) Choice. Conditional branching in Pabble is performed by label branching and selection. We use the example given in Table 6.3 to explain. The deciding process, e.g. `P[master]`, makes a choice and executes the statements in the selected branch. Each branch starts by sending a unique label, e.g. `Branch0`, to the decision receiver, e.g. `P[worker]`. Hence for a well-formed Pabble protocol, the first line of each branch is from the deciding process to the same process but using a different label.

Note that the decision is only known between the two processes in the first statement, and other processes should be explicitly notified or use broadcast to propagate the decision. The MPI backbone is generated with a different structure as the endpoint protocol. First, the MPI backbone contains an outer if-then-else, splitting the deciding process (Line 1–9) and the decision receiver (Line 9–20). In the deciding process, a block of if-then-else-if code is generated to perform a send with different label (called MPI tag), e.g. Line 5. This statement is generated with all the queue and memory management code as described above for ordinary interaction statements. Each of the if-condition is annotated with `#pragma pabble predicate BranchLabel`, so that the conditions can be replaced by predicate kernels (see Section 6.4). For the decision receiver, `MPI_Probe` is used to peek the received label, then the `switch` statement is used to perform the correct receive (for different branches).

(7) Scatter, (8) Gather and (9) All-to-all. Collective operations are written in Pabble as multicast or multi-receive message interactions. While it is possible to convert these interactions into multiple blocks of MPI code following the rules in Table 6.2 (e.g. loop through receivers for scatter), we take advantage of the efficient and expressive collective primitives in MPI. Table 6.4 shows the conversion of Pabble statements into MPI collective operations. We describe only the most generic collective operations, i.e. `MPI_Scatter`, `MPI_Gather` and `MPI_Alltoall`.

Translating collective operations from Pabble to MPI considers both global Pabble protocol statements and endpoint protocol. If a statement involves the `__All` role as sender, receiver or both, it is a collective operation. Table 6.4 shows that translated blocks of MPI code do not use `if`-statements to distinguish between sending and receiving processes. This is because collective primitives in MPI are executed by *both* the senders and the receivers,

(7) Scatter

Global Protocol

```
Label(Type) from P[rootRole] to __All;
```

Generated MPI Backbone

```
rbuf = (Type *)calloc(
                  meta.buflen(Label), sizeof(Type));
#pragma pabble kernel Label
sbuf = pabble_sendq_dequeue();
MPI_Scatter(sbuf, meta.buflen(Label), MPI_Type,
           rbuf, meta.buflen(Label), MPI_Type, role_P(rootRole), ...);
pabble_recvq_enqueue(Label, rbuf);
#pragma pabble kernel Label
free(sbuf);
```

(8) Gather

Global Protocol

```
Label(Type) from __All to P[rootRole];
```

Generated MPI Backbone

```
rbuf = (Type *)calloc(
                  meta.buflen(Label)*meta.nprocs, sizeof(Type));
#pragma pabble kernel Label
sbuf = pabble_sendq_dequeue();
MPI_Gather(sbuf, meta.buflen(Label), MPI_Type,
          rbuf, meta.buflen(Label), MPI_Type, role_P(rootRole), ...);
pabble_recvq_enqueue(Label, rbuf);
#pragma pabble kernel Label
free(sbuf);
```

(9) All-to-All

Global Protocol

```
Label(Type) from __All to __All;
```

Generated MPI Backbone

```
rbuf = (Type *)calloc(
                  meta.buflen(Label)*meta.nprocs, sizeof(Type));
#pragma pabble kernel Label
sbuf = pabble_sendq_dequeue();
MPI_Alltoall(sbuf, meta.buflen(Label), MPI_Type,
            rbuf, meta.buflen(Label), MPI_Type, ...);
pabble_recvq_enqueue(Label, rbuf);
#pragma pabble kernel Label
free(sbuf);
```

Table 6.4.: Pabble collective operations and their corresponding code.

and the runtime decides whether it is a sender or a receiver by inspecting the `rootRole` parameter (which is a process rank) in the `MPI_Scatter` or `MPI_Gather` call. Otherwise the conversion is similar to their point-to-point counterparts in Table 6.1.

Process scaling. In addition to the translation of Pabble statements into MPI code, we also define the process mapping between a Pabble protocol and a Pabble-generated MPI program. Typical usage of MPI programs can be parameterised on the number of spawned processes at runtime via program arguments. Hence, given a Pabble protocol with *scalable* roles, we describe the rules below to map (parameterised) roles into MPI processes.

A Pabble protocol for MPI code generation can contain any number of constant values (e.g. `const M = 10`), which are converted in the backbone as C constants (e.g. `#define M 10`), but it can use at most one *scalable constant* [NY14b], and will scale with the total number of spawned processes. A scalable constant is defined as:

```
const N = 1..max;
```

The constant can then be used for defining parameterised roles, and used in indices of parameterised message interaction statements. For example, to declare an $N \times N$ role `P`, we write in the protocol:

```
global protocol P (role P[1..N][1..N])
```

which results in a total of N^2 participants in the protocol, but N is not known until execution time. MPI backbone code generated based on this Pabble protocol uses `N` throughout. Since the only parameter in a scalable MPI program is its `size` (i.e. number of spawned processes), the following code is generated in the backbone to calculate, from `size`, the value of C local variable `N`:

```
MPI_Comm_size(MPI_COMM_WORLD, &meta.nprocs); // # of processes
int N = (int)pow(meta.nprocs, 1/2); // N = sqrt(meta.nprocs)
```

6.4. Compilation Step 2: Aspect-Oriented Design-Flow

This section focuses on the final stage of our code generation framework, which merges two input components to derive the complete MPI program: (1) the communication safe MPI backbone derived automatically from a Pabble protocol (Section 6.3.1), and (2) the user supplied kernels capturing application functionality.

The MPI backbone is automatically annotated with pragma statements referencing all the labels defined in the protocol; the programmer, on the other hand, must manually annotate each kernel with the corresponding label. This way, our code generation framework can automatically merge both components.

Our approach takes a similar path as OpenMP [DM98] and OpenACC [WSTaM12], which parallelise sequential programs using non-invasive `#pragma` annotations. The difference is that while OpenMP operates on a shared memory architecture model and OpenACC operates via a host-directed execution (co-processor) model, our approach allows applications to target customised platform topologies defined by Pabble, since MPI works on both shared and distributed memory platforms.

LARA language. To support an automated merging process, our programming framework uses an aspect-oriented programming (AOP) language called LARA [CCC+12]. As far as we know, LARA is the only aspect-oriented approach that targets all stages of a development process allowing static code analysis and manipulation (e.g. source-level translation and code optimisation), toolchain execution (e.g. for design-space exploration) and application deployment (e.g. to extract dynamic behaviour). These various tasks, which are often performed manually and independently, can be described in a unified way as LARA aspects. These aspects can then drive LARA *weavers* to apply a particular strategy in a systematic and automated way. In our code generation framework, we use LARA's ability to analyse and manipulate C code to automate the merging process between the MPI backbone and the kernels sources (Section 6.4.1), and also to further optimise the MPI code by overlapping communication and computation (Section 6.4.2).

6.4.1. Merging process

To combine the MPI backbone with the kernels, our aspect-oriented designflow inserts kernel function calls into the MPI backbone code. The insertion points are realised as `#pragma`s in the MPI backbone code, generated from the input protocol as placeholders where functional code is inserted. There are multiple types of annotations whose syntax is given as:

```
#pragma pabble [<entry point type>] <entry point id> [(param0, ...)]
```

where *entry point type* is one of `kernel`, `type` or `predicate`, and *entry point id* is an alphanumeric identifier.

	Generated MPI backbone	User supplied kernel	Merged code

Kernel Function	`#pragma pabble kernel Label`	`#pragma pabble kernel Label`
		`void kernel_func(int label)`
		`{ ... }`
			`kernel_func(Label);`

Datatypes	`#pragma pabble type T`
	`typedef void T;`
	`MPI_Datatype MPI_T;`
		`#pragma pabble type T`
		`typedef double T;`
			`typedef double T;`
			`MPI_Datatype MPI_T`
			`= MPI_DOUBLE;`

Conditionals	`#pragma pabble predicate Cond`
	`while (1)`
	`{ ... }`
		`#pragma pabble predicate Cond`
		`int condition()`
		`{ ... return bool; }`
			`while (condition())`
			`{ ... }`

Table 6.5.: Annotations in backbone and kernel.

Kernel function. `#pragma pabble kernel Label` defines the insertion point of kernel functions in the MPI backbone code. `Label` is the label of the interaction statement, e.g. `Label(T)from Sender to Receiver`, and the annotation is replaced by the kernel function associated to the label `Label`. Programmers must use the same pragma to manually annotate the implementation of the kernel function. The first row in Table 6.5 shows an example.

Datatypes. `#pragma pabble type TypeName` annotates a generic type name in the backbone, and also annotates the concrete definition of the datatype in the kernels. In the second row of Table 6.5, the C datatype `T` is defined to be void since the protocol does not have any information to realise the type. The kernel defines `T` to be a concrete type of `double`, and hence our tool transforms the `typedef` in the backbone into `double` and infers the corresponding `MPI_Datatype` (MPI derived datatypes) to the built-in MPI integer primitive type, i.e. `MPI_Datatype MPI_T = MPI_DOUBLE`. Our tool also supports generating MPI datatypes for structures of primitive types, e.g. `struct { int x, int y, double m }` is transformed to its MPI-equivalent datatype.

Conditionals. `#pragma pabble predicate Label` is a pragma for annotating predicates, e.g. loop conditions or if-conditions, in the backbone. Since a Pabble communication protocol (and transitively, the MPI backbone) does not specify a loop condition, the default loop condition is `1`, i.e. always true. This annotation introduces a way to insert a conditional expression defined as a kernel function. It precedes the `while`-loop, as shown in the third row of Table 6.5, to label the loop with the name `Label`. The kernel function that defines expressions must use the same annotation as the backbone, e.g. `#pragma pabble predicate Label`. After the merge, this kernel function is called when the loop condition is evaluated.

6.4.2. Performance optimisation for overlapping communication and computation by MPI immediate operators

When designing a protocol with a session-based approach such as Pabble protocol, the resulting MPI backbone guarantees communication safety, i.e. the structures of interactions between the processes are compatible. However, that does not necessarily guarantee the most efficient communication pattern. For example the pipeline Pabble statement `T() from P[i:0..N-1] to P[i+1]` results in a communication safe pattern of Receive-Send for `P[1]` to `P[N]`. The protocol implies there is a dependency between the received message and the send message, hence each process in the pipeline must wait for the messages sent by processes up the pipeline, before they can start sending a message to processes down the pipeline. This is not optimal because the stall time between the beginning of the pipeline and when the first message is received is a waste of CPU resources. Often parallel applications can be modified such that the dependencies within the same iteration are removed, so the message passing can start sending straight away and overlap with receive using *asynchronous messaging mode.*

The use of asynchronous communication is dependent on the kernel functionality and how message dependencies must be handled. For this reason, programmers can use the `async` directive when annotating their kernels, e.g. `#pragma pabble async kernel LABEL`, in order to trigger this optimisation.

The LARA aspect-oriented weaver transforms the generated code without changing the ordering of the MPI message passing primitives, and hence preserves the communication safety guarantees of the MPI backbone.

This optimisation relies on the placement of MPI's immediate communication primitives, which is made up of two parts: (1) a primitive call (`MPI_Isend` or `MPI_Irecv`) to initiate the message transfer which returns immediately and after which the buffer should not be accessed, and a (2) second primitive call (`MPI_Wait`) to block and wait for the transfer to complete. Between the initial call and the wait, the application can perform computation in parallel with the message transfer to realise the communication-computation overlap.

The optimisation overlaps the computation which generates results to be sent in the following iteration and the communication of sending and receiving results of previous iteration to and from a neighbouring process. Since all computations are executed in parallel, and the communication overlaps with the computation, we achieve a speed-up for the parallel application over the sequential version of the same application.

Below we show an example before the optimisation (left) and after the optimisation (right) where the `MPI_Wait` is issued as late as possible:

Original

```
if (cond) {
#pragma pabble Label
 buffer = pabble_sendq_dequeue();

 MPI_Send(buffer, ...);
 free(buffer);
}
```

Optimised

```
if (cond) {
 buffer = pabble_sendq_dequeue();
 MPI_Isend(buffer, ..., request); }
...
if (cond) {
#pragma pabble Label
 MPI_Wait(request); free(buffer); }
```

Note that our transformation preserves the ordering of communication defined in the unoptimised backbone. The following presents an example that splits an ordinary MPI receive/send as in the `Stencil` example into a set of statements that interleave asynchronous receive/send.

Original

```
MPI_Recv(...);
MPI_Send(...);
```

Optimised

```
MPI_Irecv(..., request1);
MPI_Isend(..., request2);
/* Interleave with computation */
MPI_Wait(request1, ...);
MPI_Wait(request2, ...);
```

Since `MPI_Wait` is an operation that blocks until the send and receive buffers can be accessed, we can ensure that `MPI_Isend(..., request1)` is completed before `MPI_Irecv(..., request2)` even if the transmission of data for the latter primitive is finished before the former.

6.5. Evaluation

We evaluate our approach on two areas: *productivity* and *performance*.

Productivity. Our main aim of the code generation workflow is to simplify parallel programming by offering a programming methodology, with which the programmer focuses on the computation code and write high-level communication topologies in Pabble. We believe that using Pabble to develop an MPI parallel application, which generates an MPI backbone automatically, increases productivity of the programmer as the MPI backbone code contains mostly tedious and mechanical details that do not offer any extra advantage over manually written code by a pragmatic programmer. We therefore compare the relative effort of writing the same parallel application manually and by code generation by measuring their lines of code to quantify productivity. It is not an absolute measure but can be used as a general guide to compare the relative effort needed to write the parallel application.

Performance. The performance evaluations are conducted to show that our framework is sufficiently general to generate the different classes of algorithms in our chosen set of benchmarks (from different Dwarf [AWW$^+$09] categories), and that the generated code is ordinary MPI applications that has reasonable performance. Finally, we show the impact and limitations of the asynchronous optimisation in N-body simulation and Linear Equation Solver.

6.5.1. Productivity and Reusability

The table below presents a comparison of different parallel algorithms developed using our approach. The second and third columns show the input Pabble protocol and whether it is available in our protocol repository. The Dwarf column denotes the categorisations of parallel computational and structural patterns defined in [AWW$^+$09]; SG stands for ‘Structured Grid’, PM is ‘Particle Methods’; DM is ‘Dense Matrix’; and S is ‘Spectral (FFT)’. The next three columns show lines of code in the input Pabble protocol, the generated backbone, and the input user kernel file. The final column shows the effort ratio of user written code against the total ($\frac{Kernels}{Backbone+Kernels}$ for protocols in repository or $\frac{Kernels+Pabble}{Backbone+Kernels}$). The higher the ratio, relatively more effort is needed to write an equivalent program from scratch.

	Protocol	Repo.	Dwarf
heateq [BDHT]	stencil	✓	SG
nbody	ring	✓	PM
wordcount	scatter-gather	✓	
adpredictor [GCBH10]	scatter-gather	✓	
montecarlo	scatter-gather	✓	
montecarlo-mw	master-worker	✓	
LEsovler [NY14b]	wrapround mesh		SG
matvec	custom [NYL13]		DM
fft64	6-step butterfly		S

heateq is an implementation of the heat equation based on [BDHT], and uses the stencil protocol in our running example. nbody is a 2D N-body simulation implemented with a ring topology; it is optimised with the asynchronous messaging mode described in Section 6.4.2. wordcount is a simple application that counts the number of occurrences of each word in a given text, implemented using the scatter-gather pattern. adpredictor is an implementation of Microsoft's AdPredictor [GCBH10] algorithm for calculated click-through rate, also implemented in the same scatter-gather pattern, but with a different set of kernel functions. LEsolver is a linear equation solver parallelised with a custom wraparound mesh topology outlined in [NY14b]. montecarlo is Monte-Carlo π simulation, implemented with two different patterns, scatter-gather and master-worker. A remarkable difference between the two patterns is that the former uses collective operations and all processes are involved in the main calculation, whereas with the master-worker pattern workers are coordinated by a central master process by P2P communication that does not perform the main calculation. Note that the kernels used for both implementations are the same (except with different kernel labels). matvec is matrix-vector multiplication parallelised using the `MatVec` protocol outlined in [NYL13]. fft64 is an implementation of the Cooley-Tukey FFT between 64 processes using 6 steps of butterfly exchange between pairs of processes.

	Pabble	Backbone	Kernels	Effort
heateq [BDHT]	15	154	335	0.69
nbody	15	93	228	0.71
wordcount	8	76	176	0.70
adpredictor [GCBH10]	8	76	182	0.71
montecarlo	8	76	70	0.48
montecarlo-mw	10	82	70	0.46
LEsovler [NY14b]	15	132	208	0.66
matvec	15	130	117	0.41
fft64	11	64	134	0.68

Reusability Both our implementations of wordcount and adpredictor use the scatter-gather pattern. They exemplify the advantages of pattern programming – common parallel patterns are collected and stored in our protocol repository, and they are maintained separately from the user kernels so new parallel applications can be constructed by writing new kernels only. In addition to reusable protocols, some kernels can also be reused with different protocols. The scenarios for kernels to be reused are less common since partitioning of input data are usually dependent on the protocol, and the kernels are designed to be parallelised with a single protocol. For example, we show two montecarlo implementations, one with scatter-gather and another with master-worker pattern. Since the algorithm is embarrassingly parallel and does not depend on input data, both implementations can share the same kernel.

With our results we argue that our workflow saves development and debugging efforts for MPI parallel applications, especially for novice parallel programmers. The user can focus on developing and maintaining the functional behaviour of their application, knowing that the merging of updated kernels and the respective MPI backbones are correct. While our metric is not an absolute measurement of the difficulty of developing an MPI application, and that we ignore that a more complex program may have less lines of code, we believe the presence of a high-level protocol helps, rather than hinders, the understanding of the application. As a session-based framework, we can ensure that the communication aspects of the generated application are correct – which narrows down deadlocks to computation, hence reducing debugging efforts.

6.5.2. Performance

We evaluate our approach on cx1, a general purpose multi-core cluster [Imp]. All implementations are compiled with `icc` 13.0.0 with `-O3` option, and tested using Intel MPI library version 31.0.38.

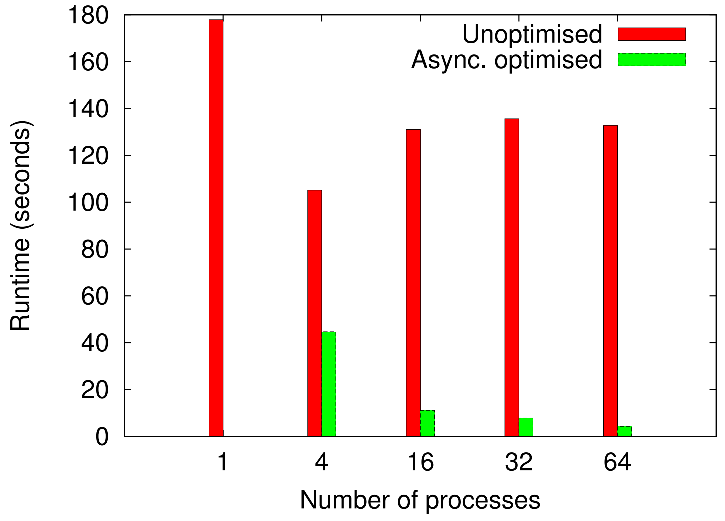


Figure 6.4.: N-body simulation (nbody).

In Figure 6.4 we compare the performance of nbody with and without asynchronous optimisation described in Section 6.4.2. The optimisation overlaps the main calculation with the communication, and the results show significant improvements over the unoptimised version.

Figure 6.5 presents the runtime performance of LEsolver which uses a custom wraparound mesh protocol with asynchronous optimisation. In comparison with nbody, the optimisation effect on LEsolver has less impact. This is partly because the asynchronous kernel implemented by nbody is more complex than the kernel implemented by LEsolver, so the time spent on communication is dominant. The asynchronous kernel in LEsolver also represents a smaller proportion of the total computations, hence it has the less effect on the overall runtime.

Figure 6.6 shows that the two implementations – wordcount and adpredictor – both of which use the scatter-gather pattern and a different set of kernels follow a similar trend in scalability, which is dependent on the size of the input data.

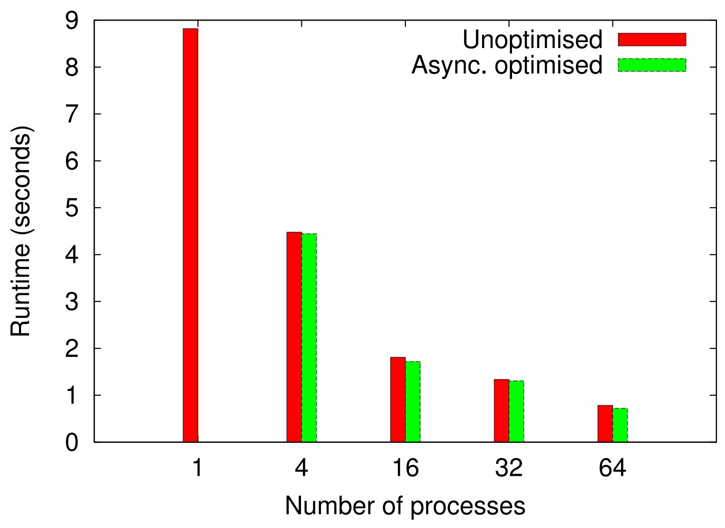


Figure 6.5.: Linear Equation Solver (LEsolver).

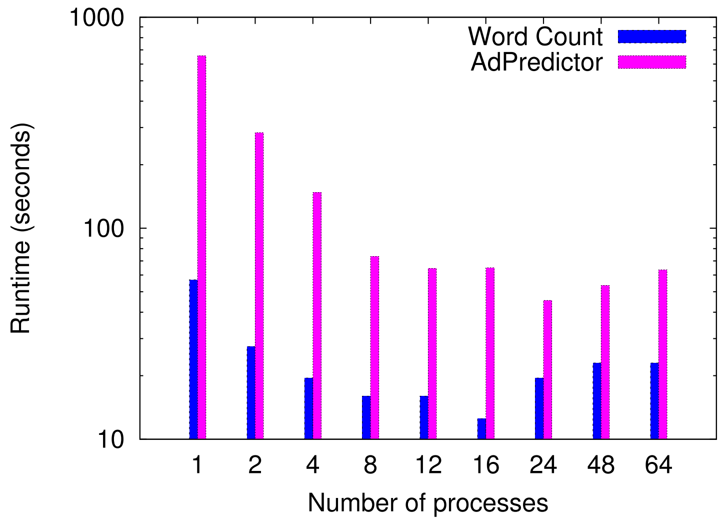


Figure 6.6.: Word Count (wordcount) and AdPredictor (adpredictor)

Figure 6.7 compares implementations in our framework running in 64 processes against sequential C versions. Results show speedup for all algorithms except fft64 due to communication overhead of the more complex butterfly topology.

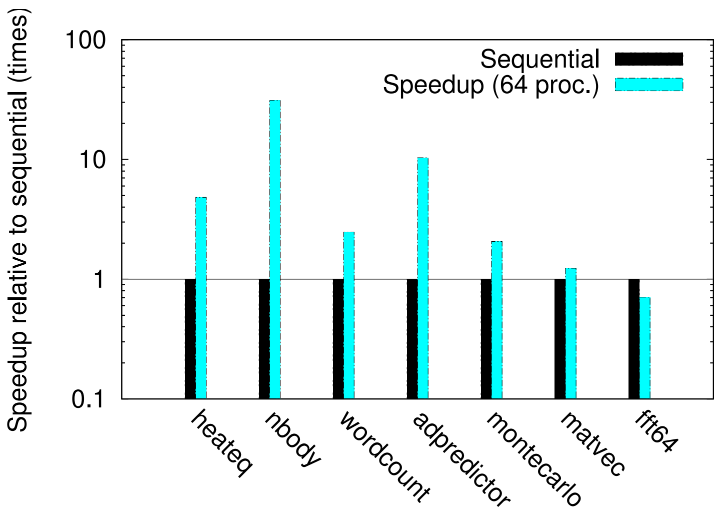


Figure 6.7.: Parallelisation speedup.

6.5.3. Limitations

Despite the flexibility we show in the overlapping of kernels, there are limitations to this optimisation approach. Take the stencil protocol for example, we are able to overlap the computation kernels associated to the communication, i.e. `LeftToRight`, `RightToLeft`, `UpToDown` and `DownToUp`, but the main calculation cannot be started after all the data have been received due to the data dependencies between the data. Since the computation associated to the communication is relative simple, only saving the received data to local and copying the data to send from local to send buffer, the performance improvement by the optimisation is not expected to be huge. This is a consequence of the parallel algorithm itself, where better performance will require unrolling of the loop three or more times to allow overlapping of the main calculation with communication.

In contrast, algorithms such as N-body simulation which can be implemented with little or no dependencies between each iteration, can take full advantage of the overlapping by asynchronous message-passing. Hence, a relatively larger impact is possible with the optimisations outlined above. The evaluation results shown above confirms our observation.

6.6. Summary and Discussion

In this chapter we presented a session-based workflow for constructing safe and efficient parallel applications. The framework consists of two parts, a safe-by-construction parallel interaction backbone, generated from the Pabble protocol description language, and an aspect-oriented compilation framework to mechanically insert computation code into the backbone and asynchronous optimisation. We argue that our approach simplifies parallel programming by making use of parallel communication patterns, described with our Pabble protocol description language, and building independent kernel code around the patterns as sequential C code. We also show the flexibility of the framework where multiple sets of kernels can share a common parallel communication pattern, where the kernels are maintained separately.

This code generation framework is an approach for parallel programming based on the theoretical framework of MPST through Pabble. Even with parameterised MPST, it is required that the (parameterised) roles involved in a communication is known at design time. In the next chapter we present an alternative to MPST for designing parallel programs by *composition* of binary sessions. The approach, called multi-channel session types, makes use of a pair of global session synchronisation primitives `inwhile` and `outwhile`, to allow inter-session synchronisations. Communication safety and deadlock freedom are guaranteed by analysing the topologies of the synchronisation, and provide the same guarantee as MPST with additional flexibility.

7 Multi-channel Session Types

Overview This chapter investigates the use of Session Java (SJ) for session-typed parallel programming, and introduces new language primitives for *chained iteration* and *multi-channel communication*. These primitives allow the efficient coordination of parallel computation across multiple processes, thus enabling SJ to express the complex communication topologies often used by parallel algorithms. We demonstrate that the new primitives yield clearer and safer code for pipeline, ring and mesh topologies through implementations of representative parallel algorithms. We then present a semantics and session typing system including the new primitives, and prove type soundness and deadlock-freedom for our implementations.

7.1. Introduction

In the previous chapters we have explored using the general framework of Multiparty Session Types (MPST) to guarantee type safety, communication correctness and deadlock freedom for interactions. However, using MPST entails the presence of a *global type* or protocol to express the global view of interactions, which requires all the participants of an interaction to be known at design time. In parallel applications that scale at runtime, the number of participants is not known at design time, and hence cannot be expressed directly in global types. Approaches such as Parameterised MPST [DYBH12a] or the Pabble protocol language introduced in Chapter 4 overcomes this requirement by embedding the information about the number of participants into the type through role indices.

The implementation of the inwhile and outwhile primitives are contributed by SJ authors Raymond Hu and Olivier Pernet, co-authors of [NYP+11]

This chapter presents an alternative approach to express the global view of interactions by composition of *binary* Session Types. We extend binary Session Types with *multi-channel* primitives, which allow us to represent complex, high-level communication patterns as globally synchronised chains of iterations. We implement the new multi-channel primitives in the Session Java (SJ) programming language [HYH08], and investigate the use the primitives in SJ.

The SJ compiler offers two strong static guarantees for session execution: (1) *communication safety*, meaning a session-typed process can never cause or encounter a communication error by sending or receiving unexpected messages; and (2) *deadlock-freedom* — a session-typed process will never block indefinitely on a message receive. However, SJ as presented in [HYH08] only guarantees progress for each session in isolation: deadlocks can still arise from the interleaving of multiple sessions in a process.

The combination of new primitives and a well-formed topology check extension to SJ compilation [HYH08] bring the benefits of type-safe, structured communications programming to HPC. The primitives can be chained, yielding a simple mechanism for structuring global control flow. We formalise these primitives as novel extensions of the *session calculus*, and the correctness condition on the shape of programs enforced by a simple extension of SJ compilation. This allows us to prove *communication safety* and *deadlock-freedom*, and offers a new, lightweight alternative to multiparty session types for global type-safety.

The following are the technical contributions of this chapter:

- We introduce SJ as a programming language for type-safe, efficient parallel programming, including our implementation of *multi-channel* session primitives, and the extended SJ tool chain for parallel programming in Section 7.2. We show that the new primitives enable clearer, more readable code.

- We discuss SJ implementations of parallel algorithms in Section 7.3 using n-body simulation (Section 7.3.1) and an Jacobi solution to the discrete Poisson equation (Section 7.3.2) as examples. Both examples use communication topology representative of a large class of parallel algorithms, and demonstrates the practical use of our multi-channel primitives.

- We define the *multi-channel session calculus*, its operational semantics, and typing system in Section 7.4. We prove that processes conforming to a *well-formed communication topology* (Definition 7.1) satisfy the subject reduction theorem (Theorem 7.1), which implies *type and communication-safety* (Theorem 7.2) and *deadlock-freedom* across multiple, interleaved sessions (Theorem 7.3).
- Finally, we present performance evaluation of the n-body simulation and the Jacobi solution we implemented in Section 7.5, , demonstrating the benefits of the new primitives. The SJ implementations using the new primitives show competitive performance against MPJ Express [BC00].

Detailed definitions, proofs, benchmark results and source code can be found in the Appendix (Section A.1).

7.2. Session-Typed Programming in SJ

This section firstly reviews the key concepts of *binary* session-typed programming using Session Java (SJ) [HYH08, Hu10]. In (1), we outline the basic methodology; in (2), the protocol structures supported by SJ. We then introduce the new session programming features developed in this chapter to provide greater expressiveness and performance gains for *session-typed parallel programming.* In (3), we explain session *iteration chaining*; and in (4), the generalisation of this concept to the *multi-channel* primitives. Finally, (5) describes the *topology verification* for parallel programs.

(1) Basic SJ programming SJ is an extension of Java for type-safe concurrent and distributed session programming. Session programming in SJ, as detailed in [HYH08], starts with the declaration of the intended communication protocols as session types; we shall often use the terms *session type* and *protocol* interchangeably. A session is the interaction between two communicating parties, and its session type is written from the viewpoint of one side of the session. The following declares a protocol named `P`:

```
protocol P !<int>.?(Data)
```

Protocol `P` specifies that, at this side of the session, we first send (`!`) a message of Java type `int`, then receive (`?`) another message, an instance of the Java class `Data`, which finishes the session. After defining the protocol, the programmer implements the processes that will perform the specified communication actions using the SJ *session primitives.* The first line in the following code implements an Alice process conforming to the `P` protocol:

```
Alice
// !<int>.?(Data)
alice.send(42);
Data d = (Data)alice.receive();
```

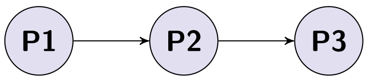
```
Bob
// ?(int).!<Data>
int i = bob.receiveInt();
bob.send(new Data());
```

The `alice` variable refers to an object of class `SJSocket`, called a *session socket*, which represents one endpoint of an active session. The session-typed primitives for session-typed communication behaviour, such as `send` and `receive`, are performed on the session socket like method invocations. `SJSocket` declarations associate a protocol to the socket variable, and the SJ compiler statically checks that the socket is indeed used according to the protocol, ensuring the *correct communication behaviour* of the process.

This simple session application also requires a counterpart Bob process to interact with Alice. For safe session execution, the Alice and Bob processes need to perform matching communication operations: when Alice sends an `int`, Bob receives an `int`, and so on. Two processes performing matching operations have session types that are *dual* to each other. The dual protocol to `P` is `protocol PDual ?(int).!<Data>`, and a dual Bob process can be implemented as in the second line of the above listing.

(2) More complex protocol structures Session types are not limited to sequences of basic message passing. Programmers can specify more complex protocols featuring *branching*, *iteration* and *recursion*.

The protocols and processes in Figure 7.1 demonstrate session iteration and branching. Process `P1` communicates with `P2` according to protocol `IntAndBoolStream`; `P2` and `P3` communicate following protocol `IntStream`. Like basic message passing, iteration and branching are coordinated by *active* and *passive* actions at each side of the session. Process `P1` actively decides whether to continue the session iteration using `outwhile(`*condition*`)`, and if so, selects a branch using `outbranch(`*label*`)`. The former action implements the `![`τ`]*` type given by `IntAndBoolStream`, where τ is the `!{ Label1:` τ_1`, Label2:` τ_2`, ... }` type implemented by the latter. Processes `P2` and `P3` passively follow the selected branch and the iteration decisions (received as internal control messages) using `inbranch` and `inwhile`, and



```
protocol IntAndBoolStream ![!{Label1: !<int>, Label2: !<boolean>}]*
protocol IntAndBoolDual   ?[?{Label1: ?<int>, Label2: ?(boolean)}]*
protocol IntStream        ![!<int>]*
protocol IntStreamDual    ?[?(int)]*
```

P1
```
// s: IntAndBoolStream
s.outwhile(x < 10) {
  s.outbranch(Label1) {
    s.send(42);
}}
```

P3
```
// s: IntStreamDual
s.inwhile {
  int i = s.receiveInt();
}
```

P3
```
// s1: IntAndBoolDual
// s2: IntStream
s2.outwhile(s1.inwhile()) {
  s1.inbranch() {
    case Label1: int i = s1.receiveInt();  s2.send(i);
    case Label2: boolean b = s1.receiveBool(); s2.send(42);
}}
```

Session socket `s` in `P1` follows `IntAndBoolStream`; `s1` and `s2` in `P2` follows `IntAndBoolDual` and `IntStream`; `s` in `P3` follows `IntStreamDual`.

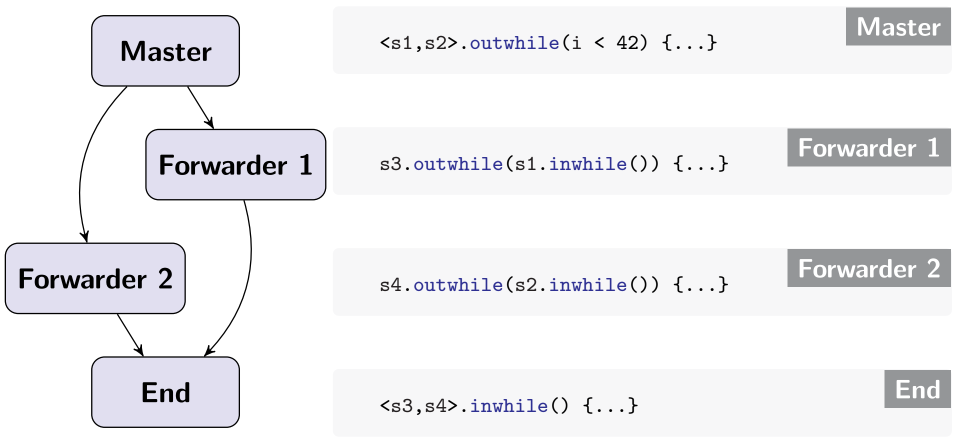
Figure 7.1.: Simple chaining of session iterations across multiple pipeline process.

proceed accordingly; the two dual protocols show the passive versions of the above iteration and branching types, denoted by `?` in place of `!`. So far, we have reviewed basic SJ programming features [HYH08] derived from standard session type theory [HVK98, YV07]; the following paragraphs discuss new features motivated by the application of session types to parallel programming in practice.

(3) Expressiveness gains from iteration chaining The three processes in Figure 7.1 additionally illustrate session *iteration chaining*, forming a linear pipeline as depicted at the top of Figure 7.1. The net effect is that `P1` controls the iteration of both its session with `P2` and transitively the session between `P2` and `P3`. This is achieved through the chaining construct `s2.outwhile(s1.inwhile())` at `P2`, which receives the iteration decision from `P1` and forwards it to `P3`. The flow of both sessions is thus controlled by the same master decision from `P1`.

Iteration chaining offers greater expressiveness than the individual iteration primitives supported in standard session types. Normally, session typing for ordinary `inwhile` or `outwhile` loops must forbid operations on any session other than the session channel that of loop, to preserve linear usage of session channels. This means that e.g. `s1.inwhile(){ s1.send(v); }` is allowed, whereas `s1.inwhile(){ s2.send(v); }` is not. With the iteration chaining construct, we can now construct a process containing two interleaved `inwhile` or `outwhile` loops on separate sessions. In fact, session iteration chaining can be further generalised as we explain below.

(4) Multi-channel iteration primitives Simple iteration chaining allows SJ programmers to combine multiple sessions into linear pipeline structures, a common pattern in parallel processing. In particular, type-safe session iteration (and branching) along a pipeline is a powerful benefit over traditional stream-based data flow [SPGV07]. More complex topologies, however, such as rings and meshes, require iteration signals to be directly forwarded from a given process to more than one other, and for multiple signals to be directed into a common sink; in SJ, this means we require the ability to send and receive multiple iteration signals over a set of session sockets. For this purpose, SJ introduces the generalised *multi-channel* primitives; the following focuses on multi-channel iteration, which extends the chaining constructs from above.



```
<s1,s2>.outwhile(i < 42) {...}
```

```
s3.outwhile(s1.inwhile()) {...}
```

```
s4.outwhile(s2.inwhile()) {...}
```

```
<s3,s4>.inwhile() {...}
```

Figure 7.2.: Multi-channel Iteration in a simple grid topology.

Figure 7.2 demonstrates multi-channel iteration for a simple grid topology.

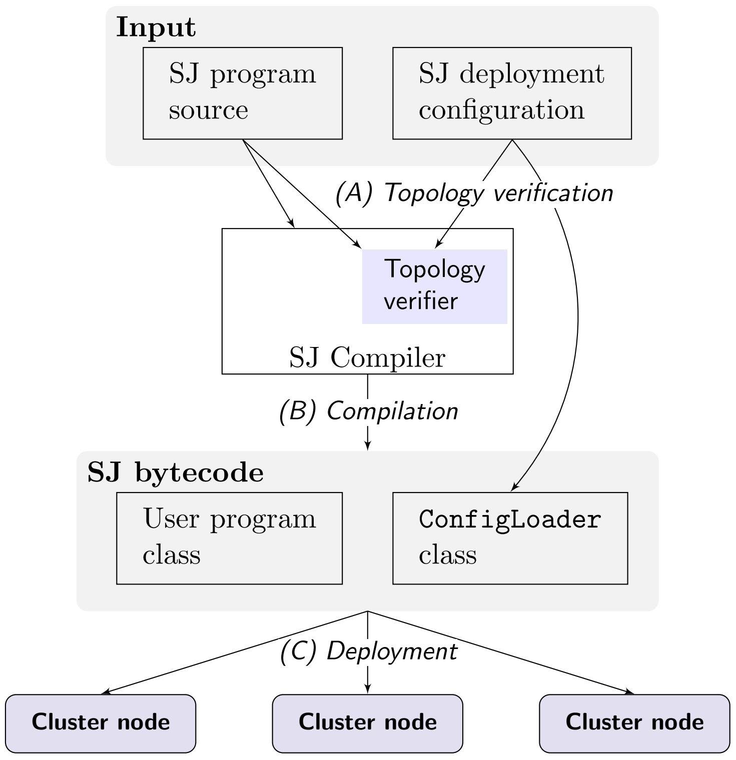


Figure 7.3.: The SJ tool chain.

Process *Master* controls the iteration on both the `s1` and `s2` session sockets under a single iteration condition. Processes *Forwarder1* and *Forwarder2* iterate following the signal from *Master* and forward the signal to *End*; thus, all four processes iterate in lockstep. Multi-channel `inwhile`, as performed by *End*, is intended for situations where multiple sessions are combined for iteration, but all are coordinated by an iteration signal from a common source; this means all the signals received from each socket of the `inwhile` will always agree — either to continue iterating, or to stop. In case this is not respected at run-time, the `inwhile` will throw an exception, resulting in session termination.

Together, multi-channel primitives enable the type-safe implementation of parallel programming patterns like scatter-gather, producer-consumer, and more complex chained topologies. The basic session primitives express only disjoint behaviour within individual sessions, whereas the multi-channel primitives implement interaction across multiple sessions as a single, integrated structure.

(5) The SJ tool chain with topology verification In previous work, the safety guarantees offered by the SJ compiler were limited to the scope of each independent *binary* (two-party) session. This means that, while any one session was guaranteed to be internally deadlock-free, this property may not hold in the presence of interleaved sessions in a process as a whole. The nodes in a parallel program typically make use of many interleaved sessions – with each of their neighbours in the chosen network topology. Furthermore, `inwhile` and `outwhile` in iteration chains must be correctly composed.

As a solution to this issue, we add a *topology verification* step to the SJ tool chain for parallel programs. Figure 7.3 summarises the SJ tool chain for developing type-safe SJ parallel program on a distributed computing cluster. An SJ parallel program is written as a collection of SJ source files, where each file corresponds to a role in the topology.

Topology verification (A) takes as input the source files and a *deployment configuration file*, listing the hosts where each process will be deployed and describing how to connect the processes. The sources and configuration files are then analysed statically to ensure the overall session topology of the parallel program conforms to a *well-formed topology* defined in Definition 7.1 in Section 7.4, and in conjunction with session duality checks in SJ, precludes *global deadlocks* in parallel SJ programs (see Theorem 7.3). The source files are then compiled (B) to bytecode, and (C) deployed on the target cluster using details on the configuration file to instantiate and establish sessions with their assigned neighbours, ensuring the runtime topology is constructed according to the verified configuration file, and therefore safe execution of the parallel program.

7.3. Parallel Algorithms in SJ

This section presents the SJ implementation of an n-body simulation implemented in a ring topology (Section 7.3.1), a Jacobi method for solving the Discrete Poisson Equation implemented in a mesh topology (Section 7.3.2) and a linear equation solver implemented in a wraparound mesh topology (Section 7.3.3). They are examples of complex communication topologies, and we use them to explain the benefits and usage of the new multi-channel primitives.

7.3.1. N-body simulation: Ring Topology

The following session type describes the communication protocol of our implementation of N-body simulation. This is the session type for a `Worker` 's interaction with its left neighbour.

```
protocol WorkerToLeft
  sbegin.         // Accept session request from left neighbour
  !<int>.         // Forward init counter to determine number of processes
  ?[              // Main loop (loop controlled by left neighbour)
   ?[             // Pipeline stages within each simulation step
     !<Particle[]>// Pass current particle state along the ring
   ]*
  ]*
```

Listing 7.1: The session type for Worker role of the n-body simulation.

The interaction with the right neighbour follows the dual protocol. The *`WorkerLast`* and *`Master`* nodes follow slightly different protocols, in order to close the ring structure and bootstrap the pipeline interaction.

In the SJ implementation, each node establishes two sessions with the left and right neighbours, and the iteration of every session in the pipeline is centrally controlled by the *`Master`* node. Without the multi-channel iteration primitives, there is no adequate way of closing the ring (sending data from the *`WorkerLast`* node to the *`Master`*); the only option is to open and close a temporary session with each iteration (Figure 7.5) [BHY10], an inefficient and counter-intuitive solution, as depicted on the left in Figure 7.4 (the loosely dashed line indicates the temporary connection).

By contrast, Figure 7.6 gives the implementation of the ring topology using a multi-`outwhile` at the `Master` node, and a multi-`inwhile` at `WorkerLast`. Data is still passed left-to-right, but the final iteration control link (the dashed arrow in Figure 7.4) is reversed. This allows the `Master` to create the final link just once (at the start of the algorithm) like the other links, and gives the `Master` full control over the whole pipeline.

The full process definition of the n-body simulation can be found in the Appendix (Section A.1.5).

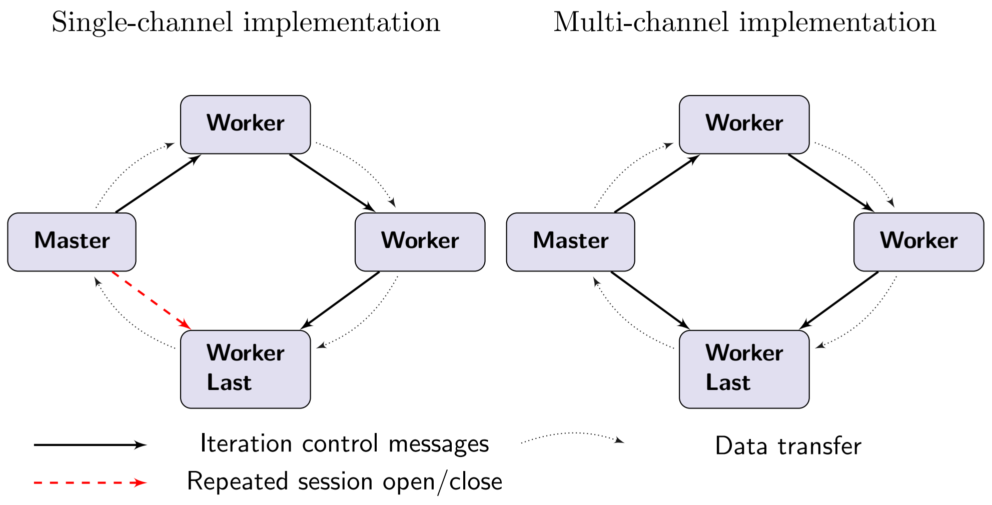

Single-channel implementation
Multi-channel implementation
Worker
Master
Worker
Worker
Last
Worker
Master
Worker
Worker
Last
Iteration control messages
Repeated session open/close
Data transfer

Figure 7.4.: Communication patterns in n-body implementations.

Master
```
right.outwhile(cond) {
  left = chanLast.request();
  right.send(data);
  processData();
  newData = left.receive();
}
```

Worker
```
right.outwhile(left.inwhile) {
  right.send(data);
  processData();
  newData = left.receive();
}
```

WorkerLast
```
left.inwhile {
  right = chanFirst.accept();
  right.send(data);
  processData();
  newData = left.receive();
}
```

Figure 7.5.: Implementation of the ring topology, single-channel primitives only.

Master
```
<left,right>.outwhile(cond) {

  right.send(data);
  processData();
  newData = left.receive();
}
```

Worker
```
right.outwhile(left.inwhile) {
  right.send(data);
  processData();
  newData = left.receive();
}
```

WorkerLast
```
<left,right>.inwhile {

  right.send(data);
  processData();
  newData = left.receive();
}
```

Figure 7.6.: Improved implementation of the ring topology using multi-channel primitives.

7.3.2. Jacobi solution of the discrete Poisson equation: Mesh Topology

Poisson's equation is a partial differential equation widely used in physics and the natural sciences. Jacobi's algorithm can be implemented using various partitioning strategies. An early session-typed implementation [BHY10] used a one-dimensional decomposition of the source matrix, resulting in a linear communication topology. The following demonstrates how the new multi-channel primitives are required to increase parallelism using a two-dimensional decomposition, i.e. using a 2D mesh communication topology. The mesh topology is used in a range of other parallel algorithms [CLR08].

The discrete two-dimensional Poisson equation $(\nabla^2 u)_{ij}$ for a $m \times n$ grid reads:

$$u_{ij} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - dx^2 g_{i,j})$$

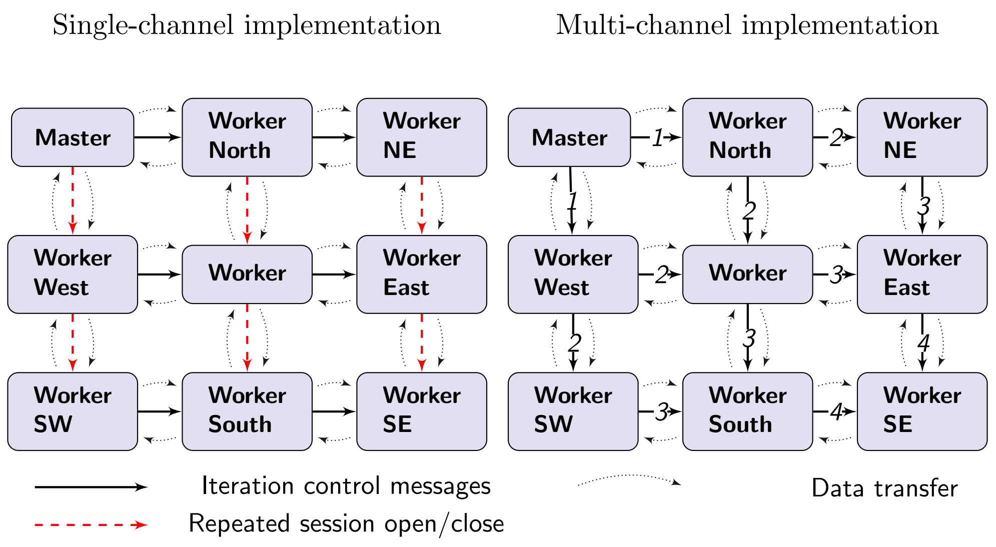
where $2 \leq i \leq m-1$, $2 \leq j \leq n-1$, and $dx = 1/(n+1)$. Jacobi's algorithm converges on a solution by repeatedly replacing each element of the matrix u by an adjusted average of its four neighbouring values and $dx^2 g_{i,j}$. For this example, we set each $g_{i,j}$ to 0. Then, from the k-th approximation of u, the next iteration calculates:

$$u_{ij}^{k+1} = \frac{1}{4}(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k)$$

Termination may be on reaching a target convergence threshold or on completing a certain number of iterations. Parallelisation of this algorithm exploits the fact that each element can be independently updated within each iteration. The decomposition divides the grid into subgrids, and each process will execute the algorithm for its assigned subgrid. To update the points along the boundaries of each subgrid, neighbouring processes need to exchange their boundary values at the beginning of each iteration.

```
protocol MasterToWorker
  cbegin.                 // Open a session with the Worker
  !<int>.!<int>.          // Send matrix dimensions
  ![                      // Main loop: checking convergence condition
    !<double[]>.          // Send our boundary values...
    ?(double[]).          // ..and receive our neighbour's
    ?(ConvergenceValues) // Convergence data for neighbouring subgrid
  ]*                      // (end of main loop)
```

Listing 7.2: The session type for the Master role of the Jacobi algorithm.



Single-channel implementation
Multi-channel implementation
Master
Worker North
Worker NE
Worker West
Worker
Worker East
Worker SW
Worker South
Worker SE
1
2
3
4
Iteration control messages
Repeated session open/close
Data transfer

Figure 7.7.: Communication patterns in Jacobi method implementations.

A 2D mesh implementation is shown in Figure 7.7. The `Master` node controls iteration from the top-left corner. Nodes in the centre of the mesh receive iteration control signals from their top and left neighbours, and propagate them to the bottom and right. Nodes at the edges only propagate iteration signals to the bottom or the right, and the final node at the bottom right only receives signals and does not propagate them further.

The session type for communication from the `Master` to either of the `Workers` under it or at its right is given in Listing 7.2. The `Worker`'s protocol for interacting with the `Master` is the dual of `MasterToWorker`; the same protocol is used for interaction with other `Workers` at their right and bottom (except for `Workers` at the edges of the mesh).

As listed in Figure 7.8, it is possible to express the complex 2D mesh using single-channel primitives only. However, this implementation suffers from a problem: without the multi-channel primitives, there is no way of sending iteration control signals both horizontally and vertically; the only option is to open and close a temporary session in every iteration (Figure 7.7), an inefficient and counter-intuitive solution. Moreover, the continuous nature of the vertical iteration sessions cannot be expressed naturally.

Having noted this weakness, Figure 7.9 lists a revised implementation, taking advantage of multi-channel `inwhile` and `outwhile`. The multi-channel

Master

```
// right: WorkerNorth
// under: WorkerWest
right.outwhile(!converged()) {
 below = chanBelow.request();
 sndBounds(right,below);
 rcvBounds(right,below);
 compute(rcvRight,rcvBelow);
 rcvConvergenceVal(right,below);
}
```

Worker

```
// above: WorkerNorth
// right: WorkerEast
// below: WorkerSouth
// left: WorkerWest
right.outwhile(left.inwhile) {
 above = chanAbove.accept();
 below = chanBelow.request();
 sndBounds(left,right,above,below);
 rcvBounds(left,right,above,below);
 compute(rcvLeft,rcvRight,rcvAbove,
      rcvBelow);
 sndConvergenceVal(left,above);
}
```

WorkerSE

```
// left: WorkerSouth
// above: WorkerEast
left.inwhile {
 above = chanAbove.request();
 sndBounds(left,above);
 rcvBounds(left,above);
 compute(rcvLeft,rcvAbove);
 sndConvergenceVal(left,above);
}
```

Figure 7.8.: Jacobi method implementation in a 2D mesh topology with single-channel primitives.

new Master

```
// right: WorkerNorth
// under: WorkerWest
<below,right>.outwhile(!converged()){

 sndBounds(right,below);
 rcvBounds(right,below);
 compute(rcvRight,rcvBelow);
 rcvConvergenceVal(right,below);
}
```

new Worker

```
// above: WorkerNorth
// right: WorkerEast
// below: WorkerSouth
// left: WorkerWest
<below,right>.outwhile(
    <above,left>.inwhile) {

  sndBounds(left,right,above,below);
  rcvBounds(left,right,above,below);
  compute(rcvLeft,rcvRight,rcvAbove,
       rcvBelow);
  sndConvergenceVal(left, above;
}
```

new WorkerSE

```
// left: WorkerSouth
// above: WorkerEast
<above,left>.inwhile {

 sndBounds(left, above);
 rcvBounds(left, above);
 compute(rcvLeft,rcvAbove);
 sndConvergenceVal(left, above);
}
```

Figure 7.9.: Jacobi method implementation in a 2D mesh topology with *multi-channel* primitives.

`inwhile` allows each `Worker` to receive iteration signals from the two processes at its top and left. Multi-channel `outwhile` lets a process control both processes at the right and bottom. Together, these two primitives completely eliminate the need for repeated opening and closing of intermediary sessions in the single-channel version. The resulting implementation is clearer and also much faster. See Section 7.5 for the benchmark results.

7.3.3. Linear Equation Solver: Wraparound Mesh Topology

We implement a parallel linear equation solver using the Jacobi method. The algorithm is introduced in Section 3.4.2. Our implementation uses p^2 processors in a $p \times p$ wrap-around mesh topology to solve an $n \times n$ system matrix. The matrix is partitioned into submatrix blocks of size $\frac{n}{p} \times \frac{n}{p}$, assigned to each of the processors (see Figure 7.10).

Each iteration of the algorithm requires multiplications (in the term $\alpha_{ij}x_j$) and summation. Multiplications dominate execution time here, hence the parallelisation concentrates on them. The horizontal part of the mesh acts as a collection of circular pipelines for multiplications. Their results are collected by the diagonal nodes, which perform the summation and the division by α_{ii}.

This gives the updated solution values for the iteration. These need to be communicated to other nodes for the next iteration. The vertical mesh connections are used for this purpose: the solution values are sent down by the diagonal node, and each worker node picks up the locally required solution values, and passes on the rest. The transmission wraps around at the bottom of the mesh, and stops at the node immediately above the diagonal, hence the lack of connectivity between the two in Figure 7.10.

Note that contrary to the non-wraparound 2D-mesh of Section 7.3.2, the sink of this well-formed topology (Section 7.4.3) is not the last node on the diagonal, but instead the node just above, called `WorkerEastLast`. This is because the diagonal nodes transmit updated values as explained above, and this transmission stops just before a complete wraparound. Figure A.2 shows node ranks for the wraparound mesh topology, along with the other topologies presented in this chapter.

Master

```
// Source node
// Initiates computation
<below,right>.outwhile(!converged()){
  prod = computeProducts();
  // horizontal ring
  ringData = prod;
  <left,right>.outwhile(count<
       nodesOnRow) {
    right.send(ringData);
    ringData = left.receive();
    computeSums(ringData);
    count++;
  }
  // pass results to diagonal node
  newX = computeDivision();
  below.send(newX);
}
```

WorkerDiagonal

```
// Diagonal Workers
// Receive result, compute,
// then propagate to next row
<below,right>.outwhile(
    left.inwhile) {
  prod = computeProducts();
  ringData = prod;
  right.outwhile(left.inwhile) {
    right.send(ringData);
    ringData = left.receive();
    computeSums(ringData);
  }
  newX = computeDivision();
  below.send(newX);
}
```

Worker

```
// Workers
// Calculate in ring
// then forward from above to below
<below,right>.outwhile(
    <left,above>.inwhile) {
  prod = computeProducts();
  ringData = prod;
  right.outwhile(left.inwhile) {
    right.send(ringData);
    ringData = left.receive();
  }
  newX = above.receive();
  below.send(newX);
}
```

WorkerEastLast

```
// Sink node
// Receives from above, left and right
<right,left,above>.inwhile {
  prod = computeProducts();
  ringData = prod;
  <left,right>.inwhile {
    right.send(ringData);
    ringData = left.receive();
  }
  newX = above.receive();
}
```

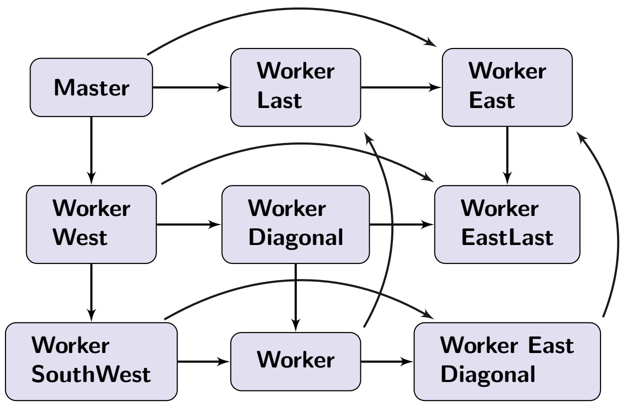


Figure 7.10.: Linear Equation Solver: Wraparound Mesh Topology.

7.4. Multi-channel Session Calculus

This section formalises the new nested iterations and multi-channel communication primitives and proves correctness of our implementation. Our proof method consists of:

1. We first define programs (i.e. starting processes) including the new primitives, and then define operational semantics with running processes modelling intermediate session communications.

2. We define a typing system for programs and running processes.

3. We prove that if a group of running processes conforms to a *well-formed topology*, then they satisfy the subject reduction theorem (Theorem 7.1) which implies type and communication-safety (Theorem 7.2) and deadlock-freedom (Theorem 7.3).

4. Since programs for our chosen parallel algorithms conform to a well-formed topology, we conclude that they satisfy the above three properties.

7.4.1. Syntax

The session calculus we treat extends the one presented in [HVK98].

Figure 7.11 defines its syntax. Channels $(u, u', ...)$ can be either of two sorts: *shared channels* (a, b, x, y) or *session channels* $(k, k', ...)$. Shared channels are used to open a new session. In accepting and requesting processes, the name a represents the public interaction point over which a session may commence. The bound variable k represents the actual channel over which the session communications will take place. Constants $(c, c', ...)$ and expressions $(e, e', ...)$ of ground types (booleans and integers) are also added to model data. *Selection* chooses an available branch, and *branching* offers alternative interaction patterns; *channel send* and *channel receive* enable session delegation [HVK98]. The *sequencing*, written P; Q , meaning that P is executed before Q. This syntax allows for complex forms of synchronisation, joining, and forking since P can include any parallel composition of arbitrary processes. The second addition is that of *multicast inwhile* and *outwhile*, following SJ syntax. Note that the definition of expressions includes multicast inwhile $\langle k_1 \ldots k_n \rangle.\mathsf{inwhile}$, in order to allow inwhile as an outwhile loop condition. The control message $k \dagger [b]$ created by outwhile appears only at runtime.

The precedence of the process-building operators is (from the strongest) "$\triangleleft, \triangleright, \{\}$", ".", ";" and "|". Moreover we define that "." associates to the right. The binders for channels and variables are standard.

The process definition is modified to include an *Err* process which represents a *while condition mismatch* in multicast *inwhile* and *outwhile. while condition mismatch* is further explained in the operational semantics in the next section (Section 7.4.2).

7.4.2. Operational Semantics

The operational semantics are based on the reduction relation $\rightarrow$, and the reduction rules are given in Figure 7.12.

Structural Congruence The session calculus is π-calculus extended with session primitives [HVK98], so definition of structural congruence $\equiv$ is similar to π-calculus. Below lists the structural congruence rules in session calculus:

$$P \equiv Q \text{ if } P \equiv_\alpha Q \quad P \mid \mathsf{0} \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\boldsymbol{\nu}\, u)(P) \mid Q \equiv (\boldsymbol{\nu}\, u)((P \mid Q)) \quad \text{ if } u \notin \mathsf{fn}(Q)$$
$$(\boldsymbol{\nu}\, u)(\mathsf{0}) \equiv \mathsf{0} \qquad \mathsf{def}\, D \,\mathsf{in}\, \mathsf{0} \equiv \mathsf{0} \qquad \mathsf{0};\ P \ \equiv P$$
$$(\boldsymbol{\nu}\, u)(\mathsf{def}\, D \,\mathsf{in}\, P) \equiv \mathsf{def}\, D \,\mathsf{in}\, (\boldsymbol{\nu}\, u)(P) \quad \text{ if } u \notin \mathsf{fn}(D)$$
$$(\mathsf{def}\, D \,\mathsf{in}\, P) \mid Q \equiv \mathsf{def}\, D \,\mathsf{in}\, (P \mid Q) \quad \text{ if } \mathsf{fpv}(D) \cap \mathsf{fpv}(Q) = \emptyset$$
$$\mathsf{def}\, D \,\mathsf{in}\, \mathsf{def}\, D' \,\mathsf{in}\, P) \equiv \ \mathsf{def}\, D \ \mathsf{and} D' \,\mathsf{in}\, P \quad \text{ if } \mathsf{fpv}(D) \cap \mathsf{fpv}(D') = \emptyset.$$

Evaluation Context To keep session reasoning simple, we introduce evaluation contexts. Evaluation contexts isolate subprocesses and allow subprocesses to reduce independent of influences external to the context. Our evaluation contexts are defined below:

	(Processes)	
P	$::= \mathsf{0}$	inaction
	$\mid T$	prefixed
	$\mid P;\ Q$	sequence
	$\mid P \ \mid\ Q$	parallel
	$\mid (\boldsymbol{\nu}\, u)(P)$	hiding
	(Prefixed processes)	
T	$::= \overline{a}(k).P$	request
	$\mid a(k).P$	accept
	$\mid \overline{k}\langle e\rangle$	sending
	$\mid k(x).P$	reception
	$\mid \overline{k}\langle k'\rangle$	sending
	$\mid k(k').P$	reception
	$\mid X[ek]$	variables
	$\mid \mathsf{def}\, D\, \mathsf{in}\, P$	recursion
	$\mid k \ \lhd\ l;$	selection
	$\mid k \ \rhd\ \{l_i\colon\ P_i\}_{i\in\{1..n\}}$	branch
	$\mid \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q$	conditional
	$\mid \langle k_1 \ldots k_n\rangle.\mathsf{inwhile}\{Q\}$	inwhile
	$\mid \langle k_1 \ldots k_n\rangle.\mathsf{outwhile}(e)\{P\}$	outwhile
	$\mid k \dagger [b]$	message
	(Declaration)	
D	$::= \{X_i(x_i k_i)\ =\ P_i\}_{i\in\{1..n\}}$	declaration
	(Values)	
v	$::= a, b, x, y$	shared names
	$\mid \mathsf{true}, \mathsf{false}$	boolean
	$\mid n$	integer
	(Expressions)	
e	$::= v\ \mid$	value
	$\mid e + e'$	sum
	$\mid \mathsf{not}(e)$	not
	$\mid \langle k_1 \ldots k_n\rangle.\mathsf{inwhile}$	inwhile
	$\mid \ \ldots$	

Figure 7.11.: Multi-channel Session Calculus: Syntax.

$$a(k).P_1 \mid \overline{a}(k).P_2 \to (\boldsymbol{\nu}\, k)(P_1 \mid P_2) \qquad \text{[Link]}$$

$$\overline{k}\langle c\rangle \mid k(x).P_2 \to P_2\{c/x\} \qquad \text{[Com]}$$

$$k \triangleright \{l_i\colon P_i\}_{i\in\{1..n\}} \mid k \triangleleft l_i \to P_i \quad (1 \le i \le n) \qquad \text{[Lbl]}$$

$$\overline{k}\langle k'\rangle \mid k(k').P_2 \to P_2 \qquad \text{[Pass]}$$

$$\mathsf{if\ true\ then}\ P\ \mathsf{else}\ Q \to P \qquad \mathsf{if\ false\ then}\ P\ \mathsf{else}\ Q \to Q \qquad \text{[If]}$$

$$\{X_i(x_i k_i) = P_i\}_{i\in\{1..n\}}\ \mathsf{in}\ X[ck] \to \{X_i(x_i k_i) = P_i\}_{i\in\{1..n\}}\ \mathsf{in}\ P\{c/x\} \qquad \text{[Def]}$$

$$\langle k_1 \dots k_n\rangle.\mathsf{inwhile}\{P\} \mid \Pi_{i\in\{1..n\}}(k_i \dagger [\mathsf{true}]) \to P; \langle k_1 \dots k_n\rangle.\mathsf{inwhile}\{P\} \qquad \text{[Iw1]}$$

$$\langle k_1 \dots k_n\rangle.\mathsf{inwhile}\{P\} \mid \Pi_{i\in\{1..n\}}(k_i \dagger [\mathsf{false}]) \to \mathsf{0} \qquad \text{[Iw2]}$$

$$E[\langle k_1 \dots k_n\rangle.\mathsf{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\mathsf{true}] \to E[\mathsf{true}] \qquad \text{[IwE1]}$$

$$E[\langle k_1 \dots k_n\rangle.\mathsf{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\mathsf{false}] \to E[\mathsf{false}] \qquad \text{[IwE2]}$$

$$E[e] \to^* E'[\mathsf{true}] \Rightarrow \qquad \text{[Ow1]}$$
$$E[\langle k_1 \dots k_n\rangle.\mathsf{outwhile}(e)\{P\}] \to E'[P; \langle k_1 \dots k_n\rangle.\mathsf{outwhile}(e)\{P\}]$$
$$\mid \Pi_{i\in\{1..n\}} k_i \dagger [\mathsf{true}]$$

$$E[e] \to^* E'[\mathsf{false}] \Rightarrow$$
$$E[\langle k_1 \dots k_n\rangle.\mathsf{outwhile}(e)\{P\}] \to E'[\mathsf{0}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\mathsf{false}] \qquad \text{[Ow2]}$$

$$P \equiv P' \text{ and } P' \to Q' \text{ and } Q' \equiv Q \Rightarrow P \to Q \qquad \text{[Str]}$$

$$e \to e' \Rightarrow E[e] \to E[e'] \qquad P \to P' \Rightarrow E[P] \to E[P']$$
$$P \mid Q \to P' \mid Q' \Rightarrow E[P] \mid Q \to E[P'] \mid Q' \qquad \text{[Eval]}$$

In [Ow1] and [Ow2], we assume $E = E' \mid \Pi_{i\in\{1..n\}} k_i \dagger [b_i]$

Figure 7.12.: Multi-channel Session Calculus: Reduction rules.

Evaluation context E	$::= []$	
	$\mid E;\ P$	sequential composition
	$\mid E \mid P$	parallel composition
	$\mid (\boldsymbol{\nu}\, u)(E)$	hiding
	$\mid \mathsf{def}\ D\ \mathsf{in}\ E$	recursion
	$\mid \mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
	$\mid \langle k\rangle.\mathsf{outwhile}(E)\{P\}$	outwhile
	$\mid E + e \mid \dots$	expressions

Reduction Rules We use the shorthand notation $\Pi_{i\in\{1..n\}}P_i$ to denote the parallel composition of $(\ P_1 \mid \cdots \mid P_n\ )$.

Rule [Link] is a session initiation rule where a fresh channel k is created, then restricted because the leading parts now share the channel k to start private interactions. Rule [Com] sends data. Rule [Lbl] selects the i-th branch, and rule [Pass] passes a session channel k for delegation. The standard conditional and recursive agent rules [If1], [If2] and [Def] originate in [HVK98].

Rule [Iw1] synchronises with n asynchronous messages if they all carry true. In this case, it repeats again.

Rule [Iw2] is its dual and synchronises with n false messages. In this case, it moves to the next command. On the other hand, if the results are mixed (i.e. b_i is true, while b_j is false), then it is stuck. In SJ, it will raise the exception, cf. § 7.2 (4). The rules for expressions are defined similarly. The rules for outwhile generate appropriate messages. Note that the assumption $E[e] \to E'[\mathsf{true}]$ or $E[e] \to E'[\mathsf{false}]$ is needed to handle the case where e is an inwhile expression.

In order for our reduction rules to reflect SJ's actual behaviour, inwhile rules should have precedence over outwhile rules. Note that our algorithms do not cause an infinite generation of $k \dagger [b]$ by outwhile: this is ensured by the well-formed topology criteria described later, together with this priority rule.

7.4.3. Types, Typing System and Well-Formed Topologies

This subsection presents types and typing systems. The key point is an introduction of types and typing systems for asynchronous runtime messages. We then define the notation of a well-formed topology.

Types The type system in this section is designed to guarantee communication correctness and liveness property with the new syntax and operational semantics. The full type syntax is given below:

Sorts contain the standard types and the pair of dual sessions $\langle \alpha, \overline{\alpha} \rangle$.

Partial Session Types are session types that does not include the end type. Partial session types are distinguished from completed session types so that they can be sequentially composed.

Completed Session Types are types that end with end or are equal to $\bot$.

Sort S	$::= \mathsf{nat} \mid \mathsf{bool} \mid \langle \alpha, \overline{\alpha} \rangle$	
Partial session τ	$::= \epsilon$	empty
	$\mid \tau;\, \tau$	sequential composition
	$\mid\, ![S]$	send
	$\mid\, ?[S]$	receive
	$\mid\, \oplus \{l_i : \tau_i\}_{i \in \{1..n\}}$	selection
	$\mid\, \& \{l_i : \tau_i\}_{i \in \{1..n\}}$	branching
	$\mid\, ![\alpha]$	session delegation
	$\mid\, ?[\alpha]$	session receive
	$\mid\, ![\tau]^*$	outwhile
	$\mid\, ?[\tau]^*$	inwhile
	$\mid \mu\, \mathbf{t}.\tau \mid \mathbf{t}$	recursion
Completed session α	$::= \tau.\mathsf{end}$	inaction
	$\mid \bot$	bottom
Runtime session β	$::= \alpha$	completed session
	$\mid \alpha^{\dagger}$	unconsumed message
	$\mid \dagger$	message

In above syntax, $![\alpha]$ and $?[\alpha]$ are session delegation and session receive respectively. This makes use of the name-passing property from π-calculus that allows sending and receiving of channels (or *sessions* in the session calculus). The same typing syntax is used for ordinary type sending and receiving ($![S]$, $?[S]$). Iteration types ($?[\tau]^*$ and $![\tau]^*$) are introduced for $\langle a \rangle.\mathsf{inwhile}\{n\}$ d $\langle r \rangle.\mathsf{outwhile}(e)\{s\}$ pectively. With iteration types, the partial type definition τ can be repeated for a number of times until the $\langle c \rangle.\mathsf{outwhile}(o)\{n\}$ dition is no longer true.

In the syntax given, $\& \{\tau_i : ._i\}_{i \in \{1..n\}}\mathsf{end} \equiv \& \{l_1 :: \tau_1.\mathsf{end}, \ldots, l_n : \tau_n.\mathsf{end}\}$. This equivalence ensures all partial types $\tau_1 \ldots \tau_n$ of label selection choices ends and are compatible with each other in the completed session type (and vice versa).

ϵ is an empty type, and it is defined so that $\epsilon; \tau \equiv \tau$ and $\tau; \epsilon \equiv \tau$. The two equivalences allows us to continue reducing when one of the two processes $P; Q$ reduces to empty.

Runtime session syntax represents partial composed runtime message types. $\alpha^{\dagger}$ represents the situation inwhile or outwhile are composed with messages; and $\dagger$ is a type of messages. The meaning will be clearer when we define the parallel composition.

Judgements and Environments The typing judgements for expressions and processes are of the shape:

$$\Gamma; \Delta \vdash e \triangleright S \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta$$

where we define the environments as $\Gamma ::= \emptyset \mid \Gamma \cdot x\colon S \mid \Gamma \cdot X : S\alpha$ and $\Delta ::= \emptyset \mid \Delta \cdot k\colon \beta$. Γ is the *standard environment* which associates a name to a sort and a process variable to a sort and a session type. Δ is the *session environment* which associates session channels to running session types, which represents the open communication protocols. We often omit Δ or Γ from the judgement if it is empty.

Sequential and parallel compositions of environments are defined as:

$$\begin{array}{rcl} \Delta; \Delta' & = & \Delta \backslash \mathsf{dom}(\Delta') \\ & & \cup \Delta' \backslash \mathsf{dom}(\Delta) \\ & & \cup \{k\colon \Delta(k) \setminus \mathsf{end}; \Delta'(k) \mid k \in \mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta')\} \\ \Delta \circ \Delta' & = & \Delta \backslash \mathsf{dom}(\Delta') \\ & & \cup \Delta' \backslash \mathsf{dom}(\Delta) \\ & & \cup \{k\colon \Delta(k) \circ \Delta'(k) \mid k \in \mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta')\} \end{array}$$

where $\Delta(k) \setminus \mathsf{end}$ means we delete end from the tail of the types (e.g. $\tau.\mathsf{end} \setminus \mathsf{end} = \tau$). Then the resulting sequential composition is always well-defined. The parallel composition of the environments must be extended with new running message types. Hence $\beta \circ \beta'$ is defined as either (1) $\alpha \circ \overline{\alpha} = \perp$; (2) $\alpha \circ \dagger = \alpha^{\dagger}$ or (3) $\alpha \circ \overline{\alpha}^{\dagger} = \perp^{\dagger}$. Otherwise the composition is undefined. Here $\overline{\alpha}$ denotes a dual of α (defined by exchanging ! to ? and & to $\oplus$; and vice versa). (1) is the standard rule from session type algebra, which means once a pair of dual types are composed, then we cannot compose any processes with the same channel further. (2) means a composition of an iteration of type α and n-messages of type $\dagger$ becomes $\alpha^{\dagger}$. This is further composed with the dual $\overline{\alpha}$ by (3) to complete a composition. Note that $\perp^{\dagger}$ is different from $\perp$ since $\perp^{\dagger}$ represents a situation that messages are not consumed with inwhile yet.

7.4.4. Typing Rules

We explain the key typing rules for the new primitives below. The full list of rules can be found in Figure A.1 in Section A.1.1 in the Appendix.

$$\frac{\Delta = k_1 : ?[\tau_1]^{*}.\mathsf{end}, ..., k_n : ?[\tau_n]^{*}.\mathsf{end}}{\Gamma; \Delta \vdash \langle k_1 \ldots k_n \rangle.\mathsf{inwhile} \triangleright \mathsf{bool}} \quad \frac{\Gamma \vdash b \triangleright \mathsf{bool}}{\Gamma \vdash k \dagger [b] \triangleright k : \dagger} \quad \text{[EInwhile],[Message]}$$

$$\frac{\Gamma;\ \Delta \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \cdot k_1 : \tau_1.\mathsf{end} \cdot \cdots \cdot k_n : \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots k_n \rangle.\mathsf{outwhile}(e)\{P\} \triangleright \Delta \cdot k_1 : ![\tau_1]^{*}.\mathsf{end}, ..., k_n : ![\tau_n]^{*}.\mathsf{end}}$$

$$\text{[Outwhile]}$$

$$\frac{\Gamma \vdash Q \triangleright \Delta \cdot k_1 : \tau_1.\mathsf{end} \cdot \cdots \cdot k_n : \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots k_n \rangle.\mathsf{inwhile}\{P\} \triangleright \Delta \cdot k_1 : ?[\tau_1]^{*}.\mathsf{end}, ..., k_n : ?[\tau_n]^{*}\mathsf{end}} \quad \text{[Inwhile]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P;\ Q \triangleright \Delta; \Delta'} \quad \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \quad \text{[Seq],[Conc]}$$

[EInwhile] is a rule for inwhile-expression. The iteration session type of k_i is recorded in Δ. This information is used to type the nested iteration with outwhile in rule [Outwhile]. Rule [Inwhile] is dual to [Outwhile]. Rule [Message] types runtime messages as $\dagger$. Sequential and parallel compositions use the above algebras to ensure the linearity of channels.

7.4.5. Well-formed Topologies

We now define the well-formed topologies. Since our multi-channel primitives offer an effective, structured message passing synchronisation mechanism, the following simple definition is sufficient to capture deadlock-freedom in representative topologies for parallel algorithms. Common topologies in parallel algorithms such as circular pipeline (ring), mesh and wraparound mesh all conform to our well-formed topology definition below. The definition of well-formed ring (Definition A.1) and well-formed mesh topology (Definition A.2) is defined in the Appendix (Section A.1.2), and we show that they both conform to our definition of well-formed topology (Definition 7.1). Below we call P is a *base* if P is either $\mathsf{0}$, $\overline{k}\langle e\rangle$, $k(x).\mathsf{0}$, $k \lhd l$ or $k \rhd \{l_i\colon \mathsf{0}_i\}_{i\in\{1..n\}}$.

Definition 7.1 (Well-formed topology.) Suppose a group of n parallel composed processes $P = P_1 \mid \ldots \mid P_n$ such that $\Gamma \vdash P \triangleright \Delta$ with $\Delta(k) = \bot$ for all $k \in \mathsf{dom}(\Delta)$; and $k_{(i,j)}$ denotes a free session channel from P_i to P_j[1]. We say P conforms to a *well-formed topology* if P inductively satisfies one of the following conditions:

1. (inwhile and outwhile)

$$\begin{array}{rcl} P_1 & = & \langle\tilde{k}\rangle.\mathsf{outwhile}(e)\{Q_1\} \\ P_i & = & \langle\tilde{k_i}\rangle.\mathsf{outwhile}(\langle\tilde{k_i}\rangle.\mathsf{inwhile})\{Q_i\}\ (2 \le i < n) \\ P_n & = & \langle\tilde{k}'_n\rangle.\mathsf{inwhile}\{Q_n\} \\ & & \tilde{k_i} \subset k_{(i,i+1)}\cdots k_{(i,n)}, \tilde{k}'_i \subset k_{(1,i)}\cdots k_{(i-1,i)} \end{array}$$

   and $(Q_1 \mid \cdots \mid Q_n)$ conforms to a well-formed topology.

2. (sequencing)

$$P_i = Q_{1i}; ...; Q_{mi}$$

   where $(Q_{j1} \mid Q_{j2} \mid \cdots \mid Q_{jn})$ conforms to a well-formed topology for each $1 \le j \le m$.

3. (base) (1) session actions in P_i follow the order of the index (e.g. the session actions at $k_{(i,j)}$ happen before $k_{(h,g)}$ if $(i,j) < (h,g)$), then the rest is a base process P'_i; and (2) P_i includes neither shared session channels, inwhile nor outwhile.

Intuition The core idea behind the well-formed topology is to ensure at any iteration of the parallel program, all processes are synchronised by the same iteration condition. For example, when one of the processes in an iterative algorithm decides that the results has converged, all other processes in the parallel program should all terminate in the next iteration so there is no communication mismatch between the processes.

[1] P_i is a process which uses outwhile and P_j is a process which uses inwhile on the session $k_{(i,j)}$.

In order to capture this core idea in a group of parallel processes, each process is treated as a node in a directed acyclic graph (DAG) of connected processes. The directed edges between nodes represent the flow of control messages between processes, where edges out of a node are sessions with outwhile (sending a control message out) and edges into a node are sessions with inwhile (receive a control message and use as loop condition).

For this graph to follow a well-formed topology, i.e. all nodes synchronised by a single flow of control messages, the graph must be a DAG which has a single source node (only outwhile) and a single sink node (only inwhile). This ensures, at each iteration, there is only one node that can send out a control message (source node) and a only one node to consume the control message and end the global iteration (sink node). To express this condition, our definition of a well-formed topology gives all processes a partial ordering (i.e. Figure 7.13), where all processes in a well-formed topology must fit into, and the control messages flow from one end to another.

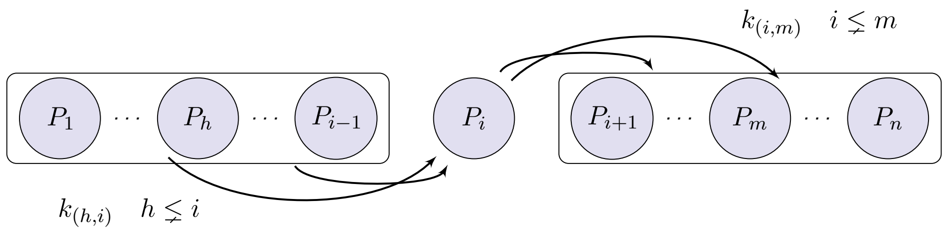


Figure 7.13.: Multi-channel ordering.

Figure 7.13 explains condition (1) of the above definition, ensuring consistency of control flows within iterations. Subprocesses P_i are ordered by their process index i. A process P_i can only send outwhile control messages to processes with a higher index via $\vec{k}_i$ (denoted by $k_{(i,m)}$), while it can receive messages from those with a lower index via $\vec{k}'_i$ (denoted by $k_{(h,i)}$). This ordering guarantees absence of cycles of communications.

There is only one source P_1 (only sends outwhile control messages) and one sink P_n (only receives those messages). (2) says that a sequential composition of well-formed topologies is again well-formed. (3) defines base cases which are commonly found in the algorithms: (3-1) means that since the order of session actions in P_i follow the order of the indices, $\Pi_i P_i$ reduces to $\Pi_i P'_i$ without deadlock; then since $\Pi_i P'_i$ is a parallel composition of base processes where each channel k has type $\perp$, $\Pi_i P'_i$ reduces to $\mathtt{0}$ without deadlock. (3-2) ensures a single global topology.

7.4.6. Subject Reduction, Communication Safety and Deadlock Freedom

We state here that process groups conforming to a well-formed topology satisfy the main theorems. The full proofs can be found in Section A.1.3 in the Appendix.

Theorem 7.1 (Subject Reduction) *Assume P forms a well-formed topology and $\Gamma \vdash P \triangleright \Delta$. Suppose $P \to^* P'$. Then we have $\Gamma \vdash P' \triangleright \Delta'$ with $\forall k$*

(1) $\Delta(k) = \alpha$ *implies* $\Delta'(k) = \alpha^\dagger$*; or*

(2) $\Delta(k) = \alpha^\dagger$ *implies* $\Delta'(k) = \alpha$*; or*

(3) $\Delta(k) = \beta$ *implies* $\Delta'(k) = \beta$.

(1) and (2) state an intermediate stage where messages are floating; or (3) the type is unchanged during the reduction. The proof requires to formulate the intermediate processes with messages which are started from a well-formed topology, and prove they satisfy the above theorem.

Proof idea Given a group of parallel processes that follow the well-formed topology (i.e. control messages are consistent between iterations), we need to show that the reduction of the processes following the reduction rules does not get stuck. Our proof focuses on the interaction between multi-channel inwhile and outwhile, where we analyse the reduction of a generic well-formed topology, which is a parallel composition between a process with outwhile only (source), processes with both outwhile and inwhile, and processes with inwhile only (sink). We consider two cases: (1) when the iteration condition is true and (2) when the iteration condition is false. In both cases, the source creates a runtime construct representing the control message ($k \dagger$ true or $k \dagger$ false) and passes the control message to all other processes. After its successful multi-step reduction to 0, all control messages are consumed by the sink so they do not interfere with subsequent iterations. Theorem 7.1 holds at every step of the reduction, and the properties below simply follow from the results of the subject reduction proof.

We say a process has a *type error* if expressions in P contain either a type error of values or constants in the standard sense (e.g. $\mathsf{if}\,100\,\mathsf{then}\,P\,\mathsf{else}\,Q$).

To formalise communication safety, we need the following notions. Write $\mathsf{inwhile}\ (Q)$ for either $\mathsf{inwhile}$ expression or $\mathsf{inwhile}\{Q\}$. We say that a process P is a *head subprocess* of a process Q if $Q \equiv E[P]$ for some evaluation context E. Then *k-process* is a head process prefixed by subject k (such as $\overline{k}\langle e\rangle$). Next, a *$k$-redex* is the parallel composition of a pair of k-processes. i.e. either in form of a pair such that $(\overline{k}\langle e\rangle, k(x).Q)$, $(k \lhd l, k \rhd \{l_i\colon Q_i\}_{i\in\{1..n\}})$, $(\overline{k}\langle k'\rangle;, k(k').P)$, $(\langle k_1 \ldots k_n\rangle.\mathsf{outwhile}(e)\{P\}, \langle k_1 \ldots k_n\rangle.\mathsf{inwhile}\{Q\})$ with $k \in \{k_1, .., k_n\} \cap \{k'_1, .., k'_m\}$ or $(k\dagger[b] \mid \langle k'_1 \ldots\ k'_m\rangle.\mathsf{inwhile}(Q))$ with $k \in \{k_1, .., k_n\}$. Then P is a *communication error* if $P \equiv (\boldsymbol{\nu}\,\tilde{u})(\mathsf{def}\,P\,\mathsf{in}\,Q\ \mid\ R)$ where Q is, for some k, the parallel composition of two or more k-processes that do not form a k-redex. The following theorem is direct from the subject reduction theorem [YV07, Theorem 2.11].

Theorem 7.2 (Type and Communication Safety) *A typable process which forms a well-formed topology never reduces to a type nor communication error.*

Below we say P is *deadlock-free* if for all P' such that $P \to^* P'$, $P' \to$ or $P' \equiv \mathsf{0}$. The following theorem shows that a group of typable multiparty processes which form a well-formed topology can always move or become the null process.

Theorem 7.3 (Deadlock Freedom) *Assume P forms a well-formed topology and $\Gamma \vdash P \rhd \Delta$. Then P is deadlock-free.*

7.5. Performance Evaluation

This section presents performance results for implementations of the n-Body simulation and Jacobi solution presented in Section 7.3.1 and 7.3.2 respectively. We evaluated our implementations on a 9-node cluster for our benchmark, and each of the points is an average of 4 runs of the benchmark. All of them comprise an AMD PhenomX4 9650 2.30GHz CPU with 8GB RAM. The main objectives of these benchmarks is (1) to investigate the benefits of the new multi-channel primitives, comparing Old SJ (without the new primitives) and Multi-channel SJ (with the new primitives); and (2)

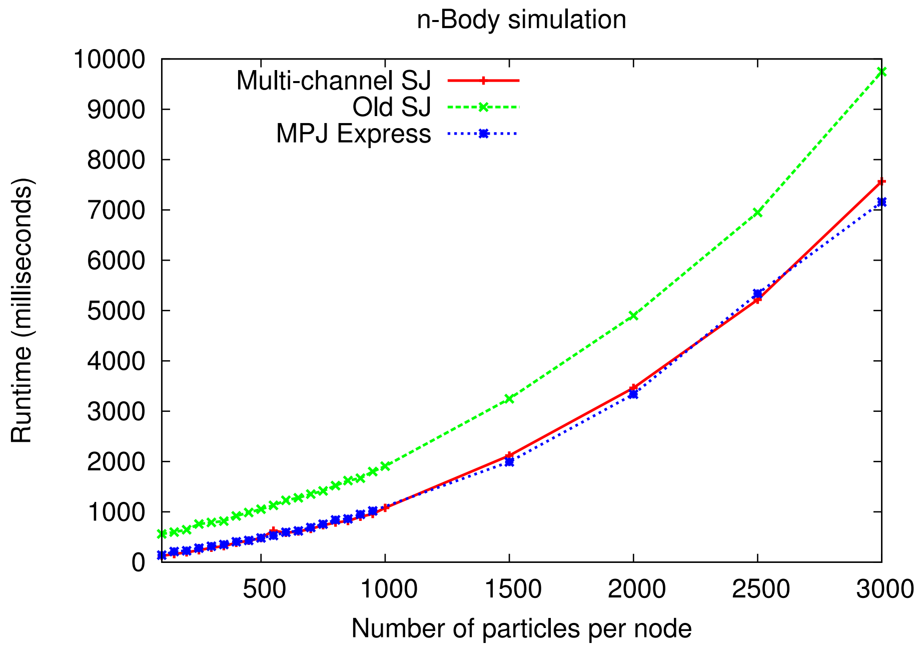


Figure 7.14.: Benchmark results: 3 nodes n-body simulation.

compare those with MPJ Express [BC00] for reference. Figure 7.15 shows a clear improvement when using the new multi-channel primitives in SJ. Multi-channel SJ also performs competitively against MPJ Express in both benchmarks. Hence SJ can be a viable alternative to MPI programming in Java, with the additional assurances of communication-safety and deadlock-free.

7.6. Summary and Discussion

This chapter introduced multi-channel session primitives and its formalisation. Combining with an additional communication topology verification, the approach offers an alternative to programming with a global view offered by MPST. It represents a new way of parallel programming, by chaining binary sessions to construct a global topology for parallel applications.

Our approach gives a clear definition of a class of communication-safe and deadlock-free programs as proved in Theorems 7.2 and 7.3, which have been statically checked without exploring all execution states for all possible thread interleavings. Our formalisation of the primitives enabled us to define

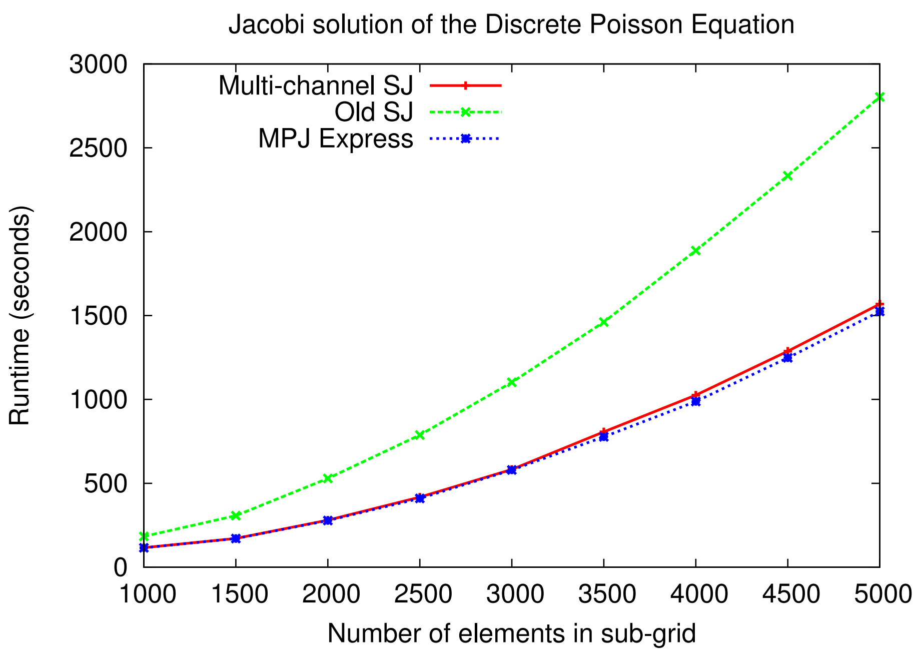


Figure 7.15.: Benchmark Results: 9 nodes Jacobi solution.

correctness conditions on the topology of interleaved sessions commonly used in session-typed parallel algorithm implementations. We have implemented a topology verifier to enforce these conditions statically on a configuration file. Finally, benchmark results in Section 7.5 demonstrate how the new primitives, implemented in the session-typed object-oriented programming language SJ, can deliver the above benefits and perform competitively against a Java-based MPI [BC00].

8 Conclusion and Future Work

This chapter begins by discussing selected related work in the light of the contributions of the thesis, before reviewing the conclusions of the thesis as a whole, and identifying directions for future research.

8.1. Related Work

This section we discuss research related to the contributions of this thesis. We focus on formally-based languages for guaranteeing correctness properties in distributed or parallel programming, and formal analysis on existing parallel programming methodologies.

8.1.1. Formally-founded Communications Programming Languages

Pilot [CGG10] is a parallel programming layer on top of standard MPI, aiming to simplify complex MPI primitives based on CSP. The communication is synchronous and channels are untyped to allow a reuse for different types. The implementation includes an analyser to detect communication deadlock at runtime. Our proposed framework Session C is static and is able to detect and prevent deadlocks before execution.

Occam-pi [Occ] is a system-level efficient concurrent language with channel-based communication based on CSP and the π-calculus. It offers various locking and barrier abstractions, but do not support deadlock analysis.

Heap-Hop [VJ11] is a verification tool for C based on dual contracts and Separation Logic. It can detect a deadlock based on contract specifications, but treats only *binary* (two parties) communications. Our work differs in that we centre on multiparty session-based abstractions for structured communications programming combined with a full formal assurance for communication safety. The authors suggested the possibility of uncovering potential deadlocks without running the program by extracting and checking the underlying CSP model from the program code.

Interprocedural Control Flow Graph (ICFG) [SKH06] and parallel Control Flow Graph (pCFG) [Bro09] are techniques to analyse MPI parallel programs for potential message leak (i.e. communication mismatch) errors. Their approach extends a traditional data-flow analysis by connecting control flow graphs of concurrent processes to their communication edges in order to derive the communication pattern and topology of a parallel program. They take a bottom-up engineering based approach, in contrast to our formally based, top-down global protocol approach, which can give a high-level understanding of the overall communication by design, in addition to the communication safety assurance by Multiparty Session Types.

8.1.2. Parameterised Multiparty Session Types

Pabble's theoretical basis is developed in [DYBH12a] where parameterised MPSTs are formalised using the dependent type theory of Gödel's System $\mathcal{T}$. The main aim in [DYBH12a] is to investigate the decidability and expressiveness of parameterisations of participants. Type checking in [DYBH12a] is undecidable when the indices are not limited to decidable arithmetic subsets or the number of the loop in the parameterised types is infinite. The design of Pabble is inspired by the LTSA tool from a concurrency modelling text book used at our university over two decades [MK06]. The notations for parameterisations from the LTSA tool offer not only practical restrictions to cope with the undecidability of parameterised MPSTs [DYBH12a], but also concise representations for parameterised parallel languages. Our Pabble work is the first to apply parameterised MPST in a practical environment and one foremost aim of our framework with Pabble and parameterised notation is to be developer friendly [HMB^{+}11, Scr] without compromising the strong formal basis of session types.

8.1.3. Formal Analysis of Parallel Applications

Formal verification for message-passing parallel programming has been actively studied in the area of MPI parallel applications. A recent survey [GKS^{+}11] summarises a wide range of model checking-based verification methods for MPI.

Among them, ISP [VVD$^+$08] is a dynamic verifier which applies model-checking techniques to identify potential communication deadlocks in MPI. Their tool uses a fixed test harness and in order to reduce the state space of possible thread interleavings of an execution, the tool exploits an independence between thread actions. Later in [VAG$^+$10], the authors improved its scheduling policy to gain efficiency of the verification.

MSPOE [SGB12] improves on ISP's partial ordering algorithm to overcome the defect and detect orphaning deadlocks. All above tools are test-based and verify correctness with a fixed harness suite. In contrast, our session type-based approach does not depend on external testing, and a valid, compiled program is guaranteed communication-safe and deadlock-free in a matter of seconds.

MUST [HPS$^+$12] is another scalable, MPI dynamic verification tool, which combines two MPI verification tools, Marmot [KBMR03] and Umpire [VdS00], and overcomes scalability challenges in previous tools by comprehensive analysis of the semantics of the primitives.

TASS [SZ11] is a tool that combines symbolic execution [SMAC08] and model checking techniques to verify safety properties of MPI programs. The tool takes a C/MPI application and an input $n \geq 1$ which restricts the input space, then constructs an abstract model with n processes and checks its functional equivalence and deadlocks by executing the model of the application. TASS does not verify properties for an unbounded number of communication participants nor treat parameterisation, whereas we can work with message-passing programs where the number of participants is unknown at compile time, if they are written in well-formed, projectable Pabble.

Above approaches are test-based or model-checking based and may not be able to cover all possible states of the model, whereas the session type-based approach does not depend on external testing or extraction of models from program code to check for safety properties. Most implementations of session types integrate tightly with the underlying programming language, and hence encourage designing communication-correct programs from the start, especially given the high-level communication structure which session types capture.

Our recent collaboration work [HMM$^+$12, MMV$^+$13] aims to use session types for deductive verification of MPI programs. This bottom-up approach focuses on accurate representing MPI primitives and datatypes, and the resulting language `MPITypes` is verified against A Verifier for Concurrent C (VCC), a concurrent C verifier tool to verify the correctness of the MPI application. While the Pabble language introduced in chapter 4 is similarly designed with influences from MPI, it is designed to be an independent high-level abstraction over distributed interactions. As a result, Pabble makes no assumption about the execution environment (e.g. collective loops in MPI and `MPITypes`), and allows Pabble to represent general protocols with separate roles in distributed systems.

8.1.4. MPI code generation

The general approach of describing parallel patterns and reusing them with different computation modules can date back to [DFH+93] by Darlington et al., where parallel patterns are described as higher order *skeleton* functions, written in a functional language. Parallel applications are implemented as functions that combine with the skeletons and transformed. Their system targets specialised parallel machines, and our approach targets MPI, a standard for parallel programming in a range of hardware configurations. The approach, also known as *algorithmic skeleton frameworks* for parallel programming, is surveyed in [GVL10]. Some of these tools also target MPI for high-level structured parallel programming, and only works with a limited set of parallel patterns. Our code generation workflow based on Pabble in Chapter 6 supports generic patterns written in Pabble and guarantees communication safety in the generated MPI code.

8.1.5. Communication Optimisations in MPI

Techniques for improving performance of MPI include building libraries for efficient transmission of data, e.g. [DPS+09] or MPI-aware optimising compilers, e.g. [FL11], which convert all MPI communication into one-sided communication for performance. Most computation and communication overlap to reduce the negative impact of the communication overhead. Our asynchronous message optimisation presented in Chapter 3 is one such instance to facilitate communication-computation overlap. Unlike Session C, existing works do not offer a similar framework, where a type-theoretic basis gives a formal safety assurance for optimised code.

8.1.6. Dependent Typing Systems

Liquid Type [RKJK08] is a dependent typing system to automatically infer memory safety properties from program source code without using verbose annotations. The work [RKJ10] introduced an analyser for the C language in the low-level imperative environment based on Liquid Types and refinement types. The recent work on Liquid Types [KRBJ12] applied the tool with SMT solvers to assist parallelisation of code regions by determining statically whether parallel threads will run on disjoint shared memory without races. Our work applies dependent session types to guarantee different kinds of safety, communication safety and deadlock freedom, in explicit message-passing based distributed and parallel programming rather than shared memory concurrency. It is an interesting future topic to integrate with model-checking tools to handle projectability with more complex indices in addition to functional correctness of session programs.

8.2. Conclusion

In the beginning of this thesis we discussed why interaction-based parallel programming is difficult to get right, and that with a formal typing discipline for concurrency, we will be able to not just model, but also ensure the lack of communication-related bad behaviours in the model. We asked the questions, *is it feasible to apply and adapt Session Types for parallel programming?*, and *how can we apply our methodology practically to scalable parallel applications?* The contributions of the thesis show that answers to both questions are affirmative. Below we present the summary and conclude our findings.

8.2.1. Session C

The Session C programming framework is our answer to the first question. With Session C, we built a programming workflow following the Multiparty Session Types theory [HYC08], where a *global type* describes the overall interaction of an application, and through a *endpoint projection* algorithm, we get *local types*, which are specifications for implementations. Conformance of the specifications are ensured by a static type checking process against a set of runtime communication primitives provided by Session C. The type checking process also considers asynchronous message passing in optimised implementations, through the support of asynchronous message passing theory [Mos09, MY09], commonly found in practices of parallel programming.

8.2.2. Pabble

Our most significant results come from the Pabble protocol description language [NY14b]. In Chapter 4 we presented the Pabble language which is parametric and designed especially for modelling interactions between distributed parallel processes that can scale. Despite the undecidability in its underlying dependent typing theory [DYBH12b], Pabble overcomes it by using only integer indices which also adds practicality to the language. We developed theory and tool to ensure that Pabble protocols are safe even when they scale up to an unbounded number of processes.

This leads us to another of our contributions in Chapter 6, a framework for parallel code generation. Pabble protocol generates an MPI application without computation code, and the application is combined with *computation kernels* developed by programmers as sequential source code, using an aspect-oriented compilation tool. The end results are parallel applications that are guaranteed lack of communication errors by construction.

8.2.3. Multi-channel Primitives

The multi-channel primitives introduced in Chapter 7 are a departure from the Multiparty Session Types framework that Session C and Pabble are based on. However, it represents a different approach to parallel programming, where the multi-channel primitives connect pairs of interacting processes, and a parallel application is simply a connected set of binary interacting processes. The overall topology, which is similar to a global protocol specification, is verified separately by a topology verifier, but the binary sessions are guaranteed to be correct by the Session Java language which we use to implement the multi-channel primitives.

8.2.4. Challenges

All three approaches have their advantages and disadvantages. Session C is a framework that can strongly guarantee safety through static checking, but is limited by its small set of runtime primitives that a user can use; Static type checking also means that with a dependent specification, e.g. Pabble, checking will likely require model checking tools to explore all possibility expansion of expressions when checking for equivalence against protocol.

Pabble code generation is simple and powerful, since it uses the standard MPI as target runtime, but the users do not need to program the complex MPI directly. While the framework is designed with performance in mind, experienced users of MPI might find our framework too conservative as we disallow most potentially unsafe operations because of our safety guarantees.

The biggest advantage of multi-channel primitives is that it does not only allow parallel programming with binary sessions, but that the primitives are useful also for expressing interleaving (multiparty) sessions. It provides a way to express synchronisation of multiple sessions in the same processes. However, the current multi-channel primitive approach may seem cumbersome when an alternative multiparty approach, such as Session C, is possible and feasible.

8.3. Future work

There are a lot of interesting future directions in this line of research, and below we list the ones that are most relevant to this thesis that can be followed up immediately.

8.3.1. Session C and Type Checking

Type checking with Session C is limited by the small set of runtime primitives that Session C provides, and that the current runtime primitives does not scale as well as MPI (Chapter 5). In order to fulfil the full potential of Session C as a session-based programming framework, we plan to extend its type checking support with Pabble for scalable applications, and adding primitives to support multirole Multiparty Session Types [DY11], which allows dynamic adding processes to and removing processes from a running session. Both of these extensions require extra verification to complement static checking: for

Pabble support, undecidability in type checking will require techniques such as symbolic execution to expand expressions in the source code if they are needed for Pabble role index calculation; for multirole session types support, dynamic checks will be employed to ensure the application remain safe after adding or removing a process from the session.

Moreover, the runtime primitives will have to be updated to match the expressiveness of Pabble and multirole session types. Since MPI is a standardised interface, and it has sufficient features to support, it would be one of the leading candidates to replace the current runtime primitives.

The development workflow will be simplified if global protocols can be synthesised as described in a recent work [DY13], so application developers can ensure correctness of existing code by global protocol synthesis.

8.3.2. Pabble and Code Generation

A number of enhancements are planned for Pabble including support for annotations which can complement the protocol description to specify assertions. The type checking process can use the extra constraints or conditions provided to combine with model checkers to also assure functional correctness of the overall application. Annotations will also enable integration with runtime monitoring described in [HNY+13] for a combined static and dynamic approach to communication correct application using Pabble.

Further extensions of Pabble may include adding theory and tooling support for modelling process creation and destroy in the protocol level with the multirole work described for runtime and checking changes.

We also plan to introduce the existential operator to capture nondeterministic message passing (such as MPI receive-any) operations. For example, to model dynamic load balancing between multiple clients:

$$\mu x.\exists w : \text{worker}_{1..N}\{c \to \text{master}\langle\text{request}\rangle; \text{master} \to w\langle\text{reply}\rangle\}; x$$

This describes the pattern where a number of workers, in unspecified order, exchange messages with the master coordinator process. It is difficult to perform static session type checking on parallel programs with such a nondeterministic interaction pattern not known in the protocol. Our early unpublished results show that such structured pattern can be modelled safely under certain conditions.

Error recovery is also a topic of interest, as large scale high performance parallel applications often need to gracefully handle unexpected errors such as hardware failures. Type-based approach to error handling and recovery will be explored as part of ongoing research on Scribble (the language which Pabble extends from).

Finally, our long term goal for Pabble is to extend the language for it to become an ubiquitous language for describing protocols used in parallel programming. We envisage it to be used for both type checking (e.g. Session C) and code generation approaches to ensuring communication correctness. Developing ways of integrating Pabble in different parallel programming environment, such as FPGA heterogeneous computing, will be a step towards our goal. For example, our kernel-based code generation framework with Pabble shares a lot of common features with the data-flow programming model, which rising in popularity for designing reconfigurable hardware/FPGAs.

8.3.3. Multi-channel Primitives

In the conclusion we hinted that the multi-channel primitives can be adapted as a general synchronisation mechanism for interleaving multiparty sessions. It is a future topic to explore how they can be implemented in other session programming approaches and what impact they can bring about. This thesis showed that multi-channel primitives can help breaking down development of larger parallel applications to modules of smaller binary sessions. There is potential that the primitives can simplify application development of large scale applications, which individual components can be developed separately with independent sessions. The components and session can then be connected when needed, using the multi-channel primitives.

Bibliography

[AH05] David Aspinall and Martin Hofmann. Dependent Types. In *Advanced Topics in Types and Programming Languages.* MIT Press, 2005.

[AWW$^+$09] Krste Asanovic, John Wawrzynek, David Wessel, Katherine Yelick, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, and Koushik Sen. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56, October 2009.

[BC00] Mark Baker and Bryan Carpenter. MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing. In *Parallel and distributed processing*, pages 552–559, 2000.

[BCD$^+$08] Lorenzo Bettini, Mario Coppo, Loris DAntoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Others, and Bettini et al. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR 2008*, volume 5201 of *LNCS*, pages 418–433, Berlin, Heidelberg, 2008. Springer.

[BDHT] Pavan Balaji, Jim Dinan, Torsten Hoefler, and Rajeev Thakur. Advanced MPI Programming (Tutorial at SC'13). `http://www.mcs.anl.gov/~thakur/sc13-mpi-tutorial/`.

[BHY10] Andi Bejleri, Raymond Hu, and Nobuko Yoshida. Session-Based Programming for Parallel Algorithms: Expressiveness and Performance. *Electronic Proceedings in Theoretical Computer Science*, 17:17–29, February 2010.

[Bro09] Greg Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *2009 International Symposium on Code Generation and Optimization*, pages 1–12. IEEE, March 2009.

[CCC+12] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. LARA: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development - AOSD '12*, page 179, New York, New York, USA, 2012. ACM Press.

[CDCP12] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On Global Types and Multi-Party Session. *Logical Methods in Computer Science*, 8(1), March 2012.

[CDL] Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.

[CGG10] J. Carter, W. B. Gardner, and G. Grewal. The pilot approach to cluster programming in C. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, April 2010.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*, page 519, New York, New York, USA, 2005. ACM Press.

[Cha] Chapel homepage. `http://chapel.cray.com/`.

[CLR08] Henri Casanova, Arnaud Legrand, and Yves Robert. *Parallel Algorithms*. Chapman & Hall, July 2008.

[DCdLY08] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In *TGC '07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.

[DFH$^+$93] J Darlington, A Field, P Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE'93*, pages 146–160, 1993.

[DKdS$^+$05] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications - SE-HPCS '05*, pages 78–82, New York, New York, USA, 2005. ACM Press.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5, 1998.

[DPS$^+$09] Anthony Danalis, Lori Pollock, Martin Swany, John Cavazos, and Others. MPI-aware compiler optimizations for improving communication-computation overlap. In *ICS '09*, pages 316–325, New York, New York, USA, 2009. ACM Press.

[DY10] Pierre-Malo Deniélou and Nobuko Yoshida. Buffered Communication Analysis in Distributed Multiparty Sessions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '11*, page 435, New York, New York, USA, January 2011. ACM Press.

[DY12] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP*, LNCS, pages 194–213, Berlin, Heidelberg, 2012. Springer-Verlag.

[DY13] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.

[DYBH12a] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.

[DYBH12b] Pierre-Malo Denielou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised Multiparty Session Types. *Logical Methods in Computer Science*, 8(4):1–46, October 2012.

[FL11] Andrew Friedley and Andrew Lumsdaine. Communication Optimization Beyond MPI. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 2018–2021. IEEE, May 2011.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Notices*, volume 33, pages 212–223, 1998.

[For] Fortress homepage. `http://projectfortress.java.net/`.

[FWH13] C. Ferner, B. Wilkinson, and B. Heath. Toward using higher-level abstractions to teach Parallel Computing. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1291–1296. IEEE, 2013.

[Gay08] Simon Gay. Bounded Polymorphism in Session Types. *MSCS*, pages 895–930, 2008.

[GCBH10] Thore Graepel, Joaquin Quinonero Candela, Thomas Borchert, and Ralf Herbrich. Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsofts Bing Search Engine. In *Proceedings of the 27th International Conference on Machine Learning ICML 2010*, June 2010.

[GFB$^+$04] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H Castain, David J Daniel, Richard L Graham, and Timothy S Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *In Proceedings,* *11th European PVM/MPI Users Group Meeting*, pages 97–104, 2004.

[GH05] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[GKKG03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 edition, January 2003.

[GKS+11] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. Formal analysis of MPI-based parallel programs. *Communications of the ACM*, 54(12):82, December 2011.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[Gro02] William Gropp. MPICH2: A new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 2474, 2002.

[GSB+06] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2006.

[GVL10] Horacio González-Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, 2010.

[GVR+10] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on*

Principles of programming languages - POPL '10, page 299, New York, New York, USA, 2010. ACM Press.

[HKP$^+$10] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-Safe Eventful Sessions in Java. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 329–353, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[HMB$^+$11] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling Interactions with a Formal Foundation. In Raja Natarajan and Adegboyega Ojo, editors, *ICDCIT 2011*, volume 6536 of *LNCS*, pages 55–75, Berlin, Heidelberg, 2011. Springer.

[HMM$^+$12] Kohei Honda, Eduardo R B Marques, Francisco Martins, Nicholas Ng, Vasco T Vasconcelos, and Nobuko Yoshida. Verifications of MPIPrograms using Session Types. In *EuroMPI'12*, LNCS, pages 291–293, Berlin, Heidelberg, 2012. LNCS.

[HNY$^+$13] Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations: Distributed dynamic verication with session types and Python. In *RV*, volume 8174 of *LNCS*, pages 130–148, 2013.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HPS$^+$12] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *SC '12*, pages 1–11. IEEE, 2012.

[HT91] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *ECOOP 1991*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.

[Hu10] Raymond Hu. *Structured, Safe and High-level Communications Programming with Session Types.* PhD thesis, Imperial College London, 2010.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98 Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*, volume 5201 of *LNCS*, page 273, New York, New York, USA, 2008. ACM Press.

[HYH08] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP '08 Proceedings of the 22nd European conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 516–541, Berlin, Heidelberg, 2008. Springer.

[Imp] Imperial College High Performance Computing service. `http://www.imperial.ac.uk/ict/services/teachingandresearchservices/highperformancecomputing`.

[Jac] Jacobi and Gauss-Seidel Iteration. `http://math.fullerton.edu/mathews/n2003/GaussSeidelMod.html`.

[KBMR03] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: an MPI analysis and checking tool. In *PARCO 2003*, pages 493–500, 2003.

[KRBJ12] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic parallelism via liquid effects. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, page 45, New York, New York, USA, August 2012. ACM Press.

[LA04] Chris Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International*

Symposium on Code Generation and Optimization, 2004. CGO 2004., CGO '04, pages 75–86, Washington, DC, USA, 2004. IEEE.

[Lei91] Frank Thomson Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes.* Morgan Kaufmann, 1991.

[LP10] Jonathan K. Lee and Jens Palsberg. Featherweight X10. In R Govindarajan, David A Padua, and Mary W Hall, editors, *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, page 25, New York, New York, USA, 2010. ACM Press.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.).* Wiley, 2006.

[MMP10] Emilio P. Mancini, Gregory Marsh, and Dhabaleswar K. Panda. An MPI-stream hybrid programming model for computational clusters. In *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pages 323–330, 2010.

[MMV$^+$13] Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, Nicholas Ng, and Nuno Martins. Towards deductive verification of MPI programs against session types. *Electronic Proceedings in Theoretical Computer Science*, 137:103–113, December 2013.

[Mos09] Dimitris Mostrous. *Session Types in Concurrent Calculi: Higher-Order Processes and Objects.* PhD thesis, Imperial College London, 2009.

[MPI] Message-Passing Interface. `http://www.mcs.anl.gov/research/projects/mpi/`.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Process, Part I. *Journal of Information andComputation*, pages 1–77, 1992.

[MVF08] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip Terascale Processor. In *2008 SC*, pages 1–11. IEEE, November 2008.

[MY09] Dimitris Mostrous and Nobuko Yoshida. Session-Based Communication Optimisation for Higher-Order Mobile Processes. In Pierre-Louis Curien, editor, *TLCA 2009*, volume 5608 of *LNCS*, pages 203–218, Berlin, Heidelberg, June 2009. Springer Berlin Heidelberg.

[MYH09] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP 2009*, volume 5502 of *LNCS*, pages 316–332. Springer, 2009.

[N-b] N-body algorithm using pipeline. `http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/advmsg/nbodypipe_c.htm`.

[NCY15] Nicholas Ng, José G.F. Coutinho, and Nobuko Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC*, LNCS. Springer, 2015.

[Ng10] Nicholas Ng. *High Performance Parallel Design based on Session Programming.* MEng thesis, Imperial College London, 2010.

[NY14a] Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised Scribble. *SOCA*, pages 1–16, 2014.

[NY14b] Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised Scribble for Parallel Programming. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014.

[NYH12] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In Carlo A. Furia and Sebastian Nanz, editors, *TOOLS'12 Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 202–218, Berlin, Heidelberg, 2012. Springer.

[NYL13] Nicholas Ng, Nobuko Yoshida, and Wayne Luk. Scalable Session Programming for Heterogeneous High-Performance Systems. In *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD and OpenCert*, volume 8368 of *LNCS*, pages 82–98. Springer, 2013.

[NYN$^+$12] Nicholas Ng, Nobuko Yoshida, Xin Yu Niu, Kuen Hung Tsoi, and Wayne Luk. Session types: towards safe and fast reconfigurable programming. *ACM SIGARCH Computer Architecture News*, 40(5):22, March 2012.

[NYP$^+$11] Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryftis. Safe Parallel Programming with Session Java. In Wolfgang Meuter and Gruia-Catalin Roman, editors, *COORDINATION*, volume 6721 of *LNCS*, pages 110–126, Berlin, Heidelberg, 2011. Springer.

[Occ] Occam-pi homepage. `http://www.occam-pi.org/`.

[OOI] Use Cases from OOI Project. `https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI`.

[Pab] Pabble project page.

[Pad13] Luca Padovani. Fair Subtyping for Open Session Types. In *Proceedings of 40th International Colloquium on Automata, Languages, and Programming (ICALP'13), Part II*, LNCS 7966, pages 373–384. Springer, 2013.

[Par] METIS and ParMETIS. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[Ran98] K.H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, MPI, 1998.

[RKJ10] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '10*, pages 131–144, New York, New York, USA, January 2010. ACM Press.

[RKJK08] Patrick M. Rondon, Ming Kawaguchi, Ranjit Jhala, and Ming Kawaguci. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, pages 159–169, New York, New York, USA, June 2008. ACM Press.

[Scr] Scribble homepage. `http://scribble.org/`.

[Ses] Session C homepage. `http://www.doc.ic.ac.uk/~cn06/sessionc/`.

[SGB12] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. A sound reduction of persistent-sets for deadlock detection in mpi applications. In *SBMF 2012*, volume 7498 of *LNCS*, pages 194–209. Springer, 2012.

[SKH06] M.M. Strout, B. Kreaseck, and P.D. Hovland. Data-Flow Analysis for MPI Programs. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 175–184. IEEE, August 2006.

[SMAC08] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM ToSEM*, 17(2):1–34, April 2008.

[SnPM$^+$10] Manuel Saldaña, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, Ralph Wittig, Henry Styles, and

Andrew Putnam. MPI as a Programming Model for High-Performance Reconfigurable Computers. *ACM Transactions on Reconfigurable Technology and Systems*, 3(4):1–29, November 2010.

[SNZE10] K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. Efficient Session Type Guided Distributed Interaction. In Dave Clarke and Gul Agha, editors, *Coordination Models and Languages*, volume 6116 of *LNCS*, pages 152–167, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[SPGV07] Jesper H Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: High-Throughput Stream Programming in Java. In *OOPSLA '07*, pages 211–228. ACM, 2007.

[SZ11] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming - PPoPP '11*, pages 309–310, New York, New York, USA, February 2011. ACM Press.

[The93] The MPI Forum. MPI : A Message Passing Interface. In *Proceedings of the Conference on High Performance Networking and Computing*, pages 878–883, 1993.

[VAG$^+$10] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, Greg Bronevetsky, and Others. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, November 2010.

[VdS00] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *SC '00*, page 51. IEEE, 2000.

[VJ11] Jules Villard and Villard Jules. *Heaps and Hops*. PhD thesis, ENS Cachan, 2011.

[VVD$^+$08] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '09*, page 261, New York, New York, USA, 2008. ACM Press.

[WSC] Web Services Choreography Requirements. `http://www.w3.org/TR/ws-chor-reqs/`.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACCFirst Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

[WVF13] B. Wilkinson, J Villalobos, and C. Ferner. Pattern programming approach for teaching parallel and distributed computing. In *The 44th ACM Technical Symposium on Computer Science Education (SIGCSE2013)*, pages 409–414. ACM, 2013.

[X10] X10 homepage. `http://x10-lang.org`.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*, pages 249–257, New York, New York, USA, May 1998. ACM Press.

[YV07] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, July 2007.

[ZMQ] ZeroMQ homepage. `http://www.zeromq.org`.

A Appendix

A.1. Appendix for Chapter 7

A.1.1. Multi-channel Session Typing Rules

This section lists the omitted full typing rules from Section 7.4, listed in Figure A.1. In this context, $\mathsf{fn}(Q)$ denotes a set of free shared and session channels, and $\mathsf{fpv}(D)$ stands for a set of free process variables. In the typing system, Δ is complete means that Δ includes only end or $\bot$. Further explanations can be found in [Ng10].

A.1.2. Well-Formed Ring and Mesh Topologies

We define well-formed ring (Definition A.1) and mesh topologies (Definition A.2). We can check that they conform to the general definition of well-formed topology (Definition 7.1). Figure A.2 shows the rank of each process for each topology, indicating how both rings and meshes map to the general definition.

Definition A.1 (Well-formed Ring Topology) *A process group*

$$P_1 \mid P_i \mid P_N$$

$$\frac{}{\Gamma \vdash 1 \triangleright \mathsf{nat}} \qquad \frac{}{\Gamma \vdash \mathsf{true}, \mathsf{false} \triangleright \mathsf{bool}} \qquad \frac{\Gamma \vdash e_i \triangleright \mathsf{nat}}{\Gamma \vdash e_1 + e_2 \triangleright \mathsf{nat}} \qquad \text{[\textsc{Nat}],[\textsc{Bool}],[\textsc{Sum}]}$$

$$\frac{}{\Gamma \cdot a \colon S \vdash a \triangleright S} \qquad \frac{\Gamma;\ \Delta \vdash e \triangleright S}{\Gamma;\ \Delta, \Delta' \vdash e \triangleright S} \qquad \text{[\textsc{Name}],[\textsc{Eval}]}$$

$$\frac{\Delta = k_1 \colon ?[\tau_1]^* \mathsf{end}, \ldots, k_n \colon ?[\tau_n]^* \mathsf{end}}{\Gamma; \Delta \vdash \langle k_1 \ldots k_n \rangle.\mathsf{inwhile} \triangleright \mathsf{bool}} \qquad \text{[\textsc{EInwhile}]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k \colon \epsilon.\mathsf{end}}{\Gamma \vdash P \triangleright \perp} \qquad \frac{\Delta \text{ complete}}{\Gamma \vdash \mathsf{0} \triangleright \Delta} \qquad \text{[\textsc{Bot}],[\textsc{Inact}]}$$

$$\frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot k \colon \overline{\alpha}}{\Gamma \vdash \overline{a}(k).P \triangleright \Delta} \qquad \text{[\textsc{Req}]}$$

$$\frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot k \colon \alpha}{\Gamma \vdash a(k).P \triangleright \Delta} \qquad \text{[\textsc{Acc}]}$$

$$\frac{\Gamma \vdash e \triangleright S}{\Gamma \vdash \overline{k}\langle e \rangle \triangleright \Delta \cdot k \colon ![S]; \mathsf{end}} \qquad \frac{\Gamma \cdot x \colon S \vdash P \triangleright \Delta \cdot k \colon \alpha}{\Gamma \vdash k(x).P \triangleright \Delta \cdot k \colon ?[S]; \alpha} \qquad \text{[\textsc{Send}],[\textsc{Rcv}]}$$

$$\frac{\Gamma \vdash P_1 \triangleright \Delta \cdot k \colon \tau_1.\mathsf{end} \quad \cdots \quad \Gamma \vdash P_n \triangleright \Delta \cdot k \colon \tau_n.\mathsf{end}}{\Gamma \vdash k \ \triangleright\ \{l_i \colon\ P_i\}_{i \in \{1..n\}} \triangleright \Delta \cdot k \colon\ \&\, \{l_i \,\colon\, \tau_i\}_{i \in \{1..n\}}.\mathsf{end}} \qquad \text{[\textsc{Br}]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k \colon \tau_j.\mathsf{end} \quad 1 \leq j \leq n}{\Gamma \vdash k \ \triangleleft\ l; P \triangleright \Delta \cdot k \colon\ \oplus\, \{l_i \,\colon\, \tau_i\}_{i \in \{1..n\}}.\mathsf{end}} \qquad \text{[\textsc{Sel}]}$$

$$\frac{}{\Gamma \vdash \overline{k}\langle k' \rangle; \triangleright \Delta \cdot k \colon ![\alpha]; \mathsf{end} \cdot k' \colon \alpha} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k \colon \beta \cdot k' \colon \alpha}{\Gamma \vdash k(k').P \triangleright \Delta \cdot k \colon ?[\alpha]; \beta} \qquad \text{[\textsc{Thr}],[\textsc{Cat}]}$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q \triangleright \Delta} \qquad \text{[\textsc{If}]}$$

$$\frac{\Gamma\Delta \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \cdot k_1 \colon \tau_1.\mathsf{end} \cdots k_n \colon \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots k_n \rangle.\mathsf{outwhile}(e)\{P\} \triangleright \Delta \cdot k_1 \colon ![\tau_i.\mathsf{end}]^*} \qquad \text{[\textsc{Outwhile}]}$$

$$\frac{\Gamma \vdash Q \triangleright \Delta \cdot k_1 \colon \tau_1.\mathsf{end} \cdots k_n \colon \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots k_n \rangle.\mathsf{inwhile}\{Q\} \triangleright \Delta \cdot k_1 \colon ?[\tau_i.\mathsf{end}]^*} \qquad \text{[\textsc{Inwhile}]}$$

$$\frac{\Gamma \vdash b_i \triangleright \mathsf{bool}}{\Gamma \vdash \Pi_{i \in \{1..n\}} k_i \dagger [b_i] \triangleright k_1 \colon \dagger, \ldots, k_n \colon \dagger} \qquad \text{[\textsc{Message}]}$$

$$\frac{\Gamma \cdot a \colon S \vdash P \triangleright \Delta}{\Gamma \vdash (\boldsymbol{\nu}\, a)(P) \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k \colon\ \perp}{\Gamma \vdash (\boldsymbol{\nu}\, k)(P) \triangleright \Delta} \qquad \text{[\textsc{NRes}],[\textsc{CRes}]}$$

$$\frac{\Gamma; \emptyset \vdash e \triangleright S}{\Gamma \cdot X \colon S\alpha \vdash X[ek] \triangleright \Delta \cdot k \colon \alpha} \qquad \text{[\textsc{Var}]}$$

$$\frac{\Gamma \cdot X \colon S\alpha \cdot x \colon S \vdash P \triangleright k \colon \alpha \qquad \Gamma \cdot X \colon S\tau \vdash Q \triangleright \Delta}{\Gamma \vdash \{X_i(x_i k_i) \ = \ P_i\}_{i \in \{1..n\}} \ \mathsf{in}\ Q \triangleright \Delta} \qquad \text{[\textsc{Def}]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P; Q \triangleright \Delta; \Delta'} \qquad \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \qquad \text{[\textsc{Seq}],[\textsc{Conc}]}$$

Figure A.1.: Multi-channel Session Types: Typing rules.

conforms to a well-formed ring topology *if:*

$$\begin{aligned}
P_1 &= \langle k_{1,2}, k_{1,n}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{12}, k_{1n}]\}\\
P_i &= \langle k_{i,i+1}\rangle.\mathsf{outwhile}(\langle k_{i-1,i}\rangle.\mathsf{inwhile})\{Q_i[k_{i,i+1}, k_{i-1,i}]\}\\
P_N &= \langle k_{1,n}, k_{n-1,n}\rangle.\mathsf{inwhile}\{Q_N[k_{1,n}, k_{n-1,n}]\}
\end{aligned}$$

where $1 \leq i \leq n$

$$\begin{aligned}
&\Gamma \vdash P_1 \triangleright \{k_{1,2} \colon T_{1,2}, k_{1,n} \colon T_{1,n}\}\\
&\Gamma \vdash P_i \triangleright \{k_{i,i+1} \colon T_{i,i+1}, k_{i-1,i} \colon T'_{i-1,i}\}\\
&\Gamma \vdash P_N \triangleright \{k_{1,n} \colon T'_{1,n}, k_{n-1,n} \colon T'_{n-1,n}\}
\end{aligned}$$

with $\overline{T_{i,j}} = T'_{i,j}$

Definition A.2 (Well-formed Mesh Topology) *A process group*

$$\begin{aligned}
&P_{NW} \mid P_{NE} \mid P_{SW} \mid P_{SE} \mid P_{N_1} \ldots \mid P_{N_m} \mid P_{S_1} \ldots \mid P_{S_m}\\
&\quad \mid P_{E_1} \ldots \mid P_{E_n} \mid P_{W_1} \ldots \mid P_{W_n} \mid P_{C_2 2} \ldots \mid P_{C_{n-1 m-1}}
\end{aligned}$$

conforms to a well-formed mesh topology *if:*

$$
\begin{aligned}
P_{NW} &= \langle t_1, l_1\rangle.\mathsf{outwhile}(e)\{Q_{NW}[t_1, l_1]\}\\
P_{N_j} &= \langle t_{j+1}, vc_{1j}\rangle.\mathsf{outwhile}(\langle t_j\rangle.\mathsf{inwhile})\{Q_{N_j}[t_{j+1}, vc_{1j}, t_j]\}\\
P_{NE} &= \langle r_1\rangle.\mathsf{outwhile}(\langle t_m\rangle.\mathsf{inwhile})\{Q_{NE}[r_1, t_m]\}\\
P_{W_i} &= \langle hc_{i1}, l_{i+1}\rangle.\mathsf{outwhile}(\langle l_i\rangle.\mathsf{inwhile})\{Q_W[hc_{i1}, l_{i+1}, l_i]\}\\
P_{C_{ij}} &= \langle vc_{i+1\ j}, hc_{i\ j+1}\rangle.\mathsf{outwhile}(\langle hc_{ij}, vc_{ij}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_{C_{ij}}[vc_{i+1\ j}, hc_{i\ j+1}, hc_{ij}, vc_{ij}]\}\\
P_{E_i} &= \langle r_{i+1}\rangle.\mathsf{outwhile}(\langle hc_{im}, r_i\rangle.\mathsf{inwhile})\{\\
&\qquad Q_{E_i}[r_{i+1}, hc_{im}, r_i]\}\\
P_{SW} &= \langle b_1\rangle.\mathsf{outwhile}(\langle l_n\rangle.\mathsf{inwhile})\{Q_{SW}[b_1, l_n]\}\\
P_{S_j} &= \langle b_{j+1}\rangle.\mathsf{outwhile}(\langle b_j, vc_{nj}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_{S_j}[b_{j+1}, b_j, vc_{nj}]\}\\
P_{SE} &= \langle b_m, r_n\rangle.\mathsf{inwhile}\{Q_{SE}[b_m, r_n]\}
\end{aligned}
$$

where $1 \leq i \leq n, 1 \leq j \leq m$

$$
\begin{aligned}
&\Gamma \vdash Q_{NW} \triangleright \{t_1 : T_{t_1}, l_1 : T_{l_1}\}\\
&\Gamma \vdash Q_{N_j} \triangleright \{t_{j+1} : T_{t_{j+1}}, vc_{1j} : T_{vc_{1j}}, t_j : T'_{t_j}\}\\
&\Gamma \vdash Q_{NE} \triangleright \{r_1 : T_{r_1}, t_m : T'_{t_m}\}\\
&\Gamma \vdash Q_W \triangleright \{hc_{i1} : T_{hc_{i1}}, l_{i+1} : T_{l_{i+1}}, l_i : T'_{l_i}\}\\
&\Gamma \vdash Q_{C_{ij}} \triangleright \{vc_{i+1\ j} : T_{vc_{i+1\ j}}, hc_{i\ j+1} : T_{hc_{i\ j+1}}, hc_{ij} : T'_{hc_{ij}}, vc_{ij} : T'_{vc_{ij}}\}\\
&\Gamma \vdash Q_{E_i} \triangleright \{r_{i+1} : T_{r_{i+1}}, hc_{im} : T_{hc_{im}}, r_i : T'_{r_i}\}\\
&\Gamma \vdash Q_{SW} \triangleright \{b_1 : T_{b_1}, l_n : T'_{l_n}\}\\
&\Gamma \vdash Q_{S_j} \triangleright \{b_{j+1} : T_{b_{j+1}}, b_j : T'_{b_j}, vc_{nj} : T'_{vc_{nj}}\}\\
&\Gamma \vdash Q_{SE} \triangleright \{b_m : T'_{b_m}, r_n : T'_{r_n}\}
\end{aligned}
$$

with $\overline{T_i} = T'_i$

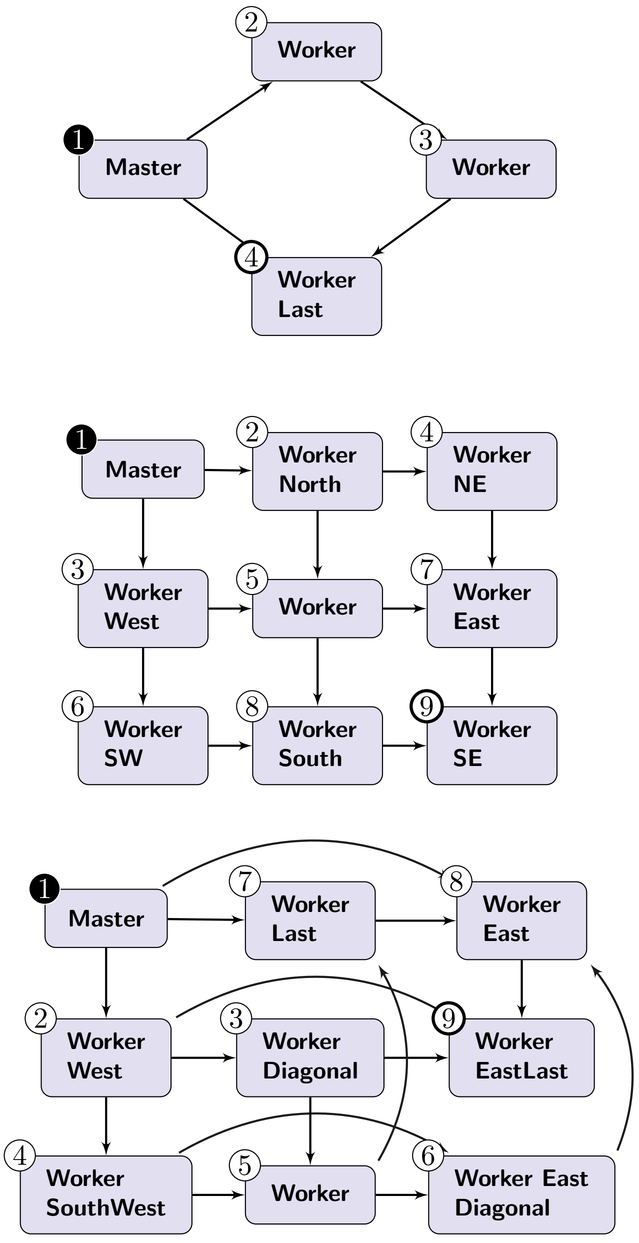


Figure A.2.: Ring, mesh, and wraparound mesh topologies, with rank annotations.

A.1.3. Proofs for Multi-channel Session Calculus Subject Reduction, and Deadlock Freedom

This subsection presents the proofs for the theorems associated to the Multi-channel session types omitted from Chapter 7.

We first present a definition for sequential composition (Definition A.3), parallel composition (Definition A.4), both involving runtime session types and completed session types. Then we present the Subject Congruence theorem (Theorem A.1) and the Subject Reduction theorem (Theorem 7.1), and finally Deadlock Freedom theorem (Theorem 7.3).

Sequential Composition

Definition A.3 *Sequential composition of session type are defined as [DCdLY08]:*

$$\tau;\ \alpha = \begin{cases} \tau.\alpha & \textit{if } \tau \textit{ is a partial session type and } \alpha \textit{ is a completed session type} \\ \bot & \textit{otherwise} \end{cases}$$

$$\begin{aligned} \Delta;\ \Delta' = \Delta \setminus \mathsf{dom}(\Delta') \cup & \\ & \Delta' \setminus \mathsf{dom}(\Delta) \cup \\ & \{k\colon \Delta(k) \setminus \mathsf{end};\ \Delta'(k) \mid k \in \mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta')\} \end{aligned}$$

The first rule concatenates a partial session type τ with a completed session type α to form a new (completed) session type. The second rule can be decomposed to three parts:

1. $\Delta \setminus \mathsf{dom}(\Delta')$ extracts session types with sessions unique in Δ
2. $\Delta' \setminus \mathsf{dom}(\Delta)$ extracts session types with sessions unique in Δ'
3. $\{k\colon \Delta(k) \setminus \mathsf{end};\ \Delta'(k) \mid k \in \mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta')\}$ modifies session types with a common session k in Δ and Δ' by removing end type from $\Delta(k)$ and concatenates the modified $\Delta(k)$ (which is now a partial session type) with $\Delta'(k)$ as described in the first rule.

Example A.1 *Suppose* $\Delta = \{k_1\colon \epsilon.\mathsf{end},\ k_2\colon ![\mathsf{nat}]; \mathsf{end}\}$ *and* $\Delta' = \{k_2\colon ?[\mathsf{bool}]; .\mathsf{end},\ k_3\colon ![\mathsf{bool}]; \mathsf{end}\}$. *Since* k_1 *is unique in* Δ *and* k_3 *is unique in* Δ'*, we have*

$$\Delta \backslash \mathsf{dom}(\Delta') = \{k_1\colon \epsilon.\mathsf{end}\} \textit{ and } \Delta' \backslash \mathsf{dom}(\Delta) = \{k_3\colon ![\mathsf{bool}]; \mathsf{end}\}$$

A new session type is constructed by removing end *in* $\Delta(k_2)$*, so the composed* *set of mappings is*

$$\Delta;\ \Delta' = \{k_1\colon \epsilon.\mathsf{end},\ k_2\colon ![\mathsf{nat}]; ?[\mathsf{bool}]; \mathsf{end}\,\mathit{Type},\ k_3\colon ![\mathsf{bool}]; \mathsf{end}\}$$

Parallel Composition

Definition A.4 *Parallel composition of session and runtime type is defined as:*

$$\Delta \circ \Delta' = \Delta \setminus \mathsf{dom}(\Delta') \cup \Delta' \setminus \mathsf{dom}(\Delta) \cup\ \{k\colon \beta \circ \beta' \mid \Delta(k) = \beta \text{ and } \Delta'(k) = \beta'\}$$

$$\text{where } \beta \circ \beta' : \begin{cases} \alpha \circ \dagger = & \alpha^\dagger \\ \alpha \circ \overline{\alpha} = & \bot \\ \alpha \circ \overline{\alpha}^\dagger = & \bot^\dagger \end{cases}$$

The parallel composition relation $\circ$ is commutative as the order of composition do not impact the end result.

Subject Congruence

Theorem A.1 *Subject congruence is defined by*

$$\Gamma \vdash P \triangleright \Delta \text{ and } P \equiv P' \text{ implies } \Gamma \vdash P' \triangleright \Delta$$

Pooof. **Case $P \mid 0 \equiv P$.** We show that if $\Gamma \vdash P \mid 0 \triangleright \Delta$, then $\Gamma \vdash P \triangleright \Delta$. Suppose

$$\Gamma \vdash P \triangleright \Delta_1 \quad \text{and} \quad \Gamma \vdash 0 \triangleright \Delta_2.$$

with $\Delta_1 \circ \Delta_2 = \Delta$. Note that Δ_2 only contains $\epsilon.\mathsf{end}$ or $\bot$, hence we can set: $\Delta_1 = \Delta_1' \circ \{k\colon \epsilon.\mathsf{end}\}$ and $\Delta_2 = \Delta_2' \cdot \{k\colon \epsilon.\mathsf{end}\}$ with $\Delta_1' \circ \Delta_2' = \Delta_1' \cdot \Delta_2'$ and $\Delta = \Delta_1' \cdot \Delta_2' \cdot \{k\colon\ \bot\}$. Then by the [Bot]-rule, we have:

$$\Gamma \vdash P \triangleright \Delta_1' \cdot \{k\colon\ \bot\}$$

Notice that, given the form of Δ above, we know that $\mathsf{dom}(\Delta_2') \cap \mathsf{dom}(\Delta_1') \cdot \{k\colon\ \bot\}) = \emptyset$. Hence by applying Weakening, we have:

$$\Gamma \vdash P \triangleright \Delta_1' \cdot \Delta_2' \cdot \{k\colon\ \bot\}$$

as required.

For the other direction, we set $\Delta = \emptyset$ in [Inact].

Case $P \mid Q \equiv Q \mid P$**.** $\circ$ relation is commutative by the definition of $\circ$ (Definition A.4)

Case $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$**.** To show $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, where

$$\Gamma \vdash P \triangleright \Delta_1 \quad \Gamma \vdash Q \triangleright \Delta_2 \quad \Gamma \vdash R \triangleright \Delta_3$$

We assume $(\Delta_1 \circ \Delta_2) \circ \Delta_3$ is defined

Suppose $k\colon \beta_1 \in \Delta_1$ and $k\colon \beta_2 \in \Delta_2$, then we have

$$\begin{cases} \beta_1 = \alpha & \beta_2 = \dagger \\ \beta_1 = \alpha & \beta_2 = \overline{\alpha} \\ \beta_1 = \alpha & \beta_2 = \overline{\alpha}^{\dagger} \\ \beta_1 = \dagger & \beta_2 = \perp \end{cases}$$

Now suppose $k\colon \beta_3 \in \Delta_3$,

if $\beta_1 = \alpha \quad \beta_2 = \dagger$, then $\beta_3 = \overline{\alpha}$

$$\begin{aligned} (\beta_1 \circ \beta_2) \circ \beta_3 &= (\{k\colon \alpha\} \circ \{k\colon \dagger\}) \circ \{k\colon \overline{\alpha}\} = \{k\colon \ \perp^{\dagger}\} \\ \equiv \beta_1 \circ (\beta_2 \circ \beta_3) &= \{k\colon \alpha\} \circ (\{k\colon \dagger\} \circ \{k\colon \overline{\alpha}\}) = \{k\colon \ \perp^{\dagger}\} \end{aligned}$$

if $\beta_1 = \alpha \quad \beta_2 = \overline{\alpha}$, then $\beta_3 = \dagger$

$$\begin{aligned} (\beta_1 \circ \beta_2) \circ \beta_3 &= (\{k\colon \alpha\} \circ \{k\colon \overline{\alpha}\}) \circ \{k\colon \dagger\} = \{k\colon \ \perp^{\dagger}\} \\ \equiv \beta_1 \circ (\beta_2 \circ \beta_3) &= \{k\colon \alpha\} \circ (\{k\colon \overline{\alpha}\} \circ \{k\colon \dagger\}) = \{k\colon \ \perp^{\dagger}\} \end{aligned}$$

in all other cases, $k \notin \mathsf{dom}(\Delta_3)$ and therefore no parallel composition is possible.

Case $(\boldsymbol{\nu}\, u)(P) \mid Q \equiv (\boldsymbol{\nu}\, u)(P \mid Q)$ if $u \notin \mathsf{fn}(Q)$**.** The case when u is a name is standard. Suppose u is channel k and assume $\Gamma \vdash (\boldsymbol{\nu}\, k)(()P \mid Q) \triangleright \Delta$. We have

$$\frac{\Gamma \vdash P \triangleright \Delta_1' \qquad \Gamma \vdash Q \triangleright \Delta_2'}{\Gamma \vdash P \mid Q \triangleright \Delta' \cdot k\colon \ \perp}$$

with $\Delta' \cdot k\colon \ \perp = \Delta_1' \circ \Delta_2'$ and $\Delta' \prec \Delta$ by [Bot]. First notice that k can be in either Δ_i' or in both. The interesting case is when it occurs in both; from Lemma A.3(1) and the fact that $k \notin \mathsf{fn}(Q)$ we know that $\Delta_1' = \Delta_1 \cdot k\colon \epsilon.\mathsf{end}$

and $\Delta_2' = \Delta_2 \cdot k\colon \epsilon.\mathsf{end}$. Then, by applying the [Bot]-rule to k in P, we have $\Gamma \vdash P \triangleright \Delta_1 \cdot k\colon \perp$, and by applying [CRes] we obtain $\Gamma \vdash (\boldsymbol{\nu}\, k)(P) \triangleright \Delta_1$. On the other hand, by Strengthening, we have $\Gamma \vdash Q \triangleright \Delta_2$. Then, the application of [Conc] yields $\Gamma \vdash (\boldsymbol{\nu}\, k)(P \mid Q) \triangleright \Delta'$. Then by applying the [Bot]-rule, we obtain $\Gamma \vdash (\boldsymbol{\nu}\, k)(P \mid Q) \triangleright \Delta$, as required. The other direction is easy.

Case $(\boldsymbol{\nu}\, u)(\mathsf{0}) \equiv \mathsf{0}$. Standard by Weakening and Strengthening.

Case $\mathsf{def}\ D\ \mathsf{in}\ \mathsf{0} \equiv \mathsf{0}$. Similar to the first case using Weakening and Strengthening.

Case $(\boldsymbol{\nu}\, u)(\mathsf{def}\ D\ \mathsf{in}\ P) \equiv \mathsf{def}\ D\ \mathsf{in}\ (\boldsymbol{\nu}\, u)(P)$ if $u \notin \mathsf{fn}(D)$. Similar to the scope opening case using Weakening and Strengthening.

Case $(\mathsf{def}\ D\ \mathsf{in}\ P) \mid Q \equiv \mathsf{def}\ D\ \mathsf{in}\ (P \mid Q)$ if $\mathsf{fpv}(D) \cap \mathsf{fpv}(Q) = \emptyset$. Similar with the scope opening case using Weakening and Strengthening.

Case $\mathsf{0}; P \equiv P$. We show that if $\Gamma \vdash \mathsf{0}; P \triangleright \Delta$, then $\Gamma \vdash P \triangleright \Delta$. Suppose

$$\Gamma \vdash \mathsf{0} \triangleright \Delta_1 \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta_2.$$

with $\Delta_1; \Delta_2 = \Delta$. Δ_2 only contains $\epsilon.\mathsf{end}$ or $\perp$, by definition of sequential composition (Definition A.3), $\Delta(k) = \Delta_1(k).\Delta_2(k) = \epsilon.\Delta_2(k) = \Delta_2(k)$ as required. □

Subject Reduction

We now present some auxiliary results for subject reduction, the following proofs are modified from [YV07], and adapted to our updated typing system.

Lemma A.1 (Weakening Lemma) *Let $\Gamma \vdash P \triangleright \Delta$.*

1. *If $X \notin \mathsf{dom}(\Gamma)$, then $\Gamma \cdot X\colon S\alpha \vdash P \triangleright \Delta$.*

2. *If $a \notin \mathsf{dom}(\Gamma)$, then $\Gamma \cdot a\colon S \vdash P \triangleright \Delta$.*

3. *If $k \notin \mathsf{dom}(\Delta)$ and $\alpha = \perp$ or $\alpha = \epsilon.\mathsf{end}$, then $\Gamma \vdash P \triangleright \Delta \cdot k\colon \alpha$.*

Pooof. A simple induction on the derivation tree of each sequent. For 3, we note that in [Inact] and [Var], Δ contains only $\epsilon.\mathsf{end}$. □

Lemma A.2 (Strengthening Lemma) *Let $\Gamma \vdash P \triangleright \Delta$.*

1. *If $X \notin \mathsf{fpv}(P)$, then $\Gamma \setminus X \vdash P \triangleright \Delta$.*

2. *If $a \notin \mathsf{fn}(P)$, then $\Gamma \setminus a \vdash P \triangleright \Delta$.*

3. *If $k \notin \mathsf{fn}(P)$, then $\Gamma \vdash P \triangleright \Delta \setminus k$.*

Pooof. Standard. $\square$

Lemma A.3 (Channel Lemma) 1. *If $\Gamma \vdash P \triangleright \Delta \cdot k\colon \alpha$ and $k \notin \mathsf{fn}(P)$, then $\alpha = \perp, \epsilon.\mathsf{end}$.*

2. *If $\Gamma \vdash P \triangleright \Delta$ and $k \in \mathsf{fn}(P)$, then $k \in \mathsf{dom}(\Delta)$*

Pooof. A simple induction on the derivation tree of each sequent.

We omit the standard renaming properties of variables and channels, but present the Substitution Lemma (Lemma A.4) for names. Note that we do not require a Substitution Lemma for channels or process variables, for they are not communicated. $\square$

Lemma A.4 (Substitution Lemma) *If $\Gamma \vdash P \triangleright \Delta \cdot k$ and $\Gamma \vdash c\colon S$, then $\Gamma \vdash P\{c/x\} \triangleright \Delta$*

Pooof. Standard.

We write $\Delta \prec \Delta'$ if we obtain Δ' from Δ by replacing $k_1\colon \epsilon.\mathsf{end}, \ldots, k_n\colon \epsilon.\mathsf{end}$ $(n \geq 0)$ in Δ by $k_1\colon \perp, \ldots, k_n\colon \perp$. If $\Delta \prec \Delta'$, we can obtain Δ' from Δ by applying the [Bot]-rule zero or more times. $\square$

Definition A.5 *A process is under a well-formed intermediate topology if:*

1. *(inwhile and outwhile)*

$$
\begin{aligned}
&P_1 = \langle k_{(1,2)}, \ldots, k_{(1,N)} \rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)}, \ldots, k_{(1,N)}]\}\\
&P_i = \langle k_{(i,i+1)}, \ldots, k_{(i,N)} \rangle.\mathsf{outwhile}(\langle k_{(1,i)}, \ldots, k_{(i-1,i)} \rangle.\mathsf{inwhile}\{\})\{\\
&\qquad Q_i[k_{(i,i+1)}, \ldots, k_{(i,N)}, k_{(1,i)}, \ldots, k_{(i-1,i)}]\}\\
&\qquad \mid k_{(i,i+1)} \dagger [b] \mid k_{(1,i)} \dagger [b] \mid \ldots \mid k_{(i-1,i)} \dagger [b]\\
&\qquad \textit{when } i \in \{2..M-1\},\ b \in \{\mathsf{true}, \mathsf{false}\}\\
&P_j = \langle k_{(1,j)}, \ldots, k_{(j-1,j)} \rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\}\\
&\qquad \mid k_{(1,j)} \dagger [b] \mid \ldots \mid k_{(j-1,j)} \dagger [b]\\
&\qquad \textit{when } j \in \{M..N\} \forall b \in \{\mathsf{true}\} \textit{ or } \forall b \in \{\mathsf{false}\}\\
&\quad \textit{and}\\
&\Gamma \vdash Q_1 \triangleright \{k_{(1,2)} : T_{(1,2)}, \cdots, k_{(1,N)} : T_{(1,N)}\}\\
&\Gamma \vdash Q_i \triangleright \{k_{(i,i+1)} : T_{(i,i+1)}, \cdots, k_{(i,N)} : T_{(i,N)},\\
&\qquad k_{(1,i)} : {T'_{(1,i)}}^{\dagger}, \cdots, k_{(i-1,i)} : T'_{i-1}i^{\dagger}\}\\
&\Gamma \vdash Q_j \triangleright \{k_{(1,j)} : {T'_{(1,j)}}^{\dagger},\ k_{(j-1,j)} : {T'_{(j-1,j)}}^{\dagger}\}\\
&\quad \textit{and}\\
&\Gamma \vdash Q_1 \mid Q_2 \mid \cdots \mid Q_n \triangleright \{\tilde{k} : \tilde{\bot}^{\dagger}\}\\
&\quad \textit{with } T_{(i,j)} = \overline{T'_{(i,j)}}
\end{aligned}
$$

2. *(sequencing) $P_i = Q_{1i}; ...; Q_{mi}$ where $(Q_{j1} \mid Q_{j2} \mid \cdots \mid Q_{jn})$ conforms a well-formed intermediate topology for each $1 \leq j \leq m$.*

3. *(base) (1) session actions in P_i follows the order of the index (e.g. the session actions at $k_{(i,j)}$ happens before $k_{(h,g)}$ if $(i,j) < (h,g)$), then the rest is a base process P'_i; and (2) P_i includes neither shared session channels, inwhile nor outwhile.*

Theorem 7.1 (Subject reduction) The following subject reduction rules hold for a *well-formed topology* (Definition 7.1). $\Gamma \vdash P \triangleright \Delta$ and $P \rightarrow P'$ implies $\Gamma \vdash P' \triangleright \Delta'$ such that

$$
\Delta(k) = \alpha \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^{\dagger} \end{cases} \qquad \Delta(k) = \alpha^{\dagger} \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^{\dagger} \end{cases}
$$

Under a *well-formed intermediate topology (Definition A.5)*
$\Gamma \vdash P \triangleright \Delta$ and $P \ \to^* \ P'$ implies $\Gamma \vdash P' \triangleright \Delta'$ such that

$$\Delta(k) = \alpha \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^\dagger \end{cases} \qquad \Delta(k) = \alpha^\dagger \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^\dagger \end{cases}$$

Pooof. We assume that

$$\Gamma \vdash e \triangleright S \quad \text{and} \quad e \downarrow c \quad \text{implies} \quad \Gamma \vdash c \triangleright S \tag{A.1}$$

and prove the result by induction on the last rule applied. For simplicity, assume all nodes are fully connected.

Case inwhile/outwhile for N processes $(\boldsymbol{\nu}\,\tilde{k})(P_1 \mid \ldots \mid P_N)$**.** Assume well-formed topology (Definition 7.1)
Case $E[e] \to E[\mathsf{true}]$
By [Ow1],

$$\begin{aligned}
&\nu\tilde{k}\ (\langle k_{(1,2)}, \ldots, k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)}, \ldots, k_{(1,N)}]\} \\
&\quad \mid \langle k_{(i,i+1)}, \ldots, k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)}, \ldots, k_{(i-1,i)}\rangle.\mathsf{inwhile})\{ \\
&\qquad Q_i[k_{(i,i+1)}, \ldots, k_{(i,N)}, k_{(1,i)}, \ldots, k_{(i-1,i)}]\} \text{ when } i \in \{2..M-1\} \\
&\quad \mid \langle k_{(1,j)}, \ldots, k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\}) \text{ when } j \in \{M..N\} \\
\to^*\ &\nu\tilde{k}\ (\ Q_1[k_{(1,2)}, \ldots, k_{(1,N)}];\ \langle k_{(1,2)}, \ldots, k_{(1,N)}\rangle.\mathsf{outwhile}(e')\{Q_1[k_{(1,2)}, \ldots, k_{(1,N)}]\} \\
&\quad \mid\ k_{(1,2)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(1,N)} \dagger [\mathsf{true}] \\
&\quad \mid \langle k_{(i,i+1)}, \ldots, k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)}, \ldots, k_{(i-1,i)}\rangle.\mathsf{inwhile})\{ \\
&\qquad Q_i[k_{(i,i+1)}, \ldots, k_{(i,N)}, k_{(1,i)}, \ldots, k_{(i-1,i)}]\} \text{ when } i \in \{2..M-1\} \\
&\quad \mid \langle k_{(1,j)}, \ldots, k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\}) \text{ when } j \in \{M..N\}
\end{aligned}$$

$$\begin{aligned}
&\Gamma \vdash (Q_1; P_1 \mid k_{(1,2)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(1,N)} \dagger [\mathsf{true}] \mid P_{i\in 2..M-1} \mid P_{j\in M..N}) \\
&\quad \triangleright\{k_{(1,2)} : T_{(1,2)}; ![T_{(1,2)}]^* \circ ?[T'_{(1,2)}]^{*\dagger}, \ldots, k_{(1,N)} : T_{(1,N)}; ![T_{(1,N)}]^* \circ ?[T'_{(1,N)}]^{*\dagger}, \\
&\qquad k_{(i,i+1)} : ![T_{(i,i+1)}]^* \circ ?[T'_{(i,i+1)}]^*, \ldots, k_{(i,N)} : ![T_{(i,N)}]^* \circ ?[T'_{(i,N)}]^*\}
\end{aligned}$$

By [IwE1],

$$\begin{aligned}
&\nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\}\\
&\qquad\mid k_{(1,2)}\dagger[\mathsf{true}]\mid\ldots\mid k_{(1,N)}\dagger[\mathsf{true}]\\
&\qquad\mid\langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad\quad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\text{ when }i\in\{2..M-1\}\\
&\qquad\mid\langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\text{ when }j\in\{M..N\}\\
\to^*\ &\nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\}\\
&\qquad\mid k_{(1,3)}\dagger[\mathsf{true}]\mid\ldots\mid k_{(1,N)}\dagger[\mathsf{true}]\\
&\qquad\mid\langle k_{(2,3)},\ldots,k_{(2,N)}\rangle.\mathsf{outwhile}(\ \mathsf{true}\ )\{\ Q_2[k_{(2,3)},\ldots,k_{(2,N)},k_{(1,2)}]\ \}\\
&\qquad\mid\langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad\quad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\text{ when }i\in\{3..M-1\}\\
&\qquad\mid\langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\text{ when }j\in\{M..N\}
\end{aligned}$$

$$\begin{aligned}
&\Gamma\vdash(Q_1;P_1\mid P_{i\in 2..M-1}\mid P_{j\in M..N}\mid k_{(1,3)}\dagger[\mathsf{true}]\mid\ldots\mid k_{(1,N)}\dagger[\mathsf{true}])\\
&\quad\triangleright\{k_{(1,2)}: T_{(1,2)};![T_{(1,2)}]^*\circ T'_{(1,2)};?[T'_{(1,2)}]^*,\\
&\qquad k_{(1,3)}: T_{(1,3)};![T_{(1,3)}]^*\circ T'_{(1,3)};?[T'_{(1,3)}]^{*\dagger},\\
&\qquad\qquad\ldots,k_{(1,N)}: T_{(1,N)};![T_{(1,N)}]^*\circ T'_{(1,N)};?[T'_{(1,N)}]^{*\dagger},\\
&\qquad k_{(2,3)}: ![T_{(2,3)}]^*\circ?[T'_{(2,3)}]^*,\ldots,k_{(N-1,N)}: ![T_{(N-1,N)}]^*\circ?[T'_{(N-1,N)}]^*
\end{aligned}$$

By [Ow1],

$$
\begin{aligned}
& \nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\}\\
&\quad \mid k_{(1,3)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(1,N)} \dagger [\mathsf{true}]\\
&\quad \mid \langle k_{(2,3)},\ldots,k_{(2,N)}\rangle.\mathsf{outwhile}(\mathsf{true})\{\ Q_2[k_{(2,3)},\ldots,k_{(2,N)},k_{(1,2)}]\ \}\\
&\quad \mid \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\ \text{when}\ i\in\{3..M-1\}\\
&\quad \mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when}\ j\in\{M..N\}\\
\to^*\ & \nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\}\\
&\quad \mid k_{(2,3)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(2,N)} \dagger [\mathsf{true}] \mid k_{(1,3)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(1,N)}\\
&\quad \mid Q_2[k_{(2,3)},\ldots,k_{(2,N)},k_{(1,2)}];\ \langle k_{(2,3)},\ldots,k_{(2,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,2)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_2[k_{(2,3)},\ldots,k_{(2,N)},k_{(1,2)}]\ \}\\
&\quad \mid \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\ \text{when}\ i\in\{3..M-1\}\\
&\quad \mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when}\ j\in\{M..N\}
\end{aligned}
$$

$$
\begin{aligned}
& \Gamma \vdash (Q_1;P_1 \mid Q_2;P_2 \mid P_{i\in 2..M-1} \mid P_{j\in M..N} \mid k_{(2,3)} \dagger [\mathsf{true}] \mid\\
&\quad \ldots \mid k_{(2,N)} \dagger [\mathsf{true}] \mid k_{(1,3)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(N-1,N)} \dagger [\mathsf{true}])\\
&\ \triangleright\{k_{(1,2)} : T_{(1,2)};![T_{(1,2)}]^* \circ T'_{(1,2)};?[T'_{(1,2)}]^*,\\
&\quad k_{(2,3)} : T_{(2,3)};![T_{(2,3)}]^* \circ T'_{(2,3)};?[T'_{(2,3)}]^{*\dagger},\\
&\quad \ldots, k_{(2,N)} : T_{(2,N)};![T_{(2,N)}]^* \circ T'_{(2,N)};?[T'_{(2,N)}]^{*\dagger}\\
&\quad k_{(1,3)} : T_{(1,3)};![T_{(1,3)}]^* \circ T'_{(1,3)};?[T'_{(1,3)}]^{*\dagger},\\
&\quad \ldots, k_{(1,N)} : T_{(1,N)};![T_{(1,N)}]^* \circ T'_{(1,N)};?[T'_{(1,N)}]^{*\dagger},\\
&\quad k_{(3,4)} : ![T_{(3,4)}]^*\circ ?[T'_{(3,4)}]^*,\ldots,k_{(N-1,N)} : ![T_{(N-1,N)}]^*\circ ?[T'_{(N-1,N)}]^*
\end{aligned}
$$

By repeatedly apply [Ow1] and [IwE1] as above on processes $P_3..P_{M-1}$

$$
\begin{aligned}
& \nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\} \\
&\quad \mid k_{(2,3)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(2,N)} \dagger [\mathsf{true}] \mid k_{(1,3)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(N-1,N)} \\
&\quad \mid Q_2[k_{(2,3)},\ldots,k_{(2,N)},k_{(1,2)}];\langle k_{(2,3)},\ldots,k_{(2,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,2)}\rangle.\mathsf{inwhile})\{ \\
&\qquad Q_2[k_{(2,3)},\ldots,k_{(2,N)},k_{(1,2)}]\ \} \\
&\quad \mid \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{ \\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\ \text{when } i \in \{3..M-1\} \\
&\quad \mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_i[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when } j \in \{M..N\} \\
\to^*\to^*\ & \nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\} \\
&\quad \mid k_{(1,j)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(j-1,j)} \dagger [\mathsf{true}] \\
&\quad \mid Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]; \\
&\qquad \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{ \\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\ \}\ \text{when } i \in \{2..M-1\} \\
&\quad \mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when } j \in \{M..N\}
\end{aligned}
$$

$$
\begin{aligned}
& \Gamma \vdash (Q_1;P_1 \mid Q_i;P_{i\in 2..M-1} \mid P_{j\in M..N} \mid k_{(1,j)} \dagger [\mathsf{true}] \mid \ldots \mid k_{(j-1,j)} \dagger [\mathsf{true}]) \\
&\quad \triangleright \{k_{(1,2)} : T_{(1,2)};![T_{(1,2)}]^* \circ T'_{(1,2)};?[T'_{(1,2)}]^*, \\
&\qquad\qquad \ldots, k_{(1,N)} : T_{(1,N)};![T_{(1,N)}]^* \circ T'_{(1,N)};?[T'_{(1,N)}]^*, \\
&\qquad k_{(i,i+1)} : T_{(i,i+1)};![T_{(i,i+1)}]^* \circ T'_{(i,i+1)};?[T'_{(i,i+1)}]^*, \\
&\qquad\qquad \ldots, k_{(i,M-1)} : T_{(i,M-1)};![T_{(i,M-1)}]^* \circ T'_{(i,M-1)};?[T'_{i,M-1}]^*, \\
&\qquad k_{(1,j)} : ![T_{(1,j)}]^* \circ ?[T'_{(1,j)}]^{*\dagger}, \ldots, k_{(j-1,j)} : ![T_{(j-1,j)}]^* \circ ?[T'_{(j-1,j)}]^{*\dagger}\}
\end{aligned}
$$

Finally apply [Iw1],

$$
\begin{aligned}
&\nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\}\\
&\quad \mid k_{(1,j)}\dagger[\mathsf{true}]\mid\ldots\mid k_{(j-1,j)}\dagger[\mathsf{true}]\\
&\quad \mid Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}];\\
&\qquad \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\ \}\text{ when } i\in\{2..M-1\}\\
&\quad \mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\text{ when } j\in\{M..N\}\\
\to^*\ &\nu\tilde{k}\ (Q_1[k_{(1,2)},\ldots,k_{(1,N)}];\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\}\\
&\quad \mid Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}];\\
&\qquad \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\text{ when } i\in\{2..M-1\}\\
&\quad \mid Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}];\\
&\qquad \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\text{ when } j\in\{M..N\}
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash(Q_1;P_1\mid Q_i;P_{i\in 2..M-1}\mid Q_j;P_{j\in M..N})\\
&\quad\triangleright\{k_{(1,2)}:T_{(1,2)};![T_{(1,2)}]^*\circ T'_{(1,2)};?[T'_{(1,2)}]^*,\\
&\qquad\qquad\ldots,k_{(1,N)}:T_{(1,N)};![T_{(1,N)}]^*\circ T_{(1,N)};?[T_{(1,N)}]^*,\\
&\qquad k_{(i,i+1)}:T_{(i,i+1)};![T_{(i,i+1)}]^*\circ T'_{(i,i+1)};?[T'_{(i,i+1)}]^*,\\
&\qquad\qquad\ldots,k_{(i,N)}:T_{(i,N)};![T_{(i,N)}]^*\circ T'_{(i,N)};?[T'_{(i,N)}]^*\}\\
&\Gamma\vdash(Q_1;P_1\mid Q_i;P_{i\in 2..M-1}\mid Q_j;P_{j\in M..N})\\
&\quad\triangleright\{k_{(1,2)}:\bot,\ldots,k_{(1,N)}:\bot,k_{(i,i+1)}:\bot,\ldots,k_{(i,N)}:\bot\}
\end{aligned}
$$

Case $E[e]\to E[\mathsf{false}]$

By [Ow2],

$$
\begin{aligned}
&\nu\tilde{k}\ (\langle k_{(1,2)},\ldots,k_{(1,N)}\rangle.\mathsf{outwhile}(e)\{Q_1[k_{(1,2)},\ldots,k_{(1,N)}]\}\\
&\quad\mid \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\ \text{when } i\in\{2..M-1\}\\
&\quad\mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_i[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when } j\in\{M..N\}\\
\to^*\ &\nu\tilde{k}\ (\ \mathsf{0}\mid k_{(1,2)}\dagger[\mathsf{false}]\mid\ldots\mid k_{(1,N)}\dagger[\mathsf{false}]\\
&\quad\mid \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\ \text{when } i\in\{2..M-1\}\\
&\quad\mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_i[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when } j\in\{M..N\}
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash(\mathsf{0}\mid P_{i\in 2..M-1}\mid P_{j\in M..N}\mid k_{(1,2)}\dagger[\mathsf{false}]\mid\ldots\mid k_{(1,N)}\dagger[\mathsf{false}])\\
&\quad\triangleright\{k_{(1,2)}:\tau.\mathsf{end}\circ?[T_{(1,2)}]^{*\dagger},\ldots,k_{(1,N)}:\tau.\mathsf{end}\circ?[T_{(1,N)}]^{*\dagger},\\
&\qquad k_{(i,i+1)}:![T_{(i,i+1)}]^{*}\circ?[T'_{(i,i+1)}]^{*},\ldots,k_{(i,N)}:![T_{(i,N)}]^{*}\circ?[T'_{(i,N)}]^{*}\}
\end{aligned}
$$

By [IwE2],

$$
\begin{aligned}
&\nu\tilde{k}\ (\mathsf{0}\mid k_{(1,2)}\dagger[\mathsf{false}]\mid\ldots\mid k_{(1,N)}\dagger[\mathsf{false}]\\
&\quad\mid \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\ \text{when } i\in\{2..M-1\}\\
&\quad\mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_i[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when } j\in\{M..N\}\\
\to^*\to\ &\tilde{k}\ (\mathsf{0}\mid k_{(1,3)}\dagger[\mathsf{false}]\mid\ldots\mid k_{(1,N)}\dagger[\mathsf{false}]\\
&\quad\mid \langle k_{(2,3)},\ldots,k_{(2,N)}\rangle.\mathsf{outwhile}(\ \mathsf{false}\ )\{\\
&\qquad Q_i[k_{(2,3)},\ldots,k_{(2,N)},k_{(1,2)}]\}\\
&\quad\mid \langle k_{(i,i+1)},\ldots,k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)},\ldots,k_{(i-1,i)}\rangle.\mathsf{inwhile})\{\\
&\qquad Q_i[k_{(i,i+1)},\ldots,k_{(i,N)},k_{(1,i)},\ldots,k_{(i-1,i)}]\}\ \text{when } i\in\{3..M-1\}\\
&\quad\mid \langle k_{(1,j)},\ldots,k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)},\ldots,k_{(j-1,j)}]\})\ \text{when } j\in\{M..N\}
\end{aligned}
$$

$$\Gamma \vdash (0 \mid P_{i\in 2..M-1} \mid P_{j\in M..N})$$
$$\triangleright\{k_{(1,2)} : \tau.\mathsf{end} \circ \tau.\mathsf{end}, k_{(1,3)} : \tau.\mathsf{end}\circ?[T'_{(1,3)}]^{*\dagger} \ldots, k_{(1,N)} : \tau.\mathsf{end}\circ?[T'_{(1,N)}]^{*\dagger}$$
$$k_{(i,i+1)} : ![T_{(i,i+1)}]^{*}\circ?[T'_{(i,i+1)}]^{*}, \ldots, k_{(i,N)} : ![T_{(i,N)}]^{*}\circ?[T'_{(i,N)}]^{*}\}$$

By [Ow2],

$$\nu\tilde{k}\ (0 \mid k_{(1,3)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(1,N)} \dagger [\mathsf{false}]$$
$$\mid \langle k_{(2,3)}, \ldots, k_{(2,N)}\rangle.\mathsf{outwhile}(\ \mathsf{false}\ )\{$$
$$Q_i[k_{(2,3)}, \ldots, k_{(2,N)}, k_{(1,2)}]\}$$
$$\mid \langle k_{(i,i+1)}, \ldots, k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)}, \ldots, k_{(i-1,i)}\rangle.\mathsf{inwhile})\{$$
$$Q_i[k_{(i,i+1)}, \ldots, k_{(i,N)}, k_{(1,i)}, \ldots, k_{(i-1,i)}]\}\ \text{when}\ i \in \{3..M-1\}$$
$$\mid \langle k_{(1,j)}, \ldots, k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\})\ \text{when}\ j \in \{M..N\}$$
$$\rightarrow^{*}\ \nu\tilde{k}\ (0\ \mid k_{(2,3)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(2,N)} \dagger [\mathsf{false}] \mid k_{(1,3)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(1,N)} \dagger [\mathsf{false}]$$
$$\mid\ 0$$
$$\mid \langle k_{(i,i+1)}, \ldots, k_{(i,N)}\rangle.\mathsf{outwhile}(\langle k_{(1,i)}, \ldots, k_{(i-1,i)}\rangle.\mathsf{inwhile})\{$$
$$Q_i[k_{(i,i+1)}, \ldots, k_{(i,N)}, k_{(1,i)}, \ldots, k_{(i-1,i)}]\}\ \text{when}\ i \in \{3..M-1\}$$
$$\mid \langle k_{(1,j)}, \ldots, k_{(j-1,j)}\rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\})\ \text{when}\ j \in \{M..N\}$$

$$\Gamma \vdash (0 \mid k_{(2,3)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(2,N)} \dagger [\mathsf{false}] \mid k_{(1,3)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(1,N)} \dagger [\mathsf{false}]$$
$$\mid 0 \mid P_i\ \text{when}\ i \in \{3..M-1\} \mid P_j\ \text{when}\ j \in \{M..N\})$$
$$\triangleright\{k_{(1,2)} : \tau.\mathsf{end}, k_{(1,3)} : \tau.\mathsf{end}\circ?[T'_{(1,3)}]^{*\dagger} \ldots, k_{(1,N)} : \tau.\mathsf{end}\circ?[T'_{(1,N)}]^{*\dagger}$$
$$k_{(2,3)} : \tau.\mathsf{end}\circ?[T'_{(2,3)}]^{*\dagger}, \ldots, k_{(2,N)} : \tau.\mathsf{end}\circ?[T'_{(2,N)}]^{*\dagger},$$
$$k_{(i,i+1)} : ![T_{(i,i+1)}]^{*}\circ?[T'_{(i,i+1)}]^{*}, \ldots, k_{(i,N)} : ![T_{(i,N)}]^{*}\circ?[T'_{(i,N)}]^{*}\}$$

By repeatedly apply [Ow2] and [IwE2] as above on processes $P_3..P_{M-1}$

$$\begin{aligned}
&\nu\tilde{k}\ (\mathsf{0}\ \mid k_{(2,3)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(2,N)} \dagger [\mathsf{false}] \mid k_{(1,3)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(1,N)} \dagger [\mathsf{false}] \\
&\quad \mid \mathsf{0} \\
&\quad \mid \langle k_{(i,i+1)}, \ldots, k_{(i,N)} \rangle.\mathsf{outwhile}(\langle k_{(1,i)}, \ldots, k_{(i-1,i)} \rangle.\mathsf{inwhile})\{ \\
&\qquad Q_i[k_{(i,i+1)}, \ldots, k_{(i,N)}, k_{(1,i)}, \ldots, k_{(i-1,i)}]\} \text{ when } i \in \{3..M-1\} \\
&\quad \mid \langle k_{(1,j)}, \ldots, k_{(j-1,j)} \rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\}) \text{ when } j \in \{M..N\} \\
\to^*\to^*\ &\nu\tilde{k}\ (\mathsf{0} \mid \mathsf{0} \text{ when } i \in \{2..M-1\} \mid k_{(1,j)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(j-1,j)} \dagger [\mathsf{false}] \\
&\quad \mid \langle k_{(1,j)}, \ldots, k_{(j-1,j)} \rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\} \text{ when } j \in \{M..N\})
\end{aligned}$$

$$\begin{aligned}
&\Gamma \vdash (\mathsf{0} \mid \mathsf{0} \text{ when } i \in \{2..M-1\} \mid k_{(1,j)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(j-1,j)} \dagger [\mathsf{false}] \mid P_{j \in M..N}) \\
&\quad \triangleright \{k_{(1,2)} : \tau.\mathsf{end}, \ldots, k_{(1,N)} : \tau.\mathsf{end}, k_{(i,i+1)} : \tau.\mathsf{end}, \ldots, k_{(i,M-1)} : \tau.\mathsf{end} \circ ?[T'_{(i,M-1)}]^*, \\
&\quad\quad k_{(1,j)} : \tau.\mathsf{end} \circ ?[T'_{(1,j)}]^{*\dagger}, \ldots, k_{(j-1,j)} : \tau.\mathsf{end} \circ ?[T'_{j-1,j}]^{*\dagger}\}
\end{aligned}$$

Finally apply [Iw2],

$$\begin{aligned}
&\nu\tilde{k}\ (\mathsf{0} \mid \mathsf{0} \text{ when } i \in \{2..M-1\} \mid k_{(1,j)} \dagger [\mathsf{false}] \mid \ldots \mid k_{(j-1,j)} \dagger [\mathsf{false}] \\
&\quad \mid \langle k_{(1,j)}, \ldots, k_{(j-1,j)} \rangle.\mathsf{inwhile}\{Q_j[k_{(1,j)}, \ldots, k_{(j-1,j)}]\}) \text{ when } j \in \{M..N\} \\
\to^*\ &\nu\tilde{k}\ (\mathsf{0} \mid \mathsf{0} \text{ when } i \in \{2..M-1\}\ \mid \mathsf{0} \text{ when } j \in \{M..N\}\ )
\end{aligned}$$

$$\begin{aligned}
&\Gamma \vdash (\mathsf{0} \mid \mathsf{0} \text{ when } i \in \{2..M-1\} \mid \mathsf{0} \text{ when } j \in \{M..N\}) \\
&\quad \triangleright \{k_{(1,2)} : \tau.\mathsf{end}, \ldots, k_{(1,N)} : \tau.\mathsf{end}, k_{(i,i+1)} : \tau.\mathsf{end}, \ldots, k_{(i,N)} : \tau.\mathsf{end}\}
\end{aligned}$$

Finally, apply [Bot].
For other cases, the proof is similar to [Ng10, P. 56-60] □

Theorem 7.3 (Deadlock Freedom) *Assume P forms a well-formed topology and $\Gamma \vdash P \triangleright \Delta$. Then P is deadlock-free.*
Pooof.

Assume $\Gamma \vdash \Pi_i P_i \triangleright \vec{k} \colon \vec{\bot}$ for all cases below. Suppose group of n parallel composed processes $\Pi_i P_i = P_1 \mid \ldots \mid P_n$ conforms to a well-formed topology (Definition 7.1).

Case 1.1 inwhile and outwhile, condition true.

$$\begin{array}{ll} P_1 = \langle \vec{k}_1 \rangle.\mathsf{outwhile}(\mathsf{true})\{Q_1\} & P_i = \langle \vec{k}_i \rangle.\mathsf{outwhile}(\langle \vec{k}'_i \rangle.\mathsf{inwhile})\{Q_i\} \ (2 \le i < n) \\ P_n = \langle \vec{k}'_n \rangle.\mathsf{inwhile}\{Q_n\} & \vec{k}_i \subset k_{(i,i+1)} \cdots k_{(i,n)}, \vec{k}'_i \subset k_{(1,i)} \cdots k_{(i-1,i)} \end{array}$$

If the $\mathsf{outwhile}$ condition is true, the iteration chain passes the true condition from the Master process, P_1 to all other processes, at the end of the iteration chain, all processes reduce to $Q_i; P_i$ where Q_i is the loop body and P_i is the next iteration of the $\mathsf{outwhile/inwhile}$ loop. We will show inductively that Q_i is deadlock free in other cases listed below. The session channel interaction sequence is shown below.

$$\begin{array}{llllllll} P_1 = & \xrightarrow{k_{(1,2)}}{}^{*} & \xrightarrow{k_{(1,i)}}{}^{*} & \xrightarrow{k_{(1,n)}}{}^{*} & & & & Q_1; P_1 \\ P_i = & & \xrightarrow{\overline{k}_{(1,i)}}{}^{*} & & \xrightarrow{\overline{k}_{(i-1,i)}}{}^{*} & \xrightarrow{k_{(i,i+1)}}{}^{*} & \xrightarrow{k_{(i,n)}}{}^{*} & Q_i; P_i \\ P_n = & & & \xrightarrow{\overline{k}_{(1,n)}}{}^{*} & & & \xrightarrow{\overline{k}_{(i,n)}}{}^{*} & Q_n; P_n \end{array}$$

The Master process P_1 initiates the interactions in each $\mathsf{outwhile/inwhile}$ iteration chain. All session interactions in each process happen after all interactions on the left is completed. From above interaction sequence, there are no processes that can only proceed depending on an interaction step not readily available. Therefore a correct process P_i will always reduce to $Q_i; P_i$ for a true condition.

Case 1.2 inwhile and outwhile, condition false.

$$\begin{array}{ll} P_1 = \langle \vec{k}_1 \rangle.\mathsf{outwhile}(\mathsf{false})\{Q_1\} & P_i = \langle \vec{k}_i \rangle.\mathsf{outwhile}(\langle \vec{k}'_i \rangle.\mathsf{inwhile})\{Q_i\} \ (2 \le i < n) \\ P_n = \langle \vec{k}'_n \rangle.\mathsf{inwhile}\{Q_n\} & \vec{k}_i \subset k_{(i,i+1)} \cdots k_{(i,n)}, \vec{k}'_i \subset k_{(1,i)} \cdots k_{(i-1,i)} \end{array}$$

Suppose group of parallel composed processes $\Pi_i P_i = P_1 \mid \ldots \mid P_n$ conforms to a well-formed topology (Definition 7.1). If the $\langle c \rangle.\mathsf{outwhile}(o)\{n\}$ dition is false, the iteration chain passes the false condition from the Master process, P_1 to all other processes, at the end of the iteration chain, all processes reduces to $\mathsf{0}$ to exit from the $\mathsf{outwhile}/\mathsf{inwhile}$ loop. The session channel interaction sequence is shown below:

$$\begin{array}{llllllll}
P_1 = & \xrightarrow{k_{(1,2)}}{}^{*} & \xrightarrow{k_{(1,i)}}{}^{*} & \xrightarrow{k_{(1,n)}}{}^{*} & & & & \mathsf{0} \\
P_i = & & \xrightarrow{\overline{k}_{(1,i)}}{}^{*} & & \xrightarrow{\overline{k}_{(i-1,i)}}{}^{*} & \xrightarrow{k_{(i,i+1)}}{}^{*} & \xrightarrow{k_{(i,n)}}{}^{*} & \mathsf{0} \\
P_n = & & & \xrightarrow{\overline{k}_{(1,n)}}{}^{*} & & & \xrightarrow{\overline{k}_{(i,n)}}{}^{*} & \mathsf{0}
\end{array}$$

The interaction sequence is same as the first case, therefore all processesP_i can reduce to $\mathsf{0}$ with similar reasoning.

Case 2. sequencing**.** Suppose for a simple case $P_i = Q_{1i}; Q_{2i}$, and both $\Pi_i Q_{1i}$ and $\Pi_i Q_{2i}$ are deadlock free. By Definition A.3, sequential composition will not permute the order of communication of each of the processes. Therefore $\Pi_i P_i$ is deadlock free.

We can show that $\Pi_i P_i$ with $P_i = Q_{i1}; Q_{i2}; ...; Q_{in}$ is deadlock free if all the subprocesses are deadlock free by induction.

Case 3. base**.** This case considers a group of processes P_i which do not include shared session channels, $\mathsf{inwhile}$ nor $\mathsf{outwhile}$. The session actions in each of P_i follows the order of the index, similar to Case 1.1 and Case 1.2. The session channel interaction sequence is shown below:

$$\begin{array}{llllllll}
P_1 = & \xrightarrow{k_{(1,2)}}{}^{*} & \xrightarrow{k_{(1,i)}}{}^{*} & \xrightarrow{k_{(1,n)}}{}^{*} & & & & P_1' \\
P_i = & & \xrightarrow{\overline{k}_{(1,i)}}{}^{*} & & \xrightarrow{\overline{k}_{(i-1,i)}}{}^{*} & \xrightarrow{k_{(i,i+1)}}{}^{*} & \xrightarrow{k_{(i,n)}}{}^{*} & P_i' \\
P_n = & & & \xrightarrow{\overline{k}_{(1,n)}}{}^{*} & & & \xrightarrow{\overline{k}_{(i,n)}}{}^{*} & P_n'
\end{array}$$

The body of the process is deadlock free with same reasoning in Case 1.1, which then reduces to a *base* process P_i', and $\Pi_i P_i'$ reduces to $\mathsf{0}$ by the reason described in Section 7.4.3. □

A.1.4. Full Process Definitions

This subsection presents the full process definitions and session typing of example parallel algorithms (N-body simulation and Jacobi solution) implemented in multi-channel session calculus omitted from Chapter 7. By showing the full process definition and typing, we show that they conform to well-formed ring and mesh topologies respectively, and hence are type and communication safe, which are both subset of our general definition of well-formed topology (Definition 7.1).

A.1.5. 3-node n-body simulation

$$
\begin{aligned}
&P_1 \equiv \langle k_{(1,2)}, k_{(1,3)} \rangle.\mathsf{outwhile}(e)\{\overline{k_{(1,2)}}\langle \texttt{Particle[]} \rangle; k_{(1,3)}(x).0\} \\
&P_2 \equiv \langle k_{(2,3)} \rangle.\mathsf{outwhile}(\langle k_{(1,2)} \rangle.\mathsf{inwhile})\{\overline{k_{(2,3)}}\langle \texttt{Particle[]} \rangle; k_{(1,2)}(x).0\} \\
&P_3 \equiv \langle k_{(1,3)}, k_{(2,3)} \rangle.\mathsf{inwhile}\{\overline{k_{(1,3)}}\langle \texttt{Particle[]} \rangle; k_{(2,3)}(x).0\}
\end{aligned}
$$

where the typing of the processes are:

$$
\begin{aligned}
\Gamma \vdash P_1 \triangleright \{\overline{k_{(1,2)}} : ![![U]; \mathsf{end}]^*, \overline{k_{(1,3)}} : ![?[U]; \mathsf{end}]^*\} \\
\Gamma \vdash P_2 \triangleright \{k_{(1,2)} : ?[?[U]; \mathsf{end}]^*, \overline{k_{(2,3)}} : ![![U]; \mathsf{end}]^*\} \\
\Gamma \vdash P_3 \triangleright \{k_{(1,3)} : ?[![U]; \mathsf{end}]^*, k_{(2,3)} : ?[?[U]; \mathsf{end}]^*\}
\end{aligned}
$$

A.1.6. Jacobi solution

Below is the full process definition and typing for the Jacobi solution.

$$
\begin{aligned}
P_1 = P_{NW} =\ & \langle k_{(1,2)}, k_{(1,4)} \rangle.\mathsf{outwhile}(e)\{\\
& \overline{k_{(1,2)}}\langle \texttt{double[]} \rangle; k_{(1,2)}(x).\overline{k_{(1,4)}}\langle \texttt{double[]} \rangle;\\
& k_{(1,4)}(y).0\}\\
P_2 = P_N =\ & \langle k_{(2,4)}, k_{(2,5)} \rangle.\mathsf{outwhile}(\langle \overline{k_{(1,2)}} \rangle.\mathsf{inwhile})\{\\
& k_{(1,2)}(x).\overline{k_{(1,2)}}\langle \texttt{double[]} \rangle; \overline{k_{(2,4)}}\langle \texttt{double[]} \rangle;\\
& k_{(2,4)}(y).\overline{k_{(2,5)}}\langle \texttt{double[]} \rangle; k_{(2,5)}(z).0\}\\
P_4 = P_{NE} =\ & \langle k_{(4,7)} \rangle.\mathsf{outwhile}(\langle \overline{k_{(2,4)}} \rangle.\mathsf{inwhile})\{\\
& k_{(2,4)}(x).\overline{k_{(2,4)}}\langle \texttt{double[]} \rangle; \overline{k_{(4,7)}}\langle \texttt{double[]} \rangle;\\
& k_{(4,7)}(y).0\}\\
P_3 = P_W =\ & \langle k_{(3,5)}, k_{(3,6)} \rangle.\mathsf{outwhile}(\langle k_{(1,3)} \rangle.\mathsf{inwhile})\{\\
& k_{(1,3)}(x).\overline{k_{(1,3)}}\langle \texttt{double[]} \rangle; \overline{k_{(3,5)}}\langle \texttt{double[]} \rangle;\\
& k_{(3,5)}(y).\overline{k_{(3,6)}}\langle \texttt{double[]} \rangle; k_{(3,6)}(z).0\}\\
P_5 = P_C =\ & \langle k_{(5,7)}, k_{(5,8)} \rangle.\mathsf{outwhile}(\langle k_{(2,5)}, k_{(3,5)} \rangle.\mathsf{inwhile})\{\\
& k_{(2,5)}(w).\overline{k_{(2,5)}}\langle \texttt{double[]} \rangle; k_{(3,5)}(x).\\
& \overline{k_{(3,5)}}\langle \texttt{double[]} \rangle; \overline{k_{(5,7)}}\langle \texttt{double[]} \rangle; k_{(5,7)}(y).\overline{k_{(5,8)}}\langle \texttt{double[]} \rangle;\\
& k_{(5,8)}(z).0\}\\
P_7 = P_E =\ & \langle k_{(7,9)} \rangle.\mathsf{outwhile}(\langle k_{(4,7)}, k_{(5,7)} \rangle.\mathsf{inwhile})\{\\
& k_{(4,7)}(x).\overline{k_{(4,7)}}\langle \texttt{double[]} \rangle; k_{(5,7)}(y).\overline{k_{(5,7)}}\langle \texttt{double[]} \rangle;\\
& \overline{k_{(7,9)}}\langle \texttt{double[]} \rangle; k_{(7,9)}(z).0\}\\
P_6 = P_{SW} =\ & \langle k_{(6,8)} \rangle.\mathsf{outwhile}(\langle k_{(3,6)} \rangle.\mathsf{inwhile})\{\\
& k_{(3,6)}(x).\overline{k_{(3,6)}}\langle \texttt{double[]} \rangle;\\
& \overline{k_{(6,8)}}\langle \texttt{double[]} \rangle; k_{(6,8)}(y).0\}\\
P_8 = P_S =\ & \langle k_{(8,9)} \rangle.\mathsf{outwhile}(\langle k_{(5,8)}, k_{(6,8)} \rangle.\mathsf{inwhile})\{\\
& k_{(5,8)}(x).\overline{k_{(5,8)}}\langle \texttt{double[]} \rangle; k_{(6,8)}(y).\overline{k_{(6,8)}}\langle \texttt{double[]} \rangle;\\
& \overline{k_{(6,8)}}\langle \texttt{double[]} \rangle; k_{(6,8)}(z).0\}\\
P_9 = P_{SE} =\ & \langle k_{(7,9)}, k_{(8,9)} \rangle.\mathsf{inwhile}\{\\
& k_{(7,9)}(x).\overline{k_{(7,9)}}\langle \texttt{double[]} \rangle;\\
& k_{(8,9)}(x).\overline{k_{(8,9)}}\langle \texttt{double[]} \rangle; 0\}
\end{aligned}
$$

where the typing of the processes are:

$$\begin{aligned}
\Gamma \vdash P_1 \triangleright \{ & \overline{k_{(1,2)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(1,3)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_2 \triangleright \{ & k_{(1,2)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(2,4)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(2,5)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_4 \triangleright \{ & k_{(2,4)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(4,7)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_3 \triangleright \{ & k_{(1,3)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(3,5)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(3,6)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_5 \triangleright \{ & k_{(2,5)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& k_{(3,5)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(5,7)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(5,8)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_7 \triangleright \{ & k_{(4,7)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& k_{(5,7)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(7,9)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_6 \triangleright \{ & k_{(3,6)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(6,8)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_8 \triangleright \{ & k_{(5,8)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& k_{(6,8)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& \overline{k_{(8,9)}} : ![![\texttt{double[]}]; ?[\texttt{double[]}]; \mathsf{end}]^* \} \\
\Gamma \vdash P_9 \triangleright \{ & k_{(7,9)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^*, \\
& k_{(8,9)} : ?[?[\texttt{double[]}]; ![\texttt{double[]}]; \mathsf{end}]^* \}
\end{aligned}$$

Now we reason Jacobi algorithm in Figure 7.9. We only show the master P_1 and the worker in the middle P_5 (the indices follow the right picture of Figure 7.7).

$$
\begin{aligned}
P_1 &= \langle k_{(1,2)}, k_{(1,4)} \rangle.\mathsf{outwhile}(e)\{\overline{k_{(1,2)}}\langle \mathtt{d}[] \rangle; k_{(1,2)}(x).\overline{k_{(1,4)}}\langle \mathtt{d}[] \rangle; k_{(1,4)}(y).\mathtt{0}\} \\
P_5 &= \langle k_{(5,7)}, k_{(5,8)} \rangle.\mathsf{outwhile}(e)\{\langle k_{(2,5)}, k_{(3,5)} \rangle.\mathsf{inwhile}\{ \\
&\qquad k_{(2,5)}(x).\overline{k_{(2,5)}}\langle \mathtt{d}[] \rangle; \overline{k_{(5,7)}}\langle \mathtt{d}[] \rangle; k_{(5,7)}(y).\overline{k_{(5,8)}}\langle \mathtt{d}[] \rangle; k_{(5,8)}(z).\mathtt{0}\}\}
\end{aligned}
$$

Given the process definition and its types, we can easily prove that they are typable and conforms to the definition of well-formed topology, satisfying the conditions (1) and (3) in Definition 7.1. Hence it is type and communication-safe (Theorem 7.2) and deadlock-free (Theorem 7.3).