UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

FORMAL VERIFICATION OF PARALLEL C+MPI PROGRAMS

Nuno Alexandre Dias Martins

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

FORMAL VERIFICATION OF PARALLEL C+MPI PROGRAMS

Nuno Alexandre Dias Martins

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Eduardo Resende Brandão Marques
e co-orientado pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

2013

Agradecimentos

Quero começar por agradecer a todos os que estiveram presentes durante estes 85 meses da minha vida, todos vós fizeram parte da minha vida e tornaram-me o que hoje sou.

Em primeiro lugar, quero agradecer ao meu pai e mãe por me terem trazido ao mundo e terem-me dado tudo e ensinado para ser bem sucedido e humilde, pois sempre me apoiaram e deram liberdade para tomar as minhas decisões nesta fase da vida, e a eles ficarei eternamente grato. Ao meu irmão também deixo um enorme obrigado por fazer o papel de irmão mais velho em todos aspectos e ser um exemplo a seguir. Aos meus avós, que apesar de já não estarem presentes entre nós, sei que estão sempre a dar-me forças para ultrapassar as dificuldades.

Agradeço também a toda a minha família, primos, tios, por me fazerem sentir em casa e darem um verdadeiro significado à palavra "família". Aproveito para agradecer mais uma vez ao meu irmão e também à Joana Almeida, por me terem dado o meu primeiro sobrinho, o João, que em breve fará 1 ano de vida e está sempre pronto para a brincadeira, com excepção da hora de comer, que para ele é sagrada. Apesar de não partilharmos o mesmo sangue, sempre foi como um irmão para mim e sempre esteve disponível para mim a qualquer hora e qualquer dia desde o meu primeiro dia de vida, ao Luís Courinha, meu irmão de pais diferentes, um enorme obrigado por estares sempre presente em todas as fases da minha vida, esta inclusive.

Como uma das fases da minha vida passou pela praxe, quero agradecer a todos os que se aventuram e que me fizeram aventurar nessa altura, em particular aos meus afilhados mais ativos (Cardozo, Garoto e Preto), ao Pombal que sempre me acompanhou nas maluqueiras, ao meu padrinho (João Casais), ao meu irmão de praxe que mais me desafiou nas aventuras, Vieira (Besta Marisco) e a toda a COPA do meu ano. Também a todos os membros da CADI que me ajudaram a organizar Informanias e o ENEI, entre eles, um obrigado especial à Diana Coelho (que tantas vezes me levou a casa), ao Cabaço (que também me levou a casa e acompanhou em projectos) e ao Açoreano (que desde o primeiro ano me apresentou a uma grande diversão no "Cenoura"e me acolheu e deu a conhecer São Miguel).

Queria deixar um agradecimento especial à professora Ana Paula Cláudio que me iniciou na investigação e sempre foi uma professora exemplar, disponível e atenciosa.

Não poderia deixar de agradecer ao meu orientador Eduardo Marques pela sua paciência durante o projecto e por tanto me ter ajudado na escrita desta dissertação. Também ao professor Vasco Vasconcelos deixo o meu sincero obrigado, meu co-orientador que conheci no meu primeiro ano a AED, que foi um dos pilares na minha aprendizagem de BOA programação e me deu a oportunidade de conhecer outro mundo, Glasgow. Também deixo aqui o meu agradecimento ao César e ao Francisco Martins que fizeram parte do projecto ATSMP e ajudaram no projecto e escrita de papers. Não podia deixar de agradecer à Juliana que sempre foi uma óptima colega de curso e de grupo de investigação.

Da minha curta estadia em Glasgow tenho de agradecer ao Professor Simon Gay que me acolheu de braços abertos e teve a paciência de partilhar um pouco do seu conhecimento comigo. Também queria agradecer ao Michele Sevegnani que apesar do pouco tempo que lá estive se tornou um amigo, me acolheu e integrou durante a minha estadia e também me deu a conhecer o maravilhoso licor italiano, Disaronno. Também agradeço ao Andrea Degasperi por tudo que fez por mim.

Ao Gonçalo Gomes, Ruben e Gonçalo Graça, um obrigado por me terem aturado e me terem proporcionado momentos inesquecíveis, quer nas funções de monitor, na CADI, no desporto, nos convívios e em muitas outras ocasiões.

Um agradecimento aos meus amigos picarotos de longa data, Luís Paulo e David Souto, que me integraram na comunidade picarota, sempre estiveram presentes para mim e me ajudaram em todo o tipo de ocasiões, a vocês um enorme obrigado e que haja em breve um jantar com “franguinho núcú".

Ao Carlos Mão de Ferro, um agradecimento muito especial por nestes últimos 3 anos ter sido um colega de trabalho exemplar e divertido, por ter sido doido o suficiente para entrar na CADI e alinhar nas minhas ideias, mas também por ter ideias doidas da mesma forma, por ter sido um óptimo colega de projectos e acima de tudo por ter se tornado um grande amigo. Que continues a dar festas na tua casa e possamos gritar finalmente "Benfica Campeão!"

Antes do último e mais significativo agradecimento, quero agradecer ao Bruno Lima (Espanhol) por me ter apresentado a Mariana Avelar, a minha cara metade e a toda a família da Mariana que desde do primeiro dia me acolheu como família. Não teria conseguido nem metade do que alcancei se não fosse a minha namorada e alma gêmea, Mariana Avelar. A ela agradeço todo o apoio incondicional que me deu em todas as ocasiões da minha vida, por me ter ajudado na escrita desta dissertação, ter relido tantas vezes quanto eu e perdido horas de sono por esta dissertação. Por me estar sempre a incentivar a estudar, trabalhar e/ou escrever a tese, mas acima de tudo por estar sempre presente e aturar-me, que eu bem sei que não é fácil. Agradeço também por ter sido sempre um amiga incondicional, me ter ajudado a encontrar o sentido da vida e por ser sem duvida a pessoa mais importante da minha vida. Sem ti nada disto seria possível, obrigado por tudo <3!

A Mariana que me fez acreditar no amor e também à Boneca e Hélio Jonilson Van-Dúnem que partiram cedo demais

Resumo

O Message Passing Interface (MPI) [6] é o padrão de referência para a programação de aplicações paralelas de alto desempenho em plataformas de execução que podem ir até centenas de milhares de *cores*. O MPI pode ser utilizado em programas C ou Fortran, sendo baseado no paradigma de troca de mensagens. De acordo com o paradigma *Single Program, Multiple Data* (SPMD), um único programa define o comportamento de vários processos, utilizando chamadas a primitivas MPI, como por exemplo para comunicações ponto-a-ponto ou para comunicações colectivas. O uso de MPI levanta questões de fiabilidade, uma vez que é muito fácil escrever um programa contendo um processo que bloqueie indefinidamente enquanto espera por uma mensagem, ou que o tipo e a dimensão dos dados enviados e esperados por dois processos não coincidam. Assim, não é possível garantir à partida (em tempo de compilação) uma série de propriedades fundamentais sobre a execução de um programa.

Lidar com este desafio não é de todo trivial. A verificação de programas MPI utiliza tipicamente técnicas avançadas como a verificação de modelos ou execução simbólica [9, 39]. Estas abordagens deparam-se frequentemente com o problema de escalabilidade, dado o espaço de estados do processo de verificação crescer exponencialmente com o número de processos considerados. A verificação em tempo útil pode estar limitada a menos de uma dezena de processos na verificação de aplicações *real-world* [41]. A verificação é ainda adicionalmente complicada por outros aspectos, como a existência de diversos tipos de primitivas MPI com diferentes semânticas de comunicação [39], ou a dificuldade em destrinçar o fluxo colectivo e individual de processos num único corpo comum de código [2].

A abordagem considerada para a verificação de programas MPI é baseada em *tipos de sessão multi-participante* [19]. A ideia base passa por especificar o protocolo de comunicação a ser respeitado por um conjunto de participantes que comunicam entre si trocando mensagens. A partir de um protocolo expresso desta forma, é possível extrair por sua vez o protocolo local de cada um dos participantes, segundo uma noção de projecção de comportamentos. Se para cada participante (processo) no programa MPI se verificar a aderência ao protocolo local respectivo, são garantidas propriedades como a ausência de condições de impasse e a segurança de tipos. A verificação desta aderência é feita sobre programas C que usam MPI, usando a ferramenta VCC da Microsoft Research [5].

Para codificar protocolos, foi utilizada uma linguagem formal de descrição de protocolos, apropriada à expressão dos padrões mais comuns de programas MPI. A partir de um protocolo expresso nessa linguagem. é gerado um *header* C que exprime o tipo num formato compatível com a ferramenta VCC [5]. Para além da codificação protocolo, a verificação é ainda guiada por um conjunto de contratos pré-definidos para primitivas MPI e por anotações no corpo do programa C. As anotações no programa são geradas automaticamente ou, em número tipicamente mais reduzido, introduzidas pelo programador.

Os protocolos que regem as comunicações globais num programa MPI são especificados numa linguagem de protocolos, desenhada especificamente para o efeito no contexto do projecto MULTICORE em complemento ao trabalho desta tese, e em associação um *plugin* Eclipse que verifica a boa formação dos protocolos e que gera a codificação do protocolo na linguagem VCC. As ações básicas dos protocolos descrevem no caso geral primitivas MPI, por exemplo para comunicação ponto-a-ponto ou comunicação colectiva. Os valores associados a ações podem ser do género inteiro ou vírgula flutuante, bem como vectores. Além disso, qualquer um destes géneros pode ser *refinado* com imposição de restrições aos valores dos dados. As ações básicas são compostas através de operadores de sequência, iteração, e ainda de fluxo de controlo coletivo em correspondência a escolhas ou ciclos executados de forma síncrona por todos os participantes.

A partir da especificação de um protocolo, a sua tradução no formato VCC define uma função de projecção. A função de projecção toma como argumento o índice do processo MPI, conhecido como *rank*, e devolve a codificação de um protocolo local a ser verificado para execução do participante, em linha com o enunciado pela noção de projeção [19]. Esta codificação reflecte de resto todas as características gerais da especificação do protocolo, em termos de ações básicas, o uso de tipos refinados, e operadores de composição. O processo de verificação tem por fim certificar a aderência do programa C+MPI face ao protocolo, para cada participante. Entre a inicialização e o término das comunicaçoes MPI, a verificação deve operar a redução progressiva do protocolo até ao protocolo vazio. As reduções são definidas mediante pontos de chamadas a primitivas MPI e características do fluxo de controlo de programa. Para manter o estado, a verificação manipula uma variável "fantasma" desde o ponto de entrada da programa (a função `main()`) que representa o protocolo. Para além da aderência ao protocolo, são ainda verificados aspectos complementares, como por exemplo se os dados usam regiões válidas de memória.

Esta verificação usa um corpo base de definições, a que chamamos a "MPI anotada". A MPI anotada compreende a lógica de protocolos e um corpo de contratos para um conjunto de primitivas MPI. A lógica de protocolos permite definir a estrutura de protocolos e definir as regras de redução, enquanto que os contratos das primitivas definem casos base para redução via ações de comunicação. Este corpo base pode ser importado para o contexto de verificação de um programa em particular, mediante a inclusão de um *header*

C, a versão anotada do *header* convencional da MPI (`mpi.h`) [6]. Usando esta lógica base, o programa C pode ser anotado para verificação. As anotações relacionam-se com uma diversidade de aspectos que impactam da verificação do programa, tais como o fluxo de controlo colectivo, contratos de funções, invariantes de ciclos, asserções respeitantes ao uso de memória, ou a declaração de assunções até aí implícitas no comportamento do programa.

O processo de anotação é um desafio para um programador, já que requer o domínio de uma lógica complexa de verificação. Para automatizar o processo, foi desenvolvido um anotador que gera uma parte significativa das anotações necessárias, transformando código C usando a biblioteca clang/LLVM [4]. O seu funcionamento base guia-se por anotações de alto nível para identificação de fluxo de controlo relevante e marcas de anotação simples introduzidas pelo programador, por forma a gerar em correspondência a um conjunto mais vasto e complexo de anotações. Após este processo automático, há anotações complementares que têm de ser introduzidas manualmente para a verificação bem sucedida de um programa. Estas últimas relacionam-se com aspectos diversos que ou são de inferência complexa, por exemplo o uso de memória, ou ainda não tratados na aproximação atual com um processo automatizado.

Esta aproximação à verificação de programas C+MPI foi testada com um conjunto de exemplos tirados de livros de texto. Além de demonstrar a aplicabilidade da aproximação geral considerada, é apresentada uma análise do esforço de anotação e do tempo de verificação. O esforço de anotação mede a comparação entre o número de anotações automáticas face ao número de anotações manuais, verificando-se no caso geral que o número de anotações manuais é inferior ao número das automáticas. O tempo de verificação diz respeito ao tempo de execução da ferramenta VCC para o código anotado final de um programa. A análise de escalabilidade do mesmo face a um número crescente de processos permitiu identificar casos distintos: casos em que o tempo de execução é insensível ao número de processos e outros em que este tempo cresce exponencialmente face ao número de processos.

Em conclusão, é definida uma metodologia para a verificação formal de programas MPI e demonstrada a sua aplicabilidade, combinando os paradigmas da teoria de tipos de sessão multi-participante e da verificação dedutiva de programas. Para lidar com uma maior gama de programas MPI, em particular programas *real-world*, ao longo do texto foram discutidos vários desafios que se colocam para uma evolução da metodologia, de tipo conceptual ou relacionados com a automação e escalabilidade. Para lidar com esses desafios, propõe-se na parte final um conjunto de linhas gerais para trabalho futuro.

Palavras-chave: MPI, programação paralela, verificação, tipos de sessão

Abstract

Message Passing Interface (MPI) is the de facto standard for message-based parallel applications. Two decades after the first version of its specification, MPI applications routinely execute on supercomputers and computer clusters.

MPI programs incarnate the Single Program Multiple Data (SPMD) paradigm. A single body of code, written in C or Fortran, defines the behavior of all participant processes in a program. The number of processes is defined at runtime, and any partial distinction of behavior between participants is based on the unique rank (numerical identifier) of each process. The communication between processes is defined through point-to-point or collective communication primitives. As a result, programs may easily exhibit intricate behaviors mixing collective and participant-specific flow, making it difficult to verify a priori desirable properties like absence of deadlocks or adherence to a desired communication protocol.

In line with the concern for verifiable program behavior in MPI, the theory of multi-party session types provides a framework for well-structured communication protocols by an arbitrary number of participant processes. By construction, a multi-party global protocol declares a desired interaction behavior that guarantees properties such as type safety and absence of deadlocks, and implicitly defines the individual local protocols per participant. Provided that the actual program specification of each participant conforms to the corresponding local protocol, the safety properties and the intended choreography of the global protocol are preserved by the program.

This thesis proposes the application of multi-party session type theory to C+MPI programs, coupled with the use of deductive software verification. A framework is proposed for the expression of multi-party session protocols in the MPI context and their verification using a deductive software tool. The framework also addresses concerns for automation through semi-automatic annotation of verification logic in program code. An evaluation of the applicability of the approach is conducted over a sample set of programs.

Keywords: MPI, parallel programming, formal verification, session types

Contents

List of Figures xix

List of Tables xxi

1 Introduction 1

1.1 Motivation . . . 1

1.2 Proposal . . . 1

1.3 Thesis structure . . . 2

2 Background 5

2.1 MPI . . . 5

2.2 Multi-party session types . . . 7

2.3 VCC . . . 8

2.4 Verification of MPI programs . . . 8

3 Methodology 11

3.1 Overview . . . 11

3.2 Running Examples . . . 12

3.2.1 Finite differences . . . 12

3.2.2 N-Body simulation . . . 14

3.3 Protocol specification . . . 16

3.3.1 Syntax . . . 16

3.3.2 Eclipse plugin module . . . 17

3.3.3 Examples . . . 17

3.4 Protocol verification . . . 18

3.4.1 VCC syntax . . . 19

3.4.2 The projection function . . . 20

3.4.3 The verification process . . . 22

3.4.4 MPI function contracts . . . 22

3.4.5 Collective loops and choices . . . 23

3.4.6 For-each protocols . . . 24

3.5 Program annotation . . . 25

3.5.1 Protocol-related annotations . . . 25
3.5.2 Automated generation of annotations . . . 26
3.5.3 Complementary manual annotations . . . 26
3.5.4 Examples . . . 28

4 Design and Implementation **33**
4.1 Annotated MPI library . . . 33
4.1.1 Mock MPI header . . . 33
4.1.2 Parameterization . . . 34
4.1.3 Refinement types . . . 34
4.1.4 Session type representation . . . 35
4.1.5 Protocol projection . . . 37
4.1.6 Structural congruence . . . 37
4.1.7 Protocol reductions . . . 37
4.2 Program annotator . . . 39
4.2.1 The Clang/LLVM framework . . . 39
4.2.2 Implementation . . . 40

5 Evaluation **45**
5.1 Sample MPI programs . . . 45
5.2 Annotation effort . . . 45
5.2.1 Annotation process . . . 45
5.2.2 Results . . . 47
5.3 Verification time . . . 47
5.3.1 Test setup and execution . . . 47
5.3.2 Results . . . 48

6 Conclusion **51**
6.1 Summary of contributions . . . 51
6.2 Future work . . . 51

Appendixes **54**

A Evaluation examples **55**
A.1 Finite differences . . . 55
A.1.1 Protocol projection . . . 55
A.1.2 Program code . . . 55
A.2 N-body simulation . . . 60
A.2.1 Protocol projection . . . 60
A.2.2 Program code . . . 60
A.3 Parallel Jacobi iteration . . . 67

A.3.1 Protocol projection 67
A.3.2 Program code 68
A.4 Parallel dot product 76
A.4.1 Protocol projection 76
A.4.2 Program code 76

Bibliography **88**

List of Figures

2.1 Example MPI program 6
2.2 Multiparty session type [19] 7

3.1 Verification approach 11
3.2 Finite differences program 13
3.3 N-body simulation program 15
3.4 Protocol communication grammar 16
3.5 Eclipse plugin for protocol specification 17
3.6 Protocol for the Finite Differences program 18
3.7 Protocol for the N-Body simulation program 19
3.8 VCC projection function for the finite differences example 20
3.9 VCC projection function for the N-body example 21
3.10 Annotated finite differences program 29
3.11 Annotated N-body simulation program 30

4.1 Overall parameterization 34
4.2 Support for refinement types 34
4.3 Session type representation 36
4.4 Message projections 36
4.5 Structural congruence predicate 37
4.6 Protocol reduction logic 38
4.7 Example use of a Clang/LLVM cursor 39
4.8 Annotator tool — overall operation 40
4.9 Annotator tool — generation of annotations 41

List of Tables

3.1 Automatic generation of annotations 27

4.1 Annotated MPI — supported communication primitives 33

5.1 Annotation effort 47
5.2 VCC execution time (seconds) 48

Chapter 1

Introduction

1.1 Motivation

Message Passing Interface (MPI) [6] is a standard for the programming of high performance parallel applications that can be deployed on execution platforms with hundreds of thousands of cores. Based on the message-passing paradigm, the MPI standard has C and Fortran bindings. These programs are encoded according to the *Single Program, Multiple Data* (SPMD) paradigm, i.e., a single program defines the behavior of the various processes at runtime. Any partial distinction of behavior between participants is based on the unique rank (a numerical identifier) of each process.

It is quite easy to write a MPI program that leads to execution errors such as blocked processes waiting for a message, data races between the process and the MPI environment, type/size between message senders and receivers, or adherence to a desired communication protocol. No guarantees exist in advance, at compilation time, for key fundamental properties regarding the execution of a program, a well-acknowledged problem by active research [10].

Several methodologies are employed in the formal verification of MPI programs, such as model checking and symbolic execution [9, 39, 42]. These approaches typically face a scalability problem, given that the state space for verification grows exponentially with the number of processes. Verifying real-world applications may restrict that state space to the interactions to no more than a few processes, as in [41]. The verification is further complicated by various additional aspects such as the existence of diverse kinds of MPI primitives with different communication semantics [39], or the difficulty in distinguishing the collective and individual behavior of processes in a single program [2].

1.2 Proposal

This thesis presents an approach for the formal verification of MPI programs based on multi-party session types [19]. The theory considers the specification of a global interac-

tion protocol among multiple participants, from which we can derive an endpoint protocol for each individual participant. A well-formed protocol is by construction type-safe and contains no deadlocks [19]. The same properties are preserved for a program that complies with that protocol, e.g., as in Session-C [31]. The idea considered in this thesis is to apply the same reasoning to MPI programs.

To verify the compliance of a MPI program against a given protocol specification, deductive software verification is employed over MPI programs written in C [5, 35, 20]. The approach requires the annotation of the target program with a verification logic for properties of interest. Thus, in the case at stake, annotations must allow for the expression of the target protocol and the necessary logic for verifying compliance of the program with the protocol. For this purpose the VCC verifier is employed [5].

The overall methodology comprises the distinct steps of protocol specification and protocol verification. The focus of this thesis is on protocol verification, although the overall traits of protocol specification are also described. A protocol specification is encoded in a formal language that is based on multi-party session types but also captures some of the main traits of MPI programs, such as the diversity of (point-to-point or collective) communication primitives and collective control flow. A well-formed protocol specification is converted to the form of the VCC logic automatically, and the latter is used in the verification process. Through complementary annotation of the target program, the compliance of the target program against the specified protocol can be verified.

The implementation of the verification framework comprises some base verification logic and a tool for semi-automated annotation of programs. The base verification logic defines how protocols can be expressed and matched, plus function contracts for a subset of MPI primitives which are called the "annotated MPI". In order to automate the annotation process as much as possible, a significant part of program annotations is automatically generated by an annotator tool.

1.3 Thesis structure

The remainder of this thesis is structured as follows:

- **Chapter 2. Background** — Some background context is provided on MPI, multiparty session type theory and tools, the VCC software verifier, and related work in the formal verification of MPI programs.

- **Chapter 3. Methodology**— In this core chapter, the approach for formal verification of MPI programs is described. The various aspects of protocol specification, protocol translation to verification logic, program verification, and semi-automated program annotation are covered. Two MPI programs are used as running examples in the discussion.

- **Chapter 4. Design and Implementation** — The design and implementation of the verification framework is explained, regarding the base VCC logic for protocol verification and the tool for semi-automated generation of program annotations.
- **Chapter 5. Evaluation** — An evaluation is made considering a sample set of MPI programs. Results are presented and discussed regarding the effort for annotating programs and the execution time of the verification process.
- **Chapter 6. Conclusion** — The final chapter makes a summary of the contributions of this thesis and identifies directions for future work.
- **Appendix A. Evaluation examples** — The evaluation examples of Chapter 5 are provided in full, regarding the protocol specification in VCC format, plus the annotated source code. In complement and due to space restraints, the source code of the "annotated MPI" and the annotator tool are provided as an annex to this thesis in the form of a ZIP archive.

Chapter 2

Background

This chapter presents the background context of this thesis, comprising a general description of MPI (Section 2.1), the multi-party session type methodology (Section 2.2), the use of VCC (Section 2.3), and a brief survey of related work on formal verification of MPI programs (Section 2.4).

2.1 MPI

MPI is a library specification targeting the development of communication-intensive parallel applications [6], which became the de facto standard for message-based parallel computing. The MPI standard is large, at this point comprising such diverse aspects such as parallel I/O, one-sided communication, process management, beyond an enormous variety of features related to message-passing itself. We concentrate on illustrating the MPI features that will be at stake in this thesis.

MPI programs can be written in C or Fortran, employing the SPMD paradigm. A single program body defines the behavior of the various processes at runtime, which exchange point-to-point messages or engage in collective communication. Any possible distinction of behavior for a process or group processes is done through the process rank, an unique integer identifier of each process at runtime.

An example MPI program in Figure 2.1 illustrates these core traits. It is a toy program to calculate an approximation of π through a numerical integration, adapted from [13]. The process with rank 0 reads the number of intervals for the required numerical integration (line 12) and sends that value to all other processes (l. 13-16) using point-to-point messages (`MPI_Send`, l. 15). In correspondence, all other processes wait to receive the data (using `MPI_Recv`, l. 19). The computation then takes place locally at each process (l. 22–28) and is followed a collective reduction operation (`MPI_Reduce`, l. 31). The reduction operation calculates the sum of all local values computed per process, and yields that result to process 0 only.

Observe that the program in Figure 2.1 can easily be changed to induce erroneous

```
1  int main(int argc, char **argv) {
2    int n, rank, procs, i;
3    double PI25DT = 3.141592653589793238462643;
4    double mypi, pi, h, sum, x;
5    MPI_Status status;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &procs);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10   if (rank == 0) {
11     printf("Enter the number of intervals: ");
12     scanf("%d",&n);
13     for (i = 1; i < procs; i++) {
14       // Send message to every other process
15       MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
16     }
17   } else{
18     // Receive a message from process 0
19     MPI_Recv(&n, 1, MPI_INT, 0 , 0, MPI_COMM_WORLD, &status);
20   }
21   // Computation
22   h = 1.0 / (double) n;
23   sum = 0.0;
24   for (i = rank + 1; i <= n; i += procs) {
25     x = h * ((double)i - 0.5);
26     sum += (4.0 / (1.0 + x*x));
27   }
28   mypi = h * sum;
29   // Reduction using sum of mypi from all processes
30   // Value becomes available at process 0.
31   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
32   if (rank == 0) {
33     printf("pi is approximately %.16f, Error is %.16f\n",
34             pi, fabs(pi - PI25DT));
35   }
36   MPI_Finalize();
37   return 0;
38 }
```

Figure 2.1: Example MPI program

behavior. For instance, if we remove the call to `MPI_Recv` (l. 19), process 0 may hang on the first call to `MPI_Send` (l. 15), since the programmer should assume no buffering for `MPI_Send` by the implementation [6]. Process 0 would also hang if instead lines 31 and 32 were swapped, i.e., if the reduction operation (`MPI_Reduce`) was initiated by process 0 alone. All processes need to eventually engage in the reduction, otherwise deadlock will occur.

2.2 Multi-party session types

Session types are a representation of a system that ensures the communication is respecting a particular protocol associated with a channel, first introduced in [18]. The structuring concept is that of a *session*, an high-level specification for the disciplined interaction between participants in a concurrent setting. Multi-party session types [19] define a theory for an arbitrary, parametric number of participants, extending previous work that mostly focussed on binary (two-party) session types.

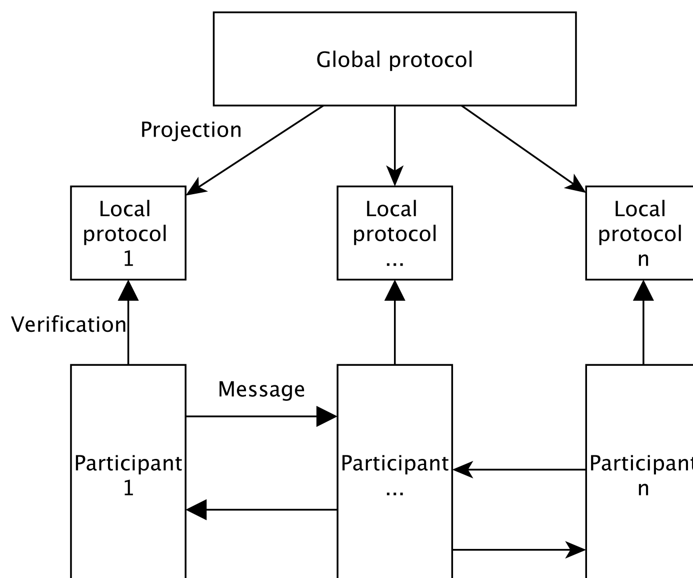

Figure 2.2: Multiparty session type [19]

The intuition for multi-party session types is depicted in Figure 2.2. On top, a global protocol specifies the intended interaction of participant programs that communicate by message-passing, shown at bottom. The global protocol implicitly defines a local protocol per each participant, shown at middle, through a notion of protocol projection. A local protocol respects to the role of a participant in the global choreography. A well-formed protocol verifies by construction type safety, communication safety and deadlock freedom [19]. The same properties are preserved for participant programs, provided the programs are certified as compliant with the corresponding local protocol. Software frameworks like Scribble [17], Session Java [32], or Session C [31], are example instantiations of the theory.

Considering the context of MPI programs, there are a few key aspects that challenge the adoption of the multi-party session type approach. The conformance of participant programs against local protocols cannot be checked modularly per participant, given the SPMD paradigm of MPI. In contrast, participants in Session-C [31] or Session-Java [32] are specified separately. MPI programs also make use of collective operations and exhibit

a collective flow which are not governed by definition by a specific participant, e.g., as defined by the directed choice and iteration operators in [17, 31, 32]. Finally, MPI programs employ synchronized communication, whereas multi-party session type theory and its instantiations assume asynchronous, buffered semantics for communication, which tend to impose less restrictions on the order of messages/operations per each participant.

2.3 VCC

VCC [5] is a deductive verifier for C programs. To perform the verification of a program using VCC, the C code must be annotated with several items, such as logical assertions, function contracts, data and loop invariants and ghost code, e.g., as in the Java Modelling Language (JML) [23] and associated verification tools. The format of annotations and associated verification logic is exposed in detail in the next two chapters.

The tool has been developed at Microsoft Research and applied to real-world projects, e.g., Hyper-V [28] comprising 60 thousand lines of operating system-level C and assembly code. VCC takes an annotated program and tries to certify the correctness of the program with respect to the annotations. For that purpose, VCC translates the code and annotations into the intermediate language of the Boogie tool [27], in turn then fed to the Z3 [30] a satisfiability modulo theories (SMT) solver.

There are other similar tools which were considered for use in this work, like Frama-C [35] or Verifast [20]. An analysis concluded that VCC had the more adequate and comprehensive set of features, such as the possibility of abstract ghost datatypes, user-specified theories, or built-in pointer validity and type-checking [5].

2.4 Verification of MPI programs

The current state-of-the-art in MPI program verification has been covered recently in [9]. Overall, the target properties of verification can be diverse, ranging from simple verifications of MPI primitive call arguments [11], traditional interaction properties like deadlock detection [12, 40, 44], up to functional equivalence to serial programs [42]. So are the adopted methodologies, ranging from "traditional" static and dynamic program analysis or a combination of both [2, 43, 44, 12, 15], to model checking and symbolic execution [40, 42, 41]. Some reference of key work in this research area is provided next.

MPI-Check MPI-CHECK [11, 12] is a tool to check Fortran 90 MPI [6] programs using compile-time and runtime checks of MPI programs. The focus of the tool is on debugging and requires explicit instrumentation by the programmer. Compile-time checks comprise the validity of calls to MPI primitives and some limited properties of their respective arguments. Runtime checks comprise the validity of runtime arguments, memory safety,

and deadlock detection. For deadlock detection, MPI-CHECK maintains a dependency graph of MPI calls at runtime.

DAMPI, P^{N}MPI and MUST DAMPI [44] is a dynamic analyzer for MPI [6] programs for detection of deadlocks and resource-leaks in real applications. It operates in automated manner without requiring program changes. The automation comes from using the P^{N}MPI [38] interposition layer, that intercepts MPI calls and and calls DAMPI monitoring routines. MUST [15], preceded by [22] and Umpire [21], uses a similar approach to DAMPI. Again, P^{N}MPI [38] is employed and the focus of runtime checks also concerns resource usage and interaction properties like deadlock detection or message loss.

ISP and TASS ISP [43] is a deadlock detection tool which uses a scheduler to explore all possible thread interleavings of an execution, combining model-checking and symbolic execution related techniques. TASS [42, 40] also combines symbolic execution and model checking. Beyond deadlocks and traditional safety properties, TASS is able to verify user-specified assertions for the interaction behavior of the program, so-called collective assertions. Finally, TASS is able to verify functional equivalence between a MPI program and a sequential program.

Parallel data-flow analysis Parallel data-flow analysis is a static analysis technique applied in [2]. The work focuses on send-receive matching in MPI source code, which helps identify message leaks and communication mismatch, by constructing a parallel control-flow graph by simple symbolic analysis on the control-flow graph of the MPI program. In [37] the authors discuss extending the technique by combining static and dynamic analysis to improve precision of the data-flow analysis.

Chapter 3

Methodology

This chapter describes the methodology for verifying C+MPI programs against communication protocols specified using multi-party session types [19]. The chapter begins with an overview of the approach (Section 3.1) and a discussion of two running examples (Section 3.2). The specification of protocols using multi-party session types (Section 3.3) and the verification of C+MPI programs against these protocols (Section 3.4) is then explained. The chapter ends with the description of the process of annotating programs for verification (Section 3.5).

3.1 Overview

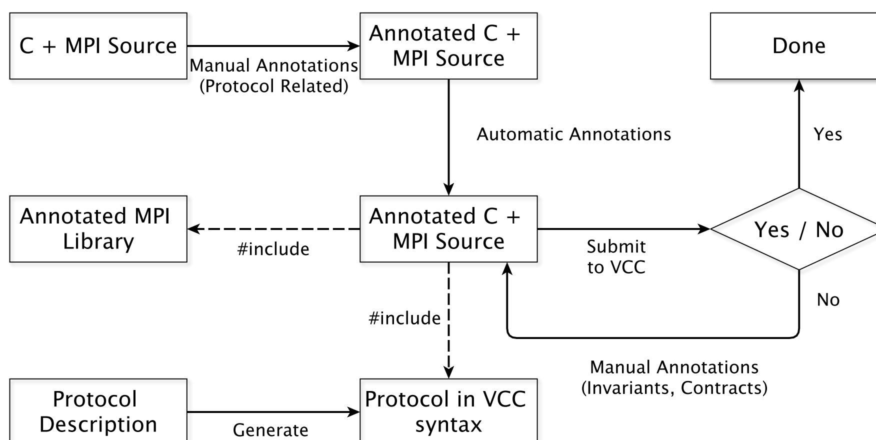

Figure 3.1: Verification approach

The overall approach for the verification of C+MPI programs is illustrated in Figure 3.1. Given a C+MPI program and a protocol description, the goal is to verify compliance of the program against the protocol.

A protocol is specified in a formal language that is based on multi-party session types, and recognized by a plugin module embedded in the Eclipse [1] platform. From a well-formed protocol description, the same module is able to generate a C header file encoding

the protocol in a format understood by the VCC verifier [5]. The C program must include this header, along with another one, a mock MPI header (`mpi.h`) that imports the base logic for verification and MPI function contracts using that base logic.

For verification, a C+MPI program must then be annotated with complementary verification logic. This takes place in semi-automated manner with the help of an automatic annotator tool and comprising different levels of annotation. High-level annotation marks are introduced by the programmer to guide the annotator in generating a larger set of low-level annotations. These high-level marks relate to protocol-related features that the annotator is unable to infer on its own. The output of the annotator is not usually sufficient for VCC to assert that the program complies with the protocol, and manually introduced annotations are required.

3.2 Running Examples

Two C+MPI programs are considered as running examples: a program that implements a one-dimensional finite differences algorithm, and an N-body simulation algorithm.

3.2.1 Finite differences

The finite differences program concerns an iterative algorithm described in [7]. Considering an initial vector X^0, successive approximations of a a possible problem solution are calculated, X^1, X^2, ..., X^n, until a condition of numerical convergence is met or a maximum number of iterations has been reached.

Figure 3.2 lists the main code of the C program, also adapted from [7]. The process number, `procs`, and the `rank` of each process, are initialy obtained using the MPI primitives `MPI_Comm_size` and `MPI_Comm_rank` (lines 5 and 6). In sequence, the participant with rank `0` begins by reading the initial vector X^0 (lines 9–10) and distributing the vector through all participants (line 11), using `MPI_Scatter`. Each participant is responsible for calculating part of the solution, with the same vector length per participant.

Following a ring topology according to the rank of the participant, the program executes a loop (lines 16–37), exchanging point-to-point messages (`MPI_Send`, `MPI_Recv`) between each participant and its left and right neighbours. The purpose of these exchanges is to distribute the border values due to each participant, which are necessary to local computations by the neighbour processes. To avoid deadlocks, a different order of calls to `MPI_Send` and `MPI_Recv` is required for participant `0` (lines 19-22), participant `procs-1` (lines 23-27) and for other participants (lines 28-32). This is necessary since the behavior of `MPI_Send` and `MPI_Recv` must be assumed as *synchronous* [6], meaning that no buffering exists and that a call to `MPI_Send` by one process will not return before a corresponding `MPI_Recv` call has been issued by another process. For example, if two

```
1  int main(int argc, char** argv) {
2    int procs;              // Number of processes
3    int rank;               // Process rank
4    MPI_Init(&argc, &argv);
5    MPI_Comm_size(MPI_COMM_WORLD, &procs);
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    ...
8    int psize = atoi(argv[1]);            // Global problem size
9    if (rank == 0)
10      read_vector(work, lsize * procs);
11   MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
12   int left  = (procs + rank - 1) % procs;  // Left neighbour
13   int right = (rank + 1) % procs;          // Right neighbour
14   int iter = 0;
15   // Loop until minimum differences converged or max iterations attained
16   while (!converged(globalerr) && iter < MAX_ITER)) {
17     ...
18     if (rank == 0) {
19       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
20       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
21       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
22       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
23     } else if (rank == procs - 1) {
24       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
25       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
26       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
27       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
28     } else {
29       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
30       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
31       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
32       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status)
33     }
34     ...
35     MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
36     ...
37   }
38   ...
39   if (converged(globalerr)) { // Gather solution at rank 0
40     MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
41     ...
42   }
43   ...
44   MPI_Finalize();
45   return 0;
46 }
```

Figure 3.2: Finite differences program

processes are assumed and lines 20 and 21 are exchanged, a deadlock will result , since participants `0` and `1` would both be waiting for the reception of a message from each other.

After exchanging messages, and still in the cycle, the global error is calculated with a reduction operation and is then disseminated to all participants using `MPI_Allreduce` (line 35). The loop terminates when there is a convergence condition or after a predefined number of iterations. When the loop terminates, and in case of convergence, the participant `0` aggregates the final solution, receiving from each participant a part of the array by using `MPI_Gather` (lines 39 and 40). Otherwise, if the loop terminates without reaching convergence, no further exchange of messages takes place.

3.2.2 N-Body simulation

An N-body simulation computes the trajectories of bodies/particles that interact through gravitational forces at discrete time intervals. The N-body simulation program considered here is adapted from [13].

As in the finite differences example, each process initiates retrieving the number of processes and the process' rank, `procs` and `rank` (lines 18–19). Similarly, the example uses a ring topology: each process locally computes the ranks of its left and right neighbours (lines 22–23) for subsequent point-to-point message exchanges.

The initial message exchange is an all-gather operation, so that all processes get the number of particles handled by all other processes (line 29). Then all processes locally calculate the total number of particles (`totpart`, line 31). The outcome is as if each process in the group had executed a `MPI_Send` primitive to all processes, including itself, and executed a `MPI_Recv` primitive from all processes, again including itself.

The program's main loop (lines 36–60) will execute the n-body simulation for a fixed number of iterations, as defined by the `cnt` variable. An inner loop is defined (lines 40–57) for a transmission of data and computation of forces. The transmission of data will occur from the process `rank` for its right neighbour, receiving the data from its left neighbour. This communication behavior will be repeated `procs - 2` times (line 42). In order to avoid deadlocks, a different behavior is needed concerning the order of use of the `MPI_Send` and `MPI_Recv` primitives for the participant with `rank` even (lines 44-47) and for other participants (lines 49-52).

These communications are necessary to share the particles information and locally compute the forces. After the **`for`** loop (line 40–57) and the computation step, the changes in position are calculated through the `ComputeNewPos` procedure (line 59). This function computes the new positions for the local particles and internally adjusts the global timestep with a global reduction (`MPI_AllReduce`) operation.

```
int main( int argc, char *argv[])
{
    float      particles[MAX_PARTICLES * 4];   /* Particles on ALL nodes */
    float      pv[MAX_PARTICLES * 6];          /* Particle velocity */
    float      sendbuf[MAX_PARTICLES * 4],     /* Pipeline buffers */
          recvbuf[MAX_PARTICLES * 4];
    int         counts[MAX_P],              /* Number on each processor */
          displs[MAX_P];              /* Offsets into particles */
    int         rank, procs, npart, i, j,
          offset;                     /* location of local particles */
    int         totpart,                    /* total number of particles */
          cnt;                        /* number of times in loop */
    float      sim_t;                      /* Simulation time */
    int         pipe, left, right;
    MPI_Status  statuses[2];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &procs );

    /* Get the best ring in the topology */
    left = (procs + rank - 1) % procs;
    right = (rank + 1) % procs;
    ...
    npart = atoi(argv[1]);
    ...
    npart = npart / procs;
   /* Get the sizes */
    MPI_Allgather( &npart, 1, MPI_INT, counts, 1, MPI_INT, MPI_COMM_WORLD );
    ... /* calculate displacements */
    totpart = displs[procs-1] + counts[procs-1];
    /* Generate the initial values */
    ...
    ctn = 10;
    sim_t = 0.0;
    while (cnt > 0)
    {
       /* Load the initial sendbuffer */
       ...
       for (pipe=0; pipe<procs; pipe++)
       {
          if (pipe != procs-1) {
             if (rank == 0) {
                MPI_Send( sendbuf, npart * 4, MPI_FLOAT, right, 0,
                                   MPI_COMM_WORLD);
                MPI_Recv( recvbuf, npart * 4, MPI_FLOAT, left,  0,
                            MPI_COMM_WORLD, &statuses[0] );
             } else {
               MPI_Recv( recvbuf, npart * 4, MPI_FLOAT, left,  0,
                                  MPI_COMM_WORLD, &statuses[0] );
               MPI_Send( sendbuf, npart * 4, MPI_FLOAT, right, 0,
                            MPI_COMM_WORLD);
            }
         }
         /* Compute forces (2D only) */
          ...
      }
      /* Once we have the forces, we compute the changes in position */
      sim_t += ComputeNewPos( particles, pv, npart, max_f, MPI_COMM_WORLD);
   }
   ...
   MPI_Finalize();
   return 0;
}

double ComputeNewPos( Particle particles, ParticleV pv, int npart,
          double max_f, MPI_Comm commring)
{
   int i;
   float a0, a1, a2;
   static float dt_old = 0.001, dt = 0.001;
   float dt_est, new_dt, dt_new;
   ...
   /* Re-Calculate the time-step control */
   ...
   /* Reduce to the minimum time-step control in the participants */
   MPI_Allreduce( &dt_est, &dt_new, 1, MPI_FLOAT, MPI_MIN, commring);
   ...
   return dt_old;
}
```

Figure 3.3: N-body simulation program

3.3 Protocol specification

A specific language was designed to describe protocols for global communications in MPI programs. This base been done in the context of the MULTICORE research project, in a complementary line of work to that of this thesis.

3.3.1 Syntax

The base syntax of the language for protocol specification is defined by the grammar shown in Figure 3.4.

T	$::=$ skip	terminated protocol
	$\mid$ $\mathsf{message}\ i\ i\ x\colon D$	point-to-point comm.
	$\mid$ $\mathsf{broadcast}\ i\ x\colon D \mid \mathsf{scatter}\ i\ x\colon D \mid \ldots$	collective comm.
	$\mid$ $T;T$	sequence
	$\mid$ $\mathsf{foreach}\ x\colon i..i\ \mathsf{do}\ T$	repetition
	$\mid$ $\mathsf{loop}\ T$	collective loop
	$\mid$ $\mathsf{choice}\ T\ \mathsf{or}\ T$	collective choice
	$\mid$ $\mathsf{val}\ x\colon D$	variable
D	$::=$ $\mathsf{int} \mid \mathsf{float} \mid D[i] \mid \{x\colon D \mid p\} \mid$ ...	index types
i	$::=$ $x \mid n \mid i+i \mid \mathsf{max}(i,i) \mid \mathsf{length}(i) \mid i[i] \mid \ldots$	index terms
p	$::=$ $\mathsf{true} \mid i \leq i \mid p\ \mathsf{and}\ p \mid a(i,\ldots i) \mid \ldots$	index propositions

Figure 3.4: Protocol communication grammar

The simplest syntactic term is `skip`, representing an empty (or terminated) protocol. A set of communication operators are then defined in correspondence to MPI point-to-point messaging, the **`message`** operator, or collective communication, like the **`broadcast`** and **`scatter`** operators and a few others.

Protocols can be composed sequentially using the base sequence operator `;` or the **`foreach`** repetition operator. Collective control flow, collective loops and choices, are defined by **`loop`** and **`choice`** operators, respectively. These collective flow operators represent consensual agreement between *all* participants. Finally, the **`val`** operator represents an abstract value computed at runtime.

Values associated to message exchange can be of primitive type, like **`int`** or **`float`**, or vectors. Furthermore, types can be refined through the imposition of constraints over the domain of values, e.g., `{ x:int | x % 5 == 0}` stands for the refined type with values ranging over the domain of integer numbers that are multiples of 5.

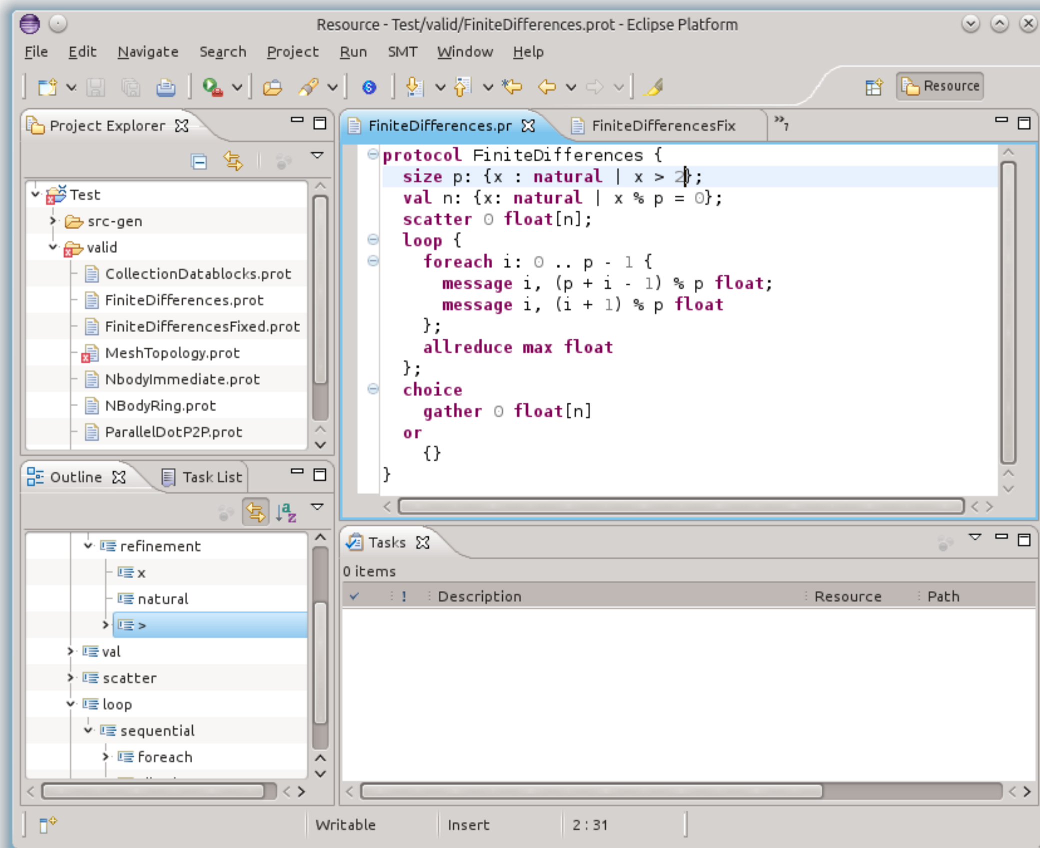

Figure 3.5: Eclipse plugin for protocol specification

3.3.2 Eclipse plugin module

The Eclipse plugin module that recognizes the protocol language has been developed by César Santos. It is implemented using the Xtext framework [46] and the SMTLib [3] library. A screenshot of the tool is shown in Figure 3.5.

3.3.3 Examples

Finite Differences protocol The protocol for the finite differences program is given in Figure 3.6. Line 2 introduces the number of processes over variable `p` and line 3 introduces the problem size variable `n`. Both are runtime parameters, that are matched, respectively, by a call to `MPI_Comm_size` and by an abstract value in the C+MPI program flow (`n`). The remaining structure corresponds to the pattern of communication in the C+MPI program. An initial data scatter operation (line 4) operation is specified, followed by a collective loop (lines 5–11) where all processes exchange messages with their left and right neighbors (lines 6–9) before engaging collectively in a global reduction (line 10). Lastly, participant 0 gathers the data from all participants in a collective choice, corre-

```
1  protocol FiniteDifferences {
2    size p: positive;                  // process number
3    val n: {x: natural | x % p == 0}; // problem size
4    scatter 0 float[n];
5    loop {
6      foreach i: 0 .. p - 1 {
7        message i, (i + 1) % p float;
8        message i, (p + i - 1) % p float
9      };
10     allreduce max float
11   };
12   choice
13     gather 0 float[n]
14   or
15     {}
16 }
```

Figure 3.6: Protocol for the Finite Differences program

sponding to the case of numerical convergence of the finite differences algorithm (recall that the final solution is discarded otherwise).

N-Body pipeline protocol The protocol for the N-body simulation program is presented in Figure 3.7. We omit an explanation, given that the syntactic traits are similar to those in the finite differences example, even if the example is slightly more intricate.

3.4 Protocol verification

A protocol specification is translated to VCC verification logic and the target program for verification must be annotated. The following aspects are at stake, described in this section:

1. A protocol specification is translated to a C header file that encodes the protocol. This header file must be included in the main C file of the program to verify;

2. The frogman's main C file must also include the standard MPI header `mpi.h`. In reality, the MPI header contains a mock definition of annotated MPI primitives with verification contracts;

3. The programs's C code in turn must be additionally modified with the insertion of annotations that allow the verification of the program against the protocol specification;

```
1  protocol nbodypipe  {
2    size p : positive; // number process
3    val n  : positive; // problem size
4    allgather int ;
5    loop {
6      loop {
7        choice {
8          foreach i: 0..p-1 {
9            message i, (i+1)%p float[n/p];
10         };
11       }
12       or
13       {}
14     };
15     allreduce min float;
16   }
17 }
```

Figure 3.7: Protocol for the N-Body simulation program

4. The generation of a significant part of program annotations is automated, but some still need to be inserted by the user. The manual annotations are either simple "hints" that guide the automatic generation of a larger set of annotations, or required by VCC to complement program verification.

3.4.1 VCC syntax

We begin by introducing some key aspects of the syntax used in the VCC verification logic. In summary, these are as follows:

- All annotations are enclosed in blocks of the form `_(annotation block)`. Annotated code may be processed normally by a C compiler, filtering out any annotations, using the following C preprocessor macro:

  ```
  #define _(a)  /* blank macro expansion */
  ```

- Logical clauses of the form `requires condition` and `ensures condition` are used to specify pre and post-conditions in a function contract, respectively, as usual in design-by-contract frameworks (e.g., JML [23]]). Likewise, `invariant condition` expresses an invariant over data structures or loops, the `ghost` keyword is used in association to "ghost" definitions, and the `pure` keyword indicates that a (C or ghost) function has no semantically observable side effects.

- A definition of the form `(\lambda type x; f[x])` encodes an anonymous `type` domain function.

```
1  _(ghost _(pure) \SessionType ftype (\integer rank)
2   _(ensures \result ==
3    seq(
4     action(size(), intRef(\lambda \integer y; y>0, 1)),
5     abs(body(\lambda \integer p;
6      seq(
7       action(val(), intRef(\lambda \integer x; x>0 && x%p==0, 1)),
8       abs(body(\lambda \integer n;
9        seq(
10        action(scatter(0), floatRef(\lambda float v; \true, n)),
11       seq(
12        loop(
13         seq(
14          foreach(0, p-1,
15           body(\lambda \integer i;
16            seq(
17             message(i, (p+i-1)%p,
18                       floatRef(\lambda float v; \true, 1))[rank],
19             message(i, (i+1)%p,
20                       floatRef(\lambda float v; \true, 1))[rank])
                            )),
21          action(allreduce(MPI_MAX),floatRef(\lambda float v; \
               true, 1)))),
22        choice(
23         action(gather(0), floatRef(\lambda float v; \true, n)),
24         skip()
25        )))))))))
26   );
27 )
```

Figure 3.8: VCC projection function for the finite differences example

3.4.2 The projection function

A protocol specification is translated to a C header file, containing a VCC ghost function with the following signature:

```
\SessionType ftype(\integer rank)
```

We call it the protocol projection function. Given a process rank as argument, it defines a local (endpoint) view of the protocol for the given rank of VCC type `\SessionType`. In line with the notion of projection in the theory of multi-party session types [19], the projection is the local protocol for verification per each participant in the global protocol.

The projection functions for the two running examples are provided in Figure 3.8 and Figure 3.9. In spite of a different syntactic context and other details we explain in Chapter 4, we can in any case observe that the projection functions resemble the original

```
1  _(ghost _(pure) \SessionType ftype (\integer rank)
2    _(ensures \result ==
3      seq(action(size(), intRef(\lambda \integer procs; procs > 0,
          1)),
4      seq(abs(body(\lambda \integer procs;
5          seq(action(val(), intRef(\lambda \integer size; size > 0,
              1)),
6          seq(abs(body(\lambda \integer size;
7              seq(action(allgather(),anyInt(procs)),
8        seq(loop(
9                  seq(loop(
10                     seq(choice(
11                         seq(foreach(0, procs-1,
12                             body(\lambda \integer i;
13                                 seq(message(i, (i+1)%procs,
                                      anyFloat(size * 4))[rank],
14                                 skip()))),
15                         skip()),
16                         skip()),
17                     skip())),
18                 seq(action(allreduce(MPI_MIN),anyFloat(1)),
19                 skip())))),
20             skip()))))),
21         skip()))))),
22     skip()))
23    );
24 )
```

Figure 3.9: VCC projection function for the N-body example

protocol specifications (Figures 3.6 and 3.7). A crucial difference exists though: the projected protocol will differ per participant according to point-to-point message exchanges. We have that the `\SessionType` term returned by `ftype` takes the `rank` parameter in consideration. A term of the form **`message`**`(from,to,dt)[rank]`, defining a point-to-point message exchange, evaluates (projects) to:

- `action(send(to), dt)` if `rank` equals `from`, encoding a message sent (from `rank`) to participant `to`;
- `action(recv(from), dt))` if `rank` equals `to`, encoding a message received (by `rank`) from process `from`;
- and `skip()` otherwise, i.e., no message exchange.

3.4.3 The verification process

The verification of a program against a projected protocol analyses the program's control flow between the initialization and termination points of the MPI library, respectively calls to `MPI_Init` and `MPI_Finalize`. The protocol is obtained through the projection function (`ftype`, described earlier), upon initialization. It must then be progressively reduced such that in the end the protocol is congruent to to `skip()`. For example, an empty collective loop —`loop{}`—or a *foreach* without any possible unfolding —`foreach(0,-1,...)`— are both congruent to `skip()`). To verification state manipulates a **ghost** variable of `\SessionType`, declared in the program's `main()` function. The contracts of `MPI_Init` and `MPI_Finalize` summarize the overall logic of verification:

```
int MPI_Init(... _(ghost ampi_glue_t gd) _(out \SessionType
    typeOut))
  _(ensures typeOut == ftype(gd->rank))
  ...
int MPI_Finalize(... _(ghost \SessionType typeIn))
  _(ensures congruence(typeIn, skip())
  ...
```

In the contract of `MPI_Init`, the `gd` term encodes the process rank within the verification logic, as shown, which is then used in the invocation `ftype`. In the contract of `MPI_Finalize`, the `congruence` predicate expresses the congruence between two instances of `\SessionType`.

3.4.4 MPI function contracts

Between initialization and termination, the verification must deal with the progressive reduction of the protocol, including calls in the program to MPI communication primitives. The contracts of the latter define the base cases for reduction. Typically, the reduction defined by a primitive works by extracting (requiring) a communication action prefix from the input protocol, and yielding (ensuring) as output a protocol continuation. Other complementary aspects are also at stake, such as verifying that buffer arguments supplied to MPI primitives are valid memory regions.

We illustrate these aspects with the following fragment of the contract for `MPI_Send`:

```
int _MPI_Send(void *buf, int count, MPI_Datatype datatype, ...
                _(ghost \SessionType type_in)
                _(out \SessionType type_out))
 _(requires actionType(first(type_in)) == send(dest))
 _(requires actionLength(first(type_in)) == count)
 _(requires refTypeCheck(refType(first(type_in)), buf, count))
 _(requires datatype == MPI_INT
      ==> \thread_local_array ((int *) buf, count))
```

```
_(requires datatype == MPI_FLOAT
     ==>\thread_local_array((float *)but, count))
...
_(ensures  type_out == next(type_in))
...
```

The contract above stipulates (in the order shown) that:

- The first action of the input protocol is `send(dest)`, where `dest`, as shown in the function signature, is the function parameter that identifies the destination rank;
- The dimension of the array in the action is equal to the `count` parameter;
- The data to be transmitted, each element of the `buf` argument, must verify the restrictions of type refinement and is a valid memory region;
- Finally, as a post-condition, the protocol to check afterwards is the continuation of the starting type.

Note that the contract defines `_MPI_Send`, not `MPI_Send`. The latter is in reality defined as a C processor macro that expands to a call to `_MPI_Send` with the necessary complementary ghost arguments:

```
#define MPI_Send(a1, a2, a3, a4, a5, a6) \
  _MPI_Send(a1, a2, a3, a4, a5, a6 _(__ampi_glue_data__) _(ghost
     _type) _(out _type))
```

In the current verification framework, we define contracts for the point-to-point MPI operations `MPI_Send` and `MPI_Recv` plus some of the most common collective collective primitives, e.g., `MPI_Bcast`, `MPI_Allreduce` or `MPI_Scatter`; The set of annotated operations is detailed in Chapter 4. Some common and important features found in MPI programs [6] are not yet supported, including non-blocking operations (e.g., `MPI_Isend` and `MPI_Irecv`) or communicator creation (only the global top-level communicator `MPI_COMM_WORLD` is supported).

3.4.5 Collective loops and choices

Given the SPMD nature of MPI programs, a program typically specifies some collective flow control, in the form of loops and conditional statements (choices), apart from rank-dependent behaviour. These must be accounted for in terms of the verification logic, matching terms of the form `loop T` and `choice T`. The related annotations can for the most part be automatically generated, through a process described later in this chapter. We focus now on its meaning and the associated verification process. Consider a fragment of the annotated finite differences program as illustration:

```
_(ghost \SessionType body = loopBody(_type);)
_(ghost \SessionType cont = next(_type);)
while (!converged(globalerr) && iter < MAX_ITER)
  ...
{
  _(ghost _type = body;)
  ...
  _(assert congruence(_type, skip()))
}
_(ghost _type = cont);
...
```

We recall that the loop is a collective one. Each process executes the loop exactly the same number of the times until a maximum number of iterations or numerical convergence is attained. The annotations above illustrates the verification logic. Just before the loop, two ghost variables are defined for the loop body (`body`) and its continuation (`cont`); for this to work, the protocol must have a prefix of the form `loop(T)` (and, if so, `T` becomes referenced by `body`). The loop body protocol is then matched by assertions within the loop, i.e., each loop iteration must reduce it to a term that is congruent with `skip()`. After the loop, the verification proceeds with the loop continuation (`cont`). The case of collective choices is handled similarly.

3.4.6 For-each protocols

A **`foreach`** protocol encodes repetition and it may be matched by an actual loop. In that case, the required annotations are similar to those for collective loops or choices. However, this is not always the case, since the projection of a **`foreach`** loop does not necessarily define an iteration for all participants. To illustrate these traits, consider a protocol where participant `0` sends a message to every other participant:

```
foreach i = 1 .. procs-1 { message 0 i float }
```

The required annotations are as follows:

```
if (rank == 0) {
  _(ghost \SessionType body = foreachBody(_type);)
  _(ghost \SessionType cont = next(_type);)
  for (to=1; to < procs; to++) ă
    ...
  {
    _(ghost _type = body;)
    ...
    MPI_Send (... to ...); // send message
    ...
    _(assert congruence(_type, skip()))
  } else {
```

```
    MPI_Recv (... 0 ...); // receive message
  }
}
```

As shown, the required annotations and verification for the program flow associated to rank `0` are similar to the case of collective loops. However, for all other participants the entire `foreach` loop is matched only by (the contract of) the `MPI_Recv` operation, since the projection of the protocol defines only the receipt of a message for those participants, i.e., the protocol is asserted as congruent to a term of the form `recv(0, ...)`. Apart from congruence, some "loop unfolding" logic supports this type of verification, described in Chapter 4.

3.5 Program annotation

After describing how protocols are overall specified and verified, we now explain the process of annotation over programs. There are three different levels of annotation, which we describe in this section:

1. The programmer may introduce simple protocol-related annotations to identify special control flow in the program, for instance collective loops;

2. Automatically generated annotations are inserted, guided by the protocol-related annotations and some program analysis;

3. Finally, complementary manual annotations are required for a number of reasons, (e.g., memory usage, unstated functional assumptions, or function contracts, which we discuss).

3.5.1 Protocol-related annotations

To identify some protocol-related features and guide verification in that sense, the programmer must introduce high-level, simple annotation marks in the program.

The first annotation mark of this kind has the form `_collective_(expr)` where `expr` is the C expression for the condition of a collective loop or choice. That is, the programmer changes

```
if (expr) { ... } else { ... }
```

to

```
if (_collective_(exptr)) { ... } else { ... }
```

when a collective choice is a stake, and

```
while (expr) { ... }
```

to

```
while (_collective_(expr)) { ... }
```

for a collective loop, which can also be a `do-while` or a `for` loop. These marks would otherwise be complex to infer automatically. Non-trivial program analysis is required to assert a given program fragment or expression is in fact collective, and our verification framework does not allows us to define built-in logic to express the simultaneous behavior of all processes (e.g., collective assertions as in [42]). A similar argument can be made for the other two types of mark described below.

The second type of annotation mark is `_foreach_(var, expr)` and applies to the identification of loops that match a `foreach` loop. We have that `var` identifies the iteration variable, and that `expr` is the loop condition expression. For instance, the programmer may change:

```
for (i=0; i < n; i++) { ... }
```

to

```
for (i=0; _foreach_(i, i < n); i++) { ... }
```

The final type of annotation mark has the form `_apply_(expr)`. This relates to the protocol action primitive `val`, used to model "injection" in the protocol of a specific value at runtime. For instance, variable `n` in the finite differences protocol (line 3, Figure 3.6) corresponds to the variable `psize` of the C program (line 8, Figure 3.2).

3.5.2 Automated generation of annotations

An annotator tool has been implemented to generate annotations automatically, greatly reducing the annotation effort. The annotator feeds only on protocol-related annotation marks in a C program, and generates another C program that essentially expands the simple annotation marks to the more complex logic discussed in Section 3.4. The process of annotation is illustrated in Table 3.1, concerning `_collective_` and `_foreach_` marks. The `_apply_` mark is maintained and dealt with by the definition of a function contract. The implementation of the annotator tool is described in Chapter 4.

3.5.3 Complementary manual annotations

The verification of a program typically requires complementary annotations that must be introduced manually by a programmer. This happens for a number of distinct reasons, which we now illustrate with examples and a discussion.

A major reason for manual annotations concerns memory usage. VCC requires fine-grained detail on memory usage, particularly on loops and function contracts, as a major aim of the tool is verifying memory safety. For instance, in the finite differences program we need to complement the annotations in the main loop as follows:

Original code	Transformed code

/* Collective choices */
if(_collective_(expr)) {
  ...
} else {
  ...
}
	_(ghost \SessionType _cTrue = choiceTrue(_type);)
	_(ghost \SessionType _cFalse = choiceFalse(_type);)
	_(ghost \SessionType _cCont = next(_type);)
	if (expr) {
	  _(ghost _type = cTrue;)
	  ...
	  _(assert congruence(_type, skip()))
	} else {
	  _(ghost _type = cFalse;)
	  ...
	  _(assert congruence(_type, skip()))
	}
	_(ghost _type = cCont;)

/* Collective loops */
while (_collective_(expr)) {
  ...
}
/* similarly for do-while
   and for loops */
	_(ghost \SessionType _lBody = loopBody (_type);)
	_(ghost \SessionType _lCont = next(_type);)
	while (expr) _ampi_loop {
	  _(ghost _type = _lBody;)
	  ...
	  _(assert congruence(_type, skip()))
	}
	_(ghost _type = lCont;)

/* foreach */
int var = ...;
...
while (_foreach_(var, expr)) {
  ...
}
/* similarly for do-while
   and for loops */
	int var = ...;
	...
	_(ghost STMap0 fBody = foreachBody(_type);)
	_(ghost \SessionType fCont = foreachCont(_type);)
	while(expr) {
	  _(ghost _type = fBody[var];)
	  ...
	  _(assert congruence(_type, skip())
	}
	_(ghost _type = fCont;)

/* Functions with MPI calls */
int functionName (int arg1) {
  ...
  MPI_call (...);
  ...
  return arg1;
}
	#define functionName (a) _functionName(a _ampi_arg)
	...
	int _functionName (int arg1 _ampi_arg_decl) _ampi_func {
	  ...
	  MPI_call (...);
	  ...
	  _ampi_on_return
	  return arg1;
	}

Table 3.1: Automatic generation of annotations

```
while (!converged(globalerr) && iter < MAX_ITER)
  _(writes &globalerr)
  _(writes \array_range(local, (unsigned) lpsize + 2))
{
  ...
}
```

The two `writes` clauses above identify memory that is updated within the loop, the `globalerr` variable and the array pointed to by variable `local`.

Manual annotations may also reflect implicit assumptions made by the programmer, which must be made explicit for verification of the program. For example, in the N-body program we have to introduce such an annotation, after the call to `MPI_Comm_size`, as

follows below:

```
...
MPI_Comm_size( MPI_COMM_WORLD, &procs );
_(assume procs <= MAX_P)
...
```

The program defines `MAX_P` as the maximum number of bodies, and `procs` corresponds to the number of processes. There is an implicit programmer assumption that `MAX_P` exceeds `procs` at runtime, as the program will fail to execute properly otherwise. Likewise, the program's verification will fail too, if the assumption is not made explicit.

A final reason for manual annotations concerns function contracts, which are required for modular (compositional) verification. The function contracts comprise manual annotations due to aspects discussed above, but also to the protocol verification itself. Consider a fragment of the contract for the `ComputeNewPos` function in the N-body example, as follows:

```
float ComputeNewPos( Particle particles, ParticleV pv, int npart,
                     float max_f, MPI_Comm commring
                        _ampi_arg_decl)
  ...
  _(writes \array_range (particles, (unsigned) (npart * 4)))
  _(writes \array_range (pv, (unsigned) (npart * 6)))
  _(requires npart < MAX_PARTICLES)
  _(requires first(_type) == action(allreduce(MPI_MIN),anyFloat
     (1)))
  _(ensures _type_out == next (_type))
{
  ...
  MPI_Allreduce( ... );
  ...
}
```

As shown also, `ComputeNewPos` contains a single MPI call to `MPI_Allreduce`. The contract of `MPI_Allreduce` in fact guarantees that a protocol reduction occurs through an action prefix `action(allreduce(MPI_MIN),anyFloat(1))`, but this has to be necessarily stated in the contract of `ComputeNewPos`. To deal with this burden, automated function inlining within the annotator logic may be particularly helpful in the future (given that MPI programs are typically not recursive in what regards communication logic).

3.5.4 Examples

An excerpt of the final result of the annotation process for our two running examples is provided in Figure 3.10 (finite differences program) and Figure 3.11 (N-body simulation program).

```
1  int main(int argc, char** argv _ampi_arg_decl) {
2    int procs;              // Number of processes
3    int rank;               // Process rank
4    MPI_Init(&argc, &argv);
5    MPI_Comm_size(MPI_COMM_WORLD, &procs);
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    ...
8    int psize = atoi(argv[1]);            // Global problem size
9    _apply_(psize);
10   if (rank == 0)
11     read_vector(work, lsize * procs);
12   MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0, MPI_COMM_WORLD);
13   int left  = (procs + rank - 1) % procs;  // Left neighbour
14   int right = (rank + 1) % procs;          // Right neighbour
15   int iter = 0;
16   // Loop until minimum differences converged or max iterations attained
17   _(ghost \SessionType lBody = loopBody(_type);)
18   _(ghost \SessionType lCont = next(_type);)
19   while (!converged(globalerr) && iter < MAX_ITER)
20     _(writes &globalerr)
21     _(writes \array_range(local, (unsigned) lpsize + 2))
22   {
23     _(ghost _type = lBody;)
24     if (rank == 0) {
25       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
26       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
27       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
28       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
29     } else if (rank == procs - 1) {
30       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
31       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
32       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
33       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
34     } else {
35       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
36       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
37       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
38       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status)
39     }
40     ...
41     MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
42     ...
43     _(assert congruence(_type,skip())
44   }
45   _(ghost _type = lCont;)
46   ...
47   _(ghost \SessionType cTrue = choiceTrue(_type);)
48   _(ghost \SessionType cFalse = choiceFalse(_type);)
49   _(ghost \SessionType cCont = next(_type);)
50   if (converged(globalerr)) { // Gather solution at rank 0
51     _(ghost _type = cTrue)
52     MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0, MPI_COMM_WORLD)
53     ...
54     _(assert _type == skip())
55   } else {
56     _(ghost _type = cFalse;)
57     ...
58     _(assert congruence(_type,skip()))
59   }
60   _(ghost _type = cCont)
61   MPI_Finalize();
62   return 0;
63 }
```

Figure 3.10: Annotated finite differences program

```
1  int main( int argc, char *argv[])
2  {
3    ...
4    MPI_Init( &argc, &argv );
5    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
6    MPI_Comm_size( MPI_COMM_WORLD, &procs );
7    _(assume procs <= MAX_P)
8    ...
9    npart = npart / procs;
10   _apply_(npart);
11   /* Get the sizes */
12   MPI_Allgather( &npart, 1, MPI_INT, counts, 1, MPI_INT, MPI_COMM_WORLD );
13   ...
14   cnt = 10;
15   sim_t = 0.0;
16   _(ghost \SessionType lb = loopBody(_type);)
17   _(ghost \SessionType lc = next(_type);)
18   while (cnt > 0)
19     _(writes \array_range (particles, (unsigned) (npart * 4)))
20     _(writes \array_range (pv, (unsigned) (npart * 6)))
21   {
22     _(ghost _type = lb;)
23     ...
24     _(ghost \SessionType lb1 = loopBody(_type);)
25     _(ghost \SessionType lc1 = next(_type);)
26     for (pipe=0; pipe<size; pipe++)
27       _(writes \array_range (particles, (unsigned) (npart * 4)))
28       _(writes \array_range (pv, (unsigned) (npart * 6)))
29     {
30       _(ghost _type = lb1;)
31       _(ghost \SessionType ct = choiceTrue(_type);)
32       _(assert congruence(choiceFalse(_type), skip()))
33       _(ghost \SessionType cc = next(_type);)
34       if (pipe != procs-1) {
35         _(ghost _type = ct;)
36         if (rank == 0) {
37           MPI_Send( sendbuf, npart * 4, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
38           MPI_Recv( recvbuf, npart * 4, MPI_FLOAT, left,  0, MPI_COMM_WORLD, &statuses[0] );
39         } else {
40           MPI_Recv( recvbuf, npart * 4, MPI_FLOAT, left,  0, MPI_COMM_WORLD, &statuses[0] );
41           MPI_Send( sendbuf, npart * 4, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
42         }
43         _(assert congruence(_type, skip()))
44       }
45       _(ghost _type = cc;)
46       /* Compute forces (2D only) */
47       ...
48       _(assert congruence(_type,skip()))
49     }
50     _(ghost _type = lc1;)
51     /* Once we have the forces, we compute the changes in position */
52     sim_t += ComputeNewPos( particles, pv, npart, max_f, MPI_COMM_WORLD _ampi_arg);
53     _(assert congruence(_type, skip()))
54   }
55   ...
56   _(ghost _type = lc;)
57   MPI_Finalize();
58   return 0;
59 }
60
61 double ComputeNewPos( Particle particles, ParticleV pv, int npart,
62             float max_f, MPI_Comm commring  _ampi_arg_decl)
63   _ampi_func
64   _(writes \array_range (particles, (unsigned) (npart * 4)))
65   _(writes \array_range (pv, (unsigned) (npart * 6)))
66   _(requires npart < MAX_PARTICLES)
67   _(requires first(_type) == action(allreduce(MPI_MIN),anyFloat(1)))
68   _(ensures _type_out == next (_type))
69   _(requires commring == MPI_COMM_WORLD)
70 {
71   ...
72   MPI_Allreduce( &dt_est, &dt_new, 1, MPI_DOUBLE, MPI_MIN, commring);
73   ...
74   _ampi_on_return
75   return dt_old;
76 }
```

Figure 3.11: Annotated N-body simulation program

Chapter 4

Design and Implementation

This chapter presents the core design and implementation aspects of the verification framework, comprising the base logic of the "annotated MPI" library (Section 4.1), and the annotator tool for automatic generation of verification annotations in the body of a program (Section 4.2). The source code for both is provided as an annex to this thesis.

4.1 Annotated MPI library

4.1.1 Mock MPI header

Primitive	Description
`MPI_Init`	Initialize the MPI execution environment
`MPI_Finalize`	Finalize the MPI execution environment
`MPI_Comm_size`	Get number of processes
`MPI_Comm_rank`	Get process rank
`MPI_Send`	Blocking send
`MPI_Recv`	Blocking receive
`MPI_Bcast`	Broadcasts from one process to all other processes
`MPI_Scatter`	Scatters data from one process to all processes
`MPI_Gather`	Gathers data from all processes to one process
`MPI_Allgather`	All-to-all gather
`MPI_Reduce`	Reduction with result available to a single process
`MPI_Allreduce`	Reduction with result available to all processes

Table 4.1: Annotated MPI — supported communication primitives

The "annotated MPI" is a core body of definitions that can be imported in the context by an annotated C program, through the inclusion of a mock MPI header, `mpi.h`. The program may be compiled normally by using the concrete header of a MPI implementation in the place of the mock one. However, for verification purposes, the program should only contain calls to the subset of MPI communication primitives listed in Table 4.1. In

terms of size, the annotated MPI library comprises 27 files with approximately 1500 lines of verification logic.

4.1.2 Parameterization

In order to verify MPI programs, two core program parameters must be parameterised: the process rank and the number of processes. This kind of "glue" must be "known" by the verifier at all times, and is defined through the `ampi_glue_t` data structure shown in Figure 4.1. For every annotated MPI function , a ghost parameter of type `_ampi_glue_t` is introduced, "propagating" this parameterization.

```
typedef struct
{
  int procs; /* Number of processes */
  int rank;  /* Process rank */
  _(invariant procs > 1)
  _(invariant rank >= 0)
  _(invariant rank < procs)
  _(invariant procs < 32768)
} ampi_glue_t;
```

Figure 4.1: Overall parameterization

On Figure 4.1, the constraints state that the number of processes lies between 2 to 32678, and that the value of rank is greater or equal to 0 and lower than the number of processes. Some additional but necessarily compatible constraints may be placed in the body of a program in the form of assumptions, e.g., stating a lower upper bound for the number of processes or actually making that number constant. Note also that MPI does allow for more than 32768 processes, but this bound was judged to be quite sufficient for the current stage of work and and the verification experiments we conducted (Chapter 5). It should be stressed that the verification operated over MPI programs *really* is parametric according to these constraints. That is, an MPI program is checked against a protocol specification *for every* possible number of processes and *every possible* process rank (every process expressed by the program, up to the number of processes), as opposed to a particular choice of values for them.

4.1.3 Refinement types

```
_(ghost typedef \bool \IRefinement[\integer]);
_(ghost typedef \bool \FRefinement[float]);
```

Figure 4.2: Support for refinement types

In association to protocols, restrictions on the value domains of transmitted data can be imposed through *refinement types*. The syntax shown in Figure 4.2 defines refinement types as logical predicates, i.e., as maps from of integers (`\IRefinement`) or floats (`\FRefinement`) to boolean values. Typically, these can be expressed as anonymous functions of the form `\lambda type; predicate` in VCC. For example, `\lambda \integer x; \true` is an instance of `\IRefinement` that imposes no value restrictions; so is `\lambda \integer x; x > 0 && x < 100`, but which restrains the value of an integer to the $(0, 100)$ interval. The use of refined types is illustrated on the example projection functions of Chapter 3 (§ 3.4.2, Figures 3.8 and 3.9).

4.1.4 Session type representation

Protocol projections are expressed as `\SessionType` terms, cf. § 3.4.2. The definition is provided on Figure 4.3. Using VCC terminology [5], `\SessionType` is an "inductively defined type". The definition can be interpreted as a grammar where `\SessionType` is the root symbol, resembling the protocol specification grammar recognized by the Eclipse/Xtext framework (§ 3.3, Figure 3.4):

- A `\SessionData` term represents message data, with an associated refined type and length;
- An `\Action` term represents a communication action and associated arguments, if any, including `send` and `recv` to represent projections and the support collective communication primitives (e.g., `bcast`, `reduce`);
- A `\SessionType` term can either be: the terminated protocol (`skip`); a communication action (`action`); a protocol sequence (`seq`); a collective choice or loop (`choice` and`loop`); a for-each protocol (`foreach`); or, finally, a function abstraction (`val`).

The meaning and use of `STMap`, `STMap0` and `struct _vcc_math_type_STMap` is quite intricate in Figure 4.3, due to some VCC technicalities. To clarify, beginning `STMap0`, an instance of `STMap0` is a map from integers to `\SessionType` terms, e.g., as in `\lambda \integer i; action(bcast(i), ...)`. An `STMap` instance merely wraps with the later with a `body` constructor, for the verification logic did not seem to work otherwise. For instance, a for-each protocol instance has the following general form: `forach(lo, hi, body(\lambda \integer i; T))`. Finally, `struct _vcc_math_type_STMap` is just the VCC way of making a forward declaration to `STMap`
.

```
_(datatype \SessionData
  {
    case intRef    (\IRefinement, \integer);
    case floatRef  (\FRefinement, \integer);
  })
_(datatype \Action
  {
    case size        ();
    case val         ();
    case send        (\integer);
    case recv        (\integer);
    case bcast       (\integer);
    case gather      (\integer);
    case scatter     (\integer);
    case reduce      (\integer, MPI_Op);
    case allreduce   (MPI_Op);
    case allgather   ();
  })
_(datatype \SessionType
  {
    case skip    ();
    case action  (\Action, \SessionData);
    case seq     (\SessionType, \SessionType);
    case choice  (\SessionType, \SessionType);
    case loop    (\SessionType);
    case foreach (\integer, \integer, struct _vcc_math_type_STMap);
    case abs     (struct _vcc_math_type_STMap);
  })
_(ghost typedef \SessionType STMap0[\integer];)
_(datatype STMap {
    case body(STMap0);
  }
)
```

Figure 4.3: Session type representation

```
_(pure STMap0 message(\integer from, \integer to, \SessionData sd);)
_(axiom \forall \integer from, to; \forall \SessionData sd;
  from != to ==> message(from,to,sd)[from] == action(send(to),sd))
_(axiom \forall \integer from, to; \forall \SessionData sd;
  from != to ==> message(from,to,sd)[to] == action(recv(from),sd))
_(axiom \forall \integer from, to, rank; \forall \SessionData sd;
  rank != from && rank != to ==> message(from,to,sd)[rank] == skip())
```

Figure 4.4: Message projections

4.1.5 Protocol projection

Protocol projections are defined by inferring `recv` and `send` actions from `message` terms in VCC projection functions (cf. § 3.4.2). The logic is defined as shown on Figure 4.4. The `message` function takes integers representing the source (`from`) and destination (`destination`) of the message, plus a session data argument (`sd`); it yields back a map from integers (process ranks) to `\SessionType` terms (an instance of `STMap0`). In line with the intended nature of projections, the projection of a message for a given rank either yields a `send(from)` action, a `recv(to)` action, or `skip()`.

4.1.6 Structural congruence

```
_(pure \bool congruence(\SessionType, \SessionType);)
_(axiom \forall \SessionType t; congruence(t, t))  // (1)
_(axiom \forall \SessionType t1,t2,t3;
  congruence(t1,t2) && congruence(t2,t3) ==> congruence(t1,t3)) // (2)
_(axiom \forall \SessionType t; congruence(seq(skip(),t), t)) // (3)
_(axiom \forall \SessionType t1,t2,t3;
  congruence(seq(seq(t1,t2),t3), seq(t1,seq(t2,t3)))) // (4)
_(axiom \forall \SessionType t;
        \forall \integer lo,hi;
        \forall STMap b;
  lo > hi ==> congruence(seq(foreach(lo,hi,b),t),t)) // (5)
_(axiom \forall \SessionType t;
        \forall \integer lo,hi;
        \forall STMap0 f;
  lo <= hi && congruence(f[lo],skip()) ==>
    congruence(seq(foreach(lo,hi,body(f)),t),
               seq(foreach(lo+1,hi,body(f)),t))) // (6)
```

Figure 4.5: Structural congruence predicate

A structural congruence predicate is defined to identify semantically equivalent protocols. This helps the verification logic with term "rewriting" for protocol reductions. The definition is shown on Figure 4.5. It comprises axioms (numbered in comments): for (1) reflexive and (2) transitive congruence of terms; (3) recognition of the terminated protocol; (4) rewriting of `seq` terms; and, finally, (5, 6) termination of for-each iterations.

4.1.7 Protocol reductions

The reduction of a `\SessionType` instance occurs through the logic of MPI function contracts (cf. § 3.4.4), plus automatically generated annotations (§ 3.5.2, § 4.2, Table 3.1) for collective choices and loops (§ 3.4.5) and for-each protocols (§ 3.4.6). The associated definitions are shown on Figure 4.6. They comprise the extraction of action prefixes

(`prefix`), protocol continuations (`next`), and bodies for collective blocks and for-each iterations (`loopBody`, `choiceTrue`, `choiceFalse`, `foreachBody`).

```
_(axiom \forall \Action p;
        \forall \SessionData mt;
        \forall \SessionType t;
        first(seq(action(p, mt), t)) == action(p, mt))
_(axiom \forall \Action p;
        \forall \SessionData mt;
        \forall \SessionType t;
        next(seq(action(p, mt), t)) == t)
_(axiom \forall \SessionType t;
        \forall STMap0 f;
        \forall \integer lo, hi;
        lo <= hi ==>
           first(seq(foreach(lo, hi, body(f)), t)) == first(seq(f[lo],
              seq(foreach(lo+1, hi, body(f)), t))))
_(axiom \forall \SessionType t;
        \forall STMap0 f;
        \forall \integer lo, hi;
        (lo <= hi && !congruence(f[lo],skip())) ==>
            next(seq(foreach(lo, hi, body(f)), t))
               ==
            next(seq(f[lo], seq(foreach(lo+1, hi, body(f)), t))))
_(axiom \forall \SessionType t1,t2;
        next(seq(loop(t1),t2)) == t2)
_(axiom \forall \SessionType t1,t2,t3;
        next(seq(choice(t1,t2),t3)) == t3)
_(axiom \forall \SessionType t;
        \forall \SessionData sd;
        next(seq(action(val(),sd),t)) == t)
_(axiom \forall \SessionType t1, t2;
        loopBody(seq(loop(t1),t2)) == t1)
_(axiom \forall \SessionType t1, t2, t3;
        choiceTrue(seq(choice(t1,t2),t3)) == t1)
_(axiom \forall \SessionType t1, t2, t3;
        choiceFalse(seq(choice(t1,t2),t3)) == t2)
_(axiom \forall \SessionType t;
        \forall STMap0 f;
        \forall \integer a,b;
        foreachBody(seq(foreach(a,b,body(f)),t))  == f)
```

Figure 4.6: Protocol reduction logic

4.2 Program annotator

This section describe the implementation of the annotator tool. The annotator source code makes use of the Clang/LLVM framework as base infrastructure. Some Clang/LLVM background is provided first, which is then followed by an overall description of the operation and implementation of the annotator.

```
1  int main(int argc, const char **argv) {
2    CXIndex Index = clang_createIndex(0,0);
3    CXTranslationUnit tu = clang_parseTranslationUnit(Index, 0,
4                             argv, argc, 0, 0, CXTranslationUnit_None);
5    CXCursor cursor = clang_getTranslationUnitCursor (tu);
6    clang_visitChildren(cursor, cursorCallback, NULL);
7    clang_disposeTranslationUnit(tu);
8    clang_disposeIndex(Index);
9    ...
10   return 0;
11 }
12 ...
13 CXChildVisitResult cursorCallback(CXCursor cursor, CXCursor parent,
       CXClientData client_data){
14    CXCursorKind kind = clang_getCursorKind(cursor);
15    ...
16    if (kind == CXCursor_WhileStmt) {
17      // Handle 'while' loops
18      WhileStmt* stmt = (WhileStmt*) cursor.data[1];
19      ...
20    }
21    ...
22    return CXChildVisit_Recurse;
23 }
```

Figure 4.7: Example use of a Clang/LLVM cursor

4.2.1 The Clang/LLVM framework

Clang [4] is an open-source compiler for the C family of programming languages, built on top of the LLVM infrastructure [24] for optimization and code generation. This combination provides provide allows portability and high-performance code generation for many target platforms. Furthermore, Clang/LLVM provides an infrastructure to write tools for static analysis and transformation of programs, in the form of source code or LLVM intermediate bytecode.

The Clang C/C++ development API provides an abstract syntax tree (AST) abstraction for C programs, and associated AST traversal mechanisms, known as visitors and cursors.

Visitors allow a syntactic-driven AST traversal in the traditional sense, but backward-compatible support is not guaranteed by the head development team for future. Such guarantee exists for cursors, the choice for this work, even if it performs AST traversal using a more low-level procedural callback mechanism.

Figure 4.7 briefly demonstrates AST traversal using cursors. At line 2, the `Index` variable represents a set of translation units compiled and linked together. Line 3 is the main entry point for the Clang C API, providing the ability to parse a source file into a translation unit that can then be queried by other functions in the API. Line 5 returns the top cursor of the source file. Finally, in line 6, the function that will start the iteration on cursor is invoked and the callback function `cursorCallback` is invoked in the process. The callback function the may process AST information, as shown for a **`while`** statement in lines 16–20, and then instruct the AST traversal to either recurse (as in line 22), stop, or resume at the same AST level.

4.2.2 Implementation

Figure 4.8 illustrates the operation of the annotator. The tool takes as input the C+MPI source code of a program, and yields back transformed source code with automatically generated annotations.

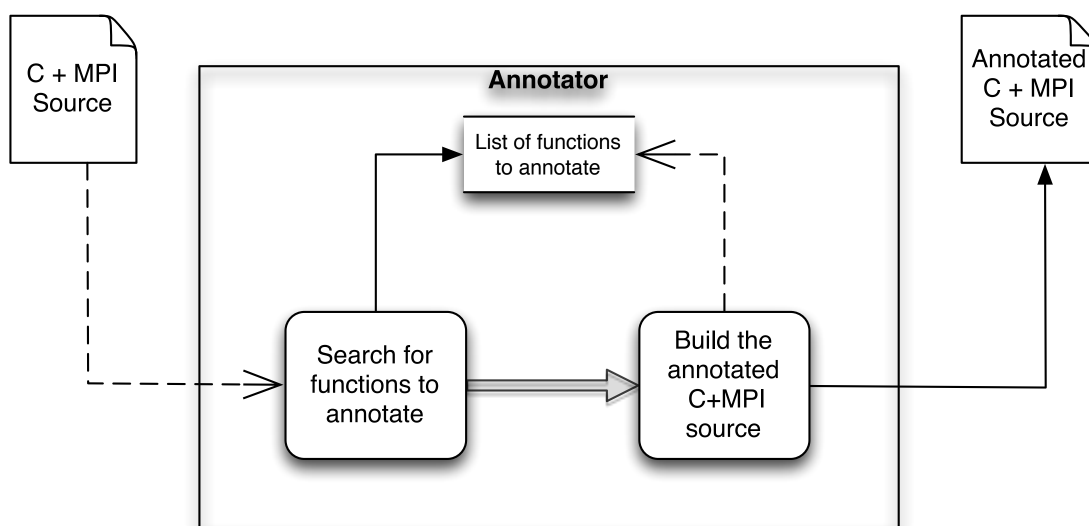

Figure 4.8: Annotator tool — overall operation

Before generating any annotated source code, the tool needs to determine the functions that need to be annotated. The base case is that of a function that uses MPI primitives or that contain protocol-related annotations introduced by the programmer (e.g.,`_collective_`, see Table 3.1). The annotator has to consider however if there are client (callee) functions of the later case. This cycle goes on, building a list of functions to annotate and associated

information, until it is asserted that no more functions need to be accounted.

After completing the list of functions to annotate, the tool proceeds with the generation of annotations in the output file. This stage illustrated in Figure 4.9 with an excerpt of the tool's source code. The fragment at stake relates to the generation of annotations of foreach protocol annotations (the `_foreach_` case of Table 3.1) in the Clang cursor callback function that produces the output file.

```
1  CXChildVisitResult anCyclesVisitor(CXCursor cursor, CXCursor parent,
     CXClientData client_data) {
2    CXCursorKind kind = clang_getCursorKind(cursor);
3    ...
4    if (kind >= CXCursor_WhileStmt && kind <= CXCursor_ForStmt) { // loop
5      ...
6      clang_visitChildren(cursor, aFindForeach, ad);
7      ...
8      if (ad->condExpr == 0) { // for & while loops
9        int lVarIdx = ad->gvCtr++;
10         if (ad->foreachVarname != 0){ // Foreach-case
11           // Annotations for extracting loop body and continuation
12          ad->out << "_(ghost \\SessionType _lBody" << lVarIdx
13                  << " = "<< "foreachBody" <<"(_type);)\n";
14          ad->out << "_(ghost \\SessionType _lCont" << lVarIdx
15                  << " = " << "foreachCont" <<"(_type);)\n";
16         }else{
17           ...
18         }
19         handleCycle(kind, cursor, (Stmt*) cursor.data[1], lVarIdx, ad);
20         ad->out << "_(ghost _type = _lCont" << lVarIdx << ";)\n";
21         ...
22         return CXChildVisit_Continue;
23      } else { // do-while loop
24        ...
25      }
26   }
27   ...
28 }
```

Figure 4.9: Annotator tool — generation of annotations

Chapter 5

Evaluation

This chapter describes the verification framework's evaluation over a sample set of MPI programs. The chapter begins with a summary of the chosen MPI programs for evaluation (Section 5.1). Results are then presented and discussed with respect to two metrics: the annotation effort required by verification (Section 5.2), and the execution time of VCC (Section 5.3). The VCC protocol projections and annotated program code for the sample programs are provided in Appendix A.

5.1 Sample MPI programs

The sample program set comprises four programs, adapted from textbooks [7, 13, 34]. Two of them were presented in Chapter 3 (§ 3.2): the finite differences [7] and N-body simulation programs [13]. The other two examples are from [34]: a vector dot product computation, and a linear system solver using the Jacobi method.

The examples had to be preliminarily adjusted in some aspects. The most important one was converting the of use non-blocking communication primitives (`MPI_Isend`, `MPI_Irecv`, `MPI_Wait` and `MPI_Waitall`) onto code that uses only blocking primitives (`MPI_Send` and `MPI_Recv`). Other aspects had to be adjusted due to limitations of VCC. For instance, VCC can not handle C functions with variable number of arguments like `printf`. VCC also can not handle floating point logic in association to flow control, e.g., the tool will crash if it finds a block of the type `if (a < b){ ...}` where `a` and `b` are of type `float` or `double`. The latter type of logic had to be replaced with function calls and mock declarations (e.g., the `converged` function in the finite differences example).

5.2 Annotation effort

5.2.1 Annotation process

All the examples were annotated according to the methodology described in Chapter 3. That is, part of the annotations were generated automatically and the complementary ones

were introduced manually.

The annotation took in account an arbitrary number of processes, except for the vector dot product case, due to a general shortcoming in the actual framework. Essentially, using the current framework, it is not possible to annotate functions' contracts to match a for-each protocol with a parametric number of iterations, but instead only a constant number of iterations. The finite differences and N-body example, also use for-each protocols, but only at the level of the `main` function. The issue can be dealt with automated function inlining (see § 3.5.3 for a discussion), or further work in the verification logic. The following code fragment illustrates the technical difficulty, using a fragment of the contract of the `Read_vector` function in the example at stake:

```
void Read_vector (...) {
...
_(requires
 _ampi_rank == 0 ==>
    first(_type) == action(send(1),anyFloat(n_bar)) &&
    first(next(_type)) == action(send(2), anyFloat(n_bar)) &&
    first(next(next(_type))) == action(send(3), anyFloat(n_bar))
 )
_(ensures
 _ampi_rank == 0 ==>
    next(next(next(_type))) == _type_out
)
_(requires
 _ampi_rank > 0 ==>
    first(_type) == action(recv(0),anyFloat(n_bar))
)
_(ensures
 _ampi_rank > 0 ==>
    next(_type) == _type_out
)
...
```

The pre and post-condition expressing protocol reduction for rank 0 are only valid for 4 processes, since the corresponding protocol fragment to match is a simple for-each protocol, encoding messages sent from participant 0 to all other processes:

```
...
foreach(1,p-1,
  body(\lambda \integer q;
    message(0, q, anyFloat(n/p))[rank])
)
```

Program	LOC	F	A	M	A / M	M / F
Finite differences	256	8	38	17	2.2	2.1
Jacobi iteration	429	9	55	56	1.0	7.0
N-body simulation	362	14	52	26	2.0	3.7
Vector dot product	357	6	38	30	1.3	5.0

LOC: lines of code; F: annotated functions; A: automated annotations; M: manual annotations

Table 5.1: Annotation effort

5.2.2 Results

Table 5.1 summarizes the annotation effort. For each example, it indicates the number of code lines in each program, the number of annotated C functions, the number of automatically and manually generated annotations and their ratio, plus the ratio of manual annotations per function. In all examples except the Jabobi iteration, the number of automatically generated annotations is greater than the number of manual annotations. The results also hint at the correlation the ratio between automatic and manual annotations (A/M) and the effort of annotating function contracts manually (M/F). In the Jacobi iteration and vector dot product examples, communication code is split modularly into several functions, leading to more manual annotations. Note that the F column in the table does not refer only to contracts of functions that contain MPI calls. In fact, the finite differences example has all communication code in the `main` function, and the N-body simulation example in just two functions (`main` and `ComputeNewPos`). In these last two examples there is a much higher A/M ratio.

5.3 Verification time

5.3.1 Test setup and execution

The execution time of VCC for each of examples was measured on a Windows 7 machine with two Intel x86 2.8 GHz cores and 4 GB of RAM. For each example, separate timings imposing were took, the restriction, that is, a VCC assumption, of a fixed number of processes, powers of 2 from 4 to 64. The exception was the vector dot product that was annotated assuming a constant number of 4 processes, for the reasons discussed in the previous section.

Program	4	8	16	32	64
Finite differences	14.8	28.0	70.1	260.0	timeout
Jacobi iteration	11.0	16.0	12.4	11.5	12.0
N-body simulation	6.1	8.3	42.8	timeout	timeout
Vector dot product	2.3	N/A	N/A	N/A	N/A

Table 5.2: VCC execution time (seconds)

5.3.2 Results

The average times in seconds of 5 VCC executions per example are shown on Table 5.2. The timeout entries indicate that the verification did not end after 5 minutes of waiting. The first observation is a stable performance in the Jacobi iteration example, which can be explained by the exclusive use of collective communication operations, no for-each loops, and a reduced distinction between the operations of several participants. On the contrary, the finite differences and N-body simulation make use of both point-to-point and collective communications plus for-each loops, and the verification time grows exponentially with the number of processes.

Chapter 6

Conclusion

6.1 Summary of contributions

This thesis presented a verification methodology for C+MPI programs, combining the theory of multi-party session types with a deductive software verification approach. The essential idea was that of expressing communication protocols as multi-party session types, and verifying the adherence of C+MPI programs to such protocols using deductive software verification. The work is also described in peer-reviewed publications over the period of this thesis [25, 26]. In detail, the core contributions were as follows:

1. The methodology has been characterized from protocol specification to verification. This comprised the translation of protocol specifications to the deductive verification logic, the overall verification process and logic, and the annotation of a program for verification.

2. Deductive verification logic has been defined in the VCC framework comprising the definition of protocols as multi-party session types and corresponding reduction through verification. A tool has also been developed to automatically generate an important fraction of C program annotations required for verification.

3. An evaluation has been conducted using sample MPI programs. Even if the set of programs considered is still small in number and constrained in terms of features, it was sufficient to illustrate the applicability of the approach and some core challenges in the verification of MPI programs.

6.2 Future work

For future work, the following challenges are considered to be of core relevance:

1. Real-world programs must be considered, as opposed to simple textbook examples, to identify extra requirements and measure the overall applicability and scalability of this approach.

2. The support of a larger MPI subset is required, for instance comprising primitives for non-blocking communications or communicator creation, which are quite common in MPI programs, even in simple textbook examples.

3. More automated program annotation techniques must be developed, to alleviate the burden of manual annotation by the programmer. For instance, even relatively simple techniques such as function inlining may in principle greatly reduce the number of manual annotations in a program.

4. The inference and distinction between rank-dependent and collective program behavior may in turn also alleviate the need for protocol-related annotations, and possibly lead to a more robust verification approach. A number of concepts and techniques may be useful in that regard, e.g., parallel control-flow graphs [2], collective assertions [42], or program slicing [45].

Appendix A

Evaluation examples

A.1 Finite differences

A.1.1 Protocol projection

```
_(ghost _(pure) \SessionType ftype (\integer rank)
 _(ensures \result ==
  seq(
   action(size(), intRef(\lambda \integer y; y>0, 1)),
   abs(body(\lambda \integer p;
    seq(
     action(val(), intRef(\lambda \integer x; x>0 && x%p==0, 1)),
     abs(body(\lambda \integer n;
      seq(
       action(scatter(0), floatRef(\lambda float v; \true, n)),
      seq(
       loop(
        seq(
         foreach(0, p-1,
          body(\lambda \integer i;
           seq(
            message(i, (p+i-1)%p,
                      floatRef(\lambda float v; \true, 1))[rank],
            message(i, (i+1)%p,
                      floatRef(\lambda float v; \true, 1))[rank]))),
         action(allreduce(MPI_MAX),floatRef(\lambda float v; \true, 1)))
            ),
       choice(
        action(gather(0), floatRef(\lambda float v; \true, n)),
        skip()
       )))))))))
  );
)
```

A.1.2 Program code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <mpi.h>
5
6  #define MAX_SIZE 1024
7  #define MAX_ITER 100
8  #define MAX_ERR 0.0001
9
10 #include "fdiff.h"
11
12 void read_array(float *data, int size)
13   _(requires \thread_local_array(data, (unsigned) size))
14   _(writes \array_range (data, (unsigned) size))
15 ;
16
17 void write_array(float *data, int size);
18
19 void compute(float *data, float dataNIter[], int size)
20   _(requires size >= 0)
21   _(requires \thread_local_array(data, (unsigned) size))
22   _(writes \array_range(dataNIter, (unsigned) size))
23 ;
24
25 float maxerror(float *data, float dataNIter[], int size)
26   _(requires size >= 0)
27   _(requires \thread_local_array(data, (unsigned) size))
28   _(requires \thread_local_array(dataNIter, (unsigned) size))
29 ;
30
31 int converged(float err);
32
33 void swap(float x[], float y[], int size)
34   _(writes \array_range(x, (unsigned) size))
35   _(writes \array_range(y, (unsigned) size))
36 ;
37 void read_array(float *data, int size) {
38   int i;
39
40 #ifndef _VCC_LIMITATIONS_
41   for (i = 0; i < size; i++)
42     scanf("%f", &data[i]);
43 #else
44   for (i = 0; i < size; i++)
45     data[i] = 31;
46 #endif
47 }
48
49 void write_array(float *data, int size) {
```

```
50   int i;
51 #ifndef _VCC_LIMITATIONS_
52   for (i = 0; i < size; i++)
53     printf("%f\n", data[i]);
54 #endif
55 }
56 
57 void compute(float *data, float dataNIter[], int size) {
58   int i;
59   for (i = 1; i < size - 1; i++)
60     dataNIter[i] = (data [i - 1] + 2 * data [i] + data [i + 1]) / 4;
61 }
62 
63 float maxerror(float *data, float dataNIter[], int size) {
64   float error = 0;
65   int i;
66   for (i = 1; i < size - 1; i++)
67 #ifndef _VCC_LIMITATIONS_
68     error += fabs(dataNIter[i] - data [i]);
69 #else
70     error = dataNIter[i] - data [i];
71 #endif
72   return error;
73 }
74 
75 int converged(float err) {
76 #ifndef _VCC_LIMITATIONS_
77   return err < MAX_ERR;
78 #else
79   return 1;
80 #endif
81 }
82 
83 void swap(float x[], float y[], int size) {
84   int i;
85   for (i = 0; i < size; i++) {
86     float swap_data = x[i];
87     x[i] = y[i];
88     y[i] = swap_data;
89   }
90 }
91 
92 int read_size(int procs)
93   _(ensures \result >0 && \result % procs == 0 && \result < MAX_SIZE-2)
94 ;
95 
96 int main(int argc, char** argv _ampi_arg_decl) {
97   int np;              // Number of processes
98   int me;              // Process rank
```

```
99
100   MPI_Init(&argc, &argv);
101   MPI_Comm_rank(MPI_COMM_WORLD, &me);
102   MPI_Comm_size(MPI_COMM_WORLD, &np);
103
104   _(assume _ampi_procs == 16)
105
106   if (argc < 2)   // too few arguments
107   return 1;
108
109   int psize = read_size(np);             // Global problem psize
110   _apply_(psize);
111
112   if (psize <= 0 || psize >= MAX_SIZE - 2 || psize % np != 0)
113     return 1;                           // returns in case of an invalid
          psize
114
115   float work[MAX_SIZE];                  // Initial input data (used only
        by rank 0)
116   if (me == 0)
117     read_array(work, psize);
118
119   // Scatter input data
120   // Two extra slots in local buffer are required for boundary exchange
121   int lsize = psize / np;              // Local problem psize
122   float local[MAX_SIZE];                  // Local data buffer per
      process.
123   MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0,
      MPI_COMM_WORLD);
124
125   float globalerr = 999.0f;
126
127   // Loop, until finite differences converge to a minimum error.
128   int iter = 0;
129   int left = (np + me - 1) % np;          // Left neighbor rank
130   int right = (me + 1) % np;               // Right neighbor rank
131
132   _(ghost \SessionType lb = loopBody(_type);)
133   _(ghost \SessionType lc = next(_type);)
134
135   while (!converged(globalerr) && iter < MAX_ITER)
136     _(writes &globalerr)
137     _(writes \array_range(local, (unsigned) lsize + 2))
138   {
139     _(ghost _type = lb;)
140     MPI_Status status;                    // MPI status data
141
142     if (me == 0) {
143       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
```

```
144        MPI_Send(&local[lsize],   1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
145        MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD,
               &status);
146        MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &
               status);
147      } else if (me == np-1) {
148        MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD,
               &status);
149        MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &
               status);
150        MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
151        MPI_Send(&local[lsize],   1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
152      } else {
153        MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &
               status);
154        MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
155        MPI_Send(&local[lsize],   1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
156        MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD,
               &status);
157      }
158      float nextLocal[MAX_SIZE];              // Local data for the next
             iteration
159      compute(local, nextLocal, lsize);
160      float localerr = maxerror(local, nextLocal, lsize);
161      MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX,
             MPI_COMM_WORLD);
162      swap (local, nextLocal, lsize + 2);
163      iter++;
164      _(assert congruence(_type,skip()))
165    }
166     _(ghost _type = lc;)
167
168
169    _(ghost \SessionType ct = choiceTrue(_type);)
170    _(ghost \SessionType cf = choiceFalse(_type);)
171    _(ghost \SessionType cc = next(_type);)
172    if (converged(globalerr)) {
173        _(ghost _type = ct)
174      // Gather data at rank 0 for converged solution
175      MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0,
             MPI_COMM_WORLD);
176
177      if (me == 0)
178        write_array(work, psize);
179      _(assert congruence(_type, skip()))
180    } else {
181      _(ghost _type = cf;)
182  #ifndef _VCC_LIMITATIONS_
183      printf ("failed to converge after %d iterations!", MAX_ITER);
```

```
184 #endif
185     _(assert congruence(_type, skip()))
186   }
187   _(ghost _type = cc;)
188
189   MPI_Finalize();
190
191   return 0;
192 }
```

A.2 N-body simulation

A.2.1 Protocol projection

```
1  _(pure \SessionData anyFloat(\integer len) _(ensures \result == floatRef
       (\lambda float n; \true, len));)
2  _(pure \SessionData anyInt(\integer len) _(ensures \result == intRef(\
       lambda \integer n; \true, len));)
3  _(pure \SessionData natural(\integer len)  _(ensures \result ==  intRef
       (\lambda \integer n; n >= 0, len));)
4
5  _(ghost _(pure) \SessionType ftype (\integer rank)
6    _(ensures \result ==
7      seq(action(size(), intRef(\lambda \integer procs; procs > 0, 1)),
8      seq(abs(body(\lambda \integer procs;
9          seq(action(val(), intRef(\lambda \integer size; size > 0, 1)),
10         seq(abs(body(\lambda \integer size;
11             seq(action(allgather(),anyInt(procs)),
12       seq(loop(
13                 seq(loop(
14                     seq(choice(
15                         seq(foreach(0, procs-1,
16                             body(\lambda \integer i;
17                                 seq(message(i, (i+1)%procs, anyFloat(
                                      size * 4))[rank],
18                                 skip()))),
19                         skip()),
20                         skip()),
21                     skip())),
22                seq(action(allreduce(MPI_MIN),anyFloat(1)),
23                skip()))),
24            skip())))),
25        skip())))),
26     skip()))
27    );
28 )
```

A.2.2 Program code

```
1  #include <mpi.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <math.h>
5  #include <string.h>
6
7  extern float drand48(void);
8  extern float fsqrt(float v);
9
10 #define MAX_SIZE 100
11 #define MAX_PARTICLES 4000
12 #define MAX_P          8
13
14 #define valParticle(v,p,e) ((v)[((p) * 4) + (e)])
15 #define valParticleV(v,p,e) ((v)[((p) * 6) + (e)])
16
17 #include "nbodypipe.h"
18
19 int lessThan (float a, float b);
20
21 int greaterThan (float a, float b);
22
23 int equalsToZero (float a);
24
25 /* Pipeline version of the algorithm... */
26 /* we really need the velocities as well... */
27 /*typedef struct {
28     float x, y, z;
29     float mass;
30     } Particle;
31 */
32
33 // x, y, z, mass
34 typedef float *Particle;
35
36 /* We use leapfrog for the time integration ... */
37 /* typedef struct {
38     float xold, yold, zold;
39     float fx, fy, fz;
40     } ParticleV;
41 */
42
43 // xold, yold, zold, fx, fy, fz
44 typedef float *ParticleV;
45
46 void InitParticles(Particle particles, ParticleV pv, int npart);
47 float ComputeForces( Particle myparticles, Particle others,
48           ParticleV pv, int npart );
49 float ComputeNewPos( Particle particles, ParticleV pv, int npart,
```

```
50            float max_f, MPI_Comm commring  _ampi_arg_decl);
51
52
53  int
54  main( int argc, char *argv[] _ampi_arg_decl) {
55      float       particles[MAX_PARTICLES * 4];   /* Particles on ALL nodes
             */
56      float       pv[MAX_PARTICLES * 6];          /* Particle velocity */
57      float       sendbuf[MAX_PARTICLES * 4],     /* Pipeline buffers */
58            recvbuf[MAX_PARTICLES * 4];
59      int           counts[MAX_P],              /* Number on each processor
            */
60            displs[MAX_P];              /* Offsets into particles */
61      int           rank, procs, npart, i, j,
62            offset;                     /* location of local particles */
63      int           totpart,                    /* total number of particles
             */
64            cnt;                        /* number of times in loop */
65      float       sim_t;                          /* Simulation time */
66      int           pipe, left, right;
67      MPI_Status  statuses[2];
68
69      _(assume _ampi_procs <= MAX_P)
70      MPI_Init( &argc, &argv );
71      MPI_Comm_rank( MPI_COMM_WORLD, &rank );
72      MPI_Comm_size( MPI_COMM_WORLD, &procs );
73
74  /* Get the best ring in the topology */
75      left = (procs + rank - 1) % procs;
76      right = (rank + 1) % procs;
77
78  /* Everyone COULD have a different size ... */
79      if (argc < 2) {
80  #ifndef _VCC_LIMITATIONS_
81    fprintf( stderr, "Usage: %s n\n", argv[0] );
82    MPI_Abort( MPI_COMM_WORLD, 1 );
83  #else
84    return 1;
85  #endif
86      }
87      npart = atoi(argv[1]);
88
89      if (npart <= 0 || npart > MAX_PARTICLES || npart % procs != 0) {
90  #ifndef _VCC_LIMITATIONS_
91    fprintf( stderr, "%d is too many; max is %d\n",
92       npart*procs, MAX_PARTICLES );
93    MPI_Abort( MPI_COMM_WORLD, 1 );
94  #else
95    return 1;
```

```
96  #endif
97      }
98
99      npart = npart / procs;
100     _apply_(npart);
101 /* Get the sizes and displacements */
102     MPI_Allgather( &npart, 1, MPI_INT, counts, 1, MPI_INT,
            MPI_COMM_WORLD );
103     //    _(assume \forall int i; i >= 0 && i < procs ==> counts [i] <
            MAX_PARTICLES)
104     displs[0] = 0;
105     for (i=1; i<procs; i++)
106       _(writes \array_range(displs, (unsigned) procs))
107     {
108   displs[i] = _(unchecked) (displs[i-1] + counts[i-1]);
109     }
110     totpart = _(unchecked) (displs[procs-1] + counts[procs-1]);
111
112 /* Generate the initial values */
113     InitParticles( particles, pv, npart);
114     offset = displs[rank];
115     cnt    = 10;
116
117     sim_t = 0.0f;
118     _(ghost \SessionType lb = loopBody(_type);)
119     _(ghost \SessionType lc = next(_type);)
120     while (cnt > 0)
121       _(writes \array_range (particles, (unsigned) (npart * 4)))
122       _(writes \array_range (pv, (unsigned) (npart * 6)))
123     _ampi_loop
124     {
125     _(ghost _type = lb;)
126        cnt--;
127        float max_f, max_f_seg;
128
129   /* Load the initial sendbuffer */
130   memcpy( sendbuf, particles, (unsigned long) (npart * sizeof(Particle))
          );
131   max_f = 0.0f;
132   _(ghost \SessionType lb1 = loopBody(_type);)
133   _(ghost \SessionType lc1 = next(_type);)
134   for (pipe=0; pipe < procs; pipe++)
135           _(writes \array_range (particles, (unsigned) (npart * 4)))
136           _(writes \array_range (pv, (unsigned) (npart * 6)))
137       _ampi_loop
138   {
139     _(ghost _type = lb1;)
140           _(ghost \SessionType ct = choiceTrue(_type))
141           _(assert congruence(choiceFalse(_type), skip()))
```

```
142             _(ghost \SessionType cc = next(_type))
143         if (pipe != procs-1) {
144               _(ghost _type = ct)
145           if (rank == 0) {
146             MPI_Send( sendbuf, npart * 4, MPI_FLOAT, right, 0,
147           MPI_COMM_WORLD);
148             MPI_Recv( recvbuf, npart * 4, MPI_FLOAT, left,  0,
149           MPI_COMM_WORLD, &statuses[0] );
150           } else {
151             MPI_Recv( recvbuf, npart * 4, MPI_FLOAT, left,  0,
152           MPI_COMM_WORLD, &statuses[0] );
153             MPI_Send( sendbuf, npart * 4, MPI_FLOAT, right, 0,
154           MPI_COMM_WORLD);
155           }
156               _(assert congruence(_type, skip()))
157         }
158
159             _(ghost _type = cc;)
160         /* Compute forces (2D only) */
161         max_f_seg = ComputeForces( particles, sendbuf, pv, npart );
162         if (greaterThan(max_f_seg, max_f)) max_f = max_f_seg;
163         /* Push pipe */
164         //      if (pipe != size-1)
165         //    MPI_Waitall( 2, request, statuses );
166         memcpy( sendbuf, recvbuf, _(unchecked) ((unsigned long) counts[pipe]
                * sizeof(Particle)) );
167         _(assert congruence(_type,skip()))
168       }
169       _(ghost _type = lc1;)
170       /* Once we have the forces, we compute the changes in position */
171       sim_t += ComputeNewPos( particles, pv, npart, max_f, MPI_COMM_WORLD
             _ampi_arg );
172
173       /* We could do graphics here (move particles on the display) */
174       _(assert congruence(_type, skip()))
175         }
176         _(ghost _type = lc;)
177         MPI_Finalize();
178         return 0;
179     }
180
181
182     void InitParticles( Particle particles, ParticleV pv, int npart )
183       _(writes \array_range (particles, (unsigned) (npart * 4)))
184       _(writes \array_range (pv, (unsigned) (npart * 6)))
185       _(requires npart < MAX_PARTICLES)
186     {
187         int i;
188         for (i=0; i<npart; i++) {
```

```
189       valParticle(particles, i, 0) = drand48();
190       valParticle(particles, i, 1) = drand48();
191       valParticle(particles, i, 2) = drand48();
192       valParticle(particles, i, 3) = 1.0f;
193       valParticleV(pv, i, 0) = valParticle(particles, i, 0);
194       valParticleV(pv, i, 1) = valParticle(particles, i, 1);
195       valParticleV(pv, i, 2) = valParticle(particles, i, 2);
196       valParticleV(pv, i, 3) = 0;
197       valParticleV(pv, i, 4) = 0;
198       valParticleV(pv, i, 5) = 0;
199     }
200 }
201
202 float ComputeForces( Particle myparticles, Particle others,
203           ParticleV pv, int npart )
204   _(requires \thread_local_array (myparticles, (unsigned) (npart * 4)))
205   _(requires \thread_local_array (others, (unsigned) (npart * 4)))
206   _(writes \array_range (pv, (unsigned) (npart * 6)))
207   _(requires npart < MAX_PARTICLES)
208 {
209   float max_f, rmin;
210   int i, j;
211
212   max_f = 0.0f;
213   for (i=0; i<npart; i++) {
214     float xi, yi, mi, rx, ry, mj, r, fx, fy;
215     rmin = 100.0f;
216     xi   = valParticle(myparticles, i, 0);
217     yi   = valParticle(myparticles, i, 1);
218     fx   = 0.0f;
219     fy   = 0.0f;
220     for (j=0; j<npart; j++) {
221       rx = xi - valParticle(others, j, 0);
222       ry = yi - valParticle(others, j, 1);
223       mj = valParticle(others, j, 2);
224       r  = rx * rx + ry * ry;
225       /* ignore overlap and same particle */
226       if (equalsToZero(r)) continue;
227       if (lessThan(r, rmin)) rmin = r;
228       /* compute forces */
229       r  = r * fsqrt(r);
230       fx -= mj * rx / r;
231       fy -= mj * ry / r;
232     }
233     valParticleV(pv, i, 3) += fx;
234     valParticleV(pv, i, 4) += fy;
235     /* Compute a rough estimate of (1/m)|df / dx| */
236     fx          = fsqrt(fx*fx + fy*fy)/rmin;
237     if (greaterThan(fx, max_f)) max_f = fx;
```

```
238     }
239     return max_f;
240   }
241
242
243   float ComputeNewPos( Particle particles, ParticleV pv, int npart,
244             float max_f, MPI_Comm commring  _ampi_arg_decl)
245     _ampi_func
246     _(writes \array_range (particles, (unsigned) (npart * 4)))
247     _(writes \array_range (pv, (unsigned) (npart * 6)))
248     _(requires npart < MAX_PARTICLES)
249     _(requires first(_type) == action(allreduce(MPI_MIN),anyFloat(1)))
250     _(ensures _type_out == next (_type))
251     _(requires commring == MPI_COMM_WORLD)
252
253   {
254     int i;
255     float      a0, a1, a2;
256     static      float dt_old = 0.001f, dt = 0.001f;
257     float      dt_est, new_dt, dt_new;
258
259     /* integation is a0 * x^+ + a1 * x + a2 * x^- = f / m */
260     a0   = 2.0f / (dt * (dt + dt_old));
261     a2   = 2.0f / (dt_old * (dt + dt_old));
262     float minus1 = 4.0f - 5.0f;
263     a1   = minus1 * (a0 + a2);      /* also -2/(dt*dt_old) */
264
265     for (i=0; i<npart; i++) {
266       float xi, yi;
267       /* Very, very simple leapfrog time integration.  We use a variable
268          step version to simplify time-step control. */
269       xi               = valParticle(particles, i, 0);
270       yi               = valParticle(particles, i, 1);
271       valParticle(particles, i, 0) = (valParticleV(pv, i, 3) - a1 * xi -
            a2 *
272               valParticleV(pv, i, 0)) / a0;
273       valParticle(particles, i, 1) = (valParticleV(pv, i, 4) - a1 * yi -
            a2 *
274               valParticleV(pv, i, 5)) / a0;
275       valParticleV(pv, i, 0) = xi;
276       valParticleV(pv, i, 1) = yi;
277       valParticleV(pv, i, 3) = 0;
278       valParticleV(pv, i, 4) = 0;
279     }
280
281     /* Recompute a time step. Stability criteria is roughly
282        2/fsqrt(1/m |df/dx|) >= dt.  We leave a little room */
283     dt_est = 1.0f/fsqrt(max_f);
284     /* Set a minimum: */
```

```
285   if (lessThan(dt_est, 1.0e-6f)) dt_est = 1.0e-6f;
286   MPI_Allreduce( &dt_est, &dt_new, 1, MPI_FLOAT, MPI_MIN, commring);
287   /* Modify time step */
288   if (lessThan(dt_new, dt)) {
289     dt_old = dt;
290     dt     = dt_new;
291   }
292   else if (greaterThan(dt_new, 4.0f * dt)) {
293     dt_old = dt;
294     dt    *= 2.0f;
295   }
296
297   _ampi_on_return
298   return dt_old;
299 }
```

A.3 Parallel Jacobi iteration

A.3.1 Protocol projection

```
1  /* Global type:
2
3  size procs: {x:nat | N % x == 0}.
4  val(MAX_DIM) ->
5  val(N) ->
6  scatter 0 float[MAX_DIM * N].
7  scatter 0 float[N].
8  allgather float[N].
9  loop
10   allgather float[N].
11   end.
12 choice
13   gather float[N].end,
14   end.
15 end
16 */
17 _(pure \SessionData anyFloat(\integer len)
18  _(ensures \result == floatRef(\lambda float n; \true, len));)
19
20 _(ghost _(pure) \SessionType ftype (\integer rank)
21   _(ensures \result ==
22       seq(action(val(), intRef(\lambda \integer maxDim; maxDim > 0,1)),
23       seq(abs(body(\lambda \integer maxDim;
24       seq(action(size(), intRef(\lambda \integer procs; procs > 0 &&
            maxDim % procs == 0, 1)),
25       seq(abs(body(\lambda \integer procs;
26           seq(action(val(),intRef(\lambda \integer size; size > 0 &&
                size % procs == 0 && size <= maxDim, 1)),
27           seq(abs(body(\lambda \integer size;
```

```
28                    seq(action(scatter(0),anyFloat(maxDim * size)),
29                    seq(action(scatter(0),anyFloat(size)),
30                    seq(action(allgather(),anyFloat(size)),
31                    seq(loop(seq(action(allgather(),anyFloat(size)), skip()
                          )),
32                    seq(choice(seq(action(gather(0),anyFloat(size)),skip())
                          , skip()),
33                    skip())))))))),
34                 skip()))))),
35              skip()))))),
36           skip()))
37       );
38   )
```

A.3.2 Program code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
5
6
7  #define MAX_DIM 1024
8  #define MAX_ITER 1000
9
10 #include "parallel_jacobi.h"
11 #ifndef _VCC_LIMITATIONS_
12 // The original definitions from Peter Pacheco
13 typedef float MATRIX_T[MAX_DIM][MAX_DIM];
14 #define val(m,r,c) ((m)[(r)][(c)])
15 #define MATRIX_DECL(var) float var[MAX_DIM][MAX_DIM]
16 #else
17 typedef float* MATRIX_T;
18 #define val(m,r,c) ((m)[((r) * MAX_DIM) + (c)])
19 #define MATRIX_DECL(var) float var[MAX_DIM * MAX_DIM]
20 #endif
21
22 void Jacobi_iteration(
23    MATRIX_T  A_local    /* in  */,
24    float*    x_local    /* out */,
25    float*    b_local    /* in  */,
26    float*    x_old      /* in  */,
27    int       n          /* in  */,
28    int       p          /* in  */,
29    int       my_rank    /* in  */
30    _ampi_arg_decl_no_type
31 )
32   _(requires my_rank == _ampi_rank)
33   _(requires p == _ampi_procs)
34   _(requires n > 0 && n <= MAX_DIM)
```

```
35   _(requires n % p == 0)
36   _(requires \thread_local_array(A_local, MAX_DIM * MAX_DIM))
37   _(requires \thread_local_array(x_old, MAX_DIM))
38   _(requires \thread_local_array(b_local, MAX_DIM))
39   _(writes \array_range(x_local, MAX_DIM))
40 ;
41
42 int converged(float *x, float *y, int n)
43   _(requires n > 1)
44   _(requires \thread_local_array(x, (unsigned) n))
45   _(requires \thread_local_array(y, (unsigned) n))
46 ;
47
48
49 int Parallel_jacobi(
50         MATRIX_T  A_local    /* in  */,
51         float*    x_local    /* out */,
52         float*    b_local    /* in  */,
53         int       n          /* in  */,
54         int       max_iter   /* in  */,
55         int       p          /* in  */,
56         int       my_rank    /* in  */
57         _ampi_arg_decl
58 )
59   _ampi_func
60   _(requires my_rank == _ampi_rank)
61   _(requires p == _ampi_procs)
62   _(requires n > 0 && n <= MAX_DIM)
63   _(requires n % p == 0)
64   _(requires max_iter > 0 && max_iter <= MAX_ITER)
65   _(requires \thread_local_array(A_local, MAX_DIM * MAX_DIM))
66   _(requires \thread_local_array(b_local, MAX_DIM))
67   _(writes \array_range(x_local, MAX_DIM))
68   _(requires
69     first(_type) == action(allgather(),anyFloat(n)) &&
70     loopBody(next(_type)) == seq(action(allgather(),anyFloat(n)),skip())
71    )
72   _(ensures _type_out == next(next(_type)))
73 ;
74
75 void Read_matrix(
76          char*     prompt   /* in  */,
77          MATRIX_T  A_local  /* out */,
78          int       n        /* in  */,
79          int       my_rank  /* in  */,
80          int       p        /* in  */
81          _ampi_arg_decl)
82   _ampi_func
83   _(requires my_rank == _ampi_rank)
```

```
84   _(requires p == _ampi_procs)
85   _(requires n > 0 && n <= MAX_DIM)
86   _(requires n % p == 0)
87   _(writes \array_range(A_local, MAX_DIM * MAX_DIM))
88   _(requires first(_type) == action(scatter(0), anyFloat(MAX_DIM*n)))
89   _(ensures _type_out == next(_type))
90 ;
91
92
93 void Read_vector(
94          char*  prompt     /* in  */,
95          float  *x_local   /* out */,
96          int    n          /* in  */,
97          int    my_rank    /* in  */,
98          int    p          /* in  */
99          _ampi_arg_decl)
100  _ampi_func
101  _(requires p == _ampi_procs)
102  _(requires my_rank == _ampi_rank)
103  _(requires n % p == 0)
104  _(requires n > 0 && n <= MAX_DIM)
105  _(writes \array_range(x_local, (unsigned) n))
106  _(requires first(_type) == action(scatter(0), anyFloat(n)))
107  _(ensures _type_out == next(_type))
108 ;
109
110 void Print_matrix(
111          char*     title      /* in */,
112          MATRIX_T  A_local    /* in */,
113          int       n          /* in */,
114          int       my_rank    /* in */,
115          int       p          /* in */
116          _ampi_arg_decl)
117  _ampi_func
118  _(requires p == _ampi_procs)
119  _(requires my_rank == _ampi_rank)
120  _(requires n > 0 & n <= MAX_DIM)
121  _(requires n % p == 0)
122  _(requires \thread_local_array(A_local, MAX_DIM*MAX_DIM))
123  _(requires first(_type) == action(gather(0), anyFloat(MAX_DIM*(n))))
124  _(ensures _type_out == next(_type))
125 ;
126
127 void Print_vector(
128          char*  title      /* in */,
129          float  x_local[]  /* in */,
130          int    n          /* in */,
131          int    my_rank    /* in */,
132          int    p          /* in */
```

```
133         _ampi_arg_decl)
134   _ampi_func
135   _(requires p == _ampi_procs)
136   _(requires my_rank == _ampi_rank)
137   _(requires n > 0 & n <= MAX_DIM)
138   _(requires n % p == 0)
139   _(requires \thread_local_array(x_local, (unsigned) n))
140   _(requires first(_type) == action(gather(0), anyFloat(n)))
141   _(ensures _type_out == next(_type))
142 ;
143
144 #ifndef _VCC_LIMITATIONS_
145 // The original definitions from Peter Pacheco
146 typedef float MATRIX_T[MAX_DIM][MAX_DIM];
147 #define val(m,r,c) ((m)[(r)][(c)])
148 #define MATRIX_DECL(var) float var[MAX_DIM][MAX_DIM]
149 #else
150 typedef float* MATRIX_T;
151 #define val(m,r,c) ((m)[((r) * MAX_DIM) + (c)])
152 #define MATRIX_DECL(var) float var[MAX_DIM * MAX_DIM]
153 #endif
154 int main(int argc, char** argv _ampi_arg_decl) {
155     int         p;
156     int         my_rank;
157     MATRIX_DECL(A_local);
158     float       x_local[MAX_DIM];
159     float       b_local[MAX_DIM];
160     int         n;
161     int         max_iter;
162     int         converged;
163
164     MPI_Init(&argc, &argv);
165     _apply_(MAX_DIM);
166     MPI_Comm_size(MPI_COMM_WORLD, &p);
167     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
168
169     if (argc < 3) {
170       return 1; // invalid number of arguments
171     }
172
173     n = atoi(argv[0]);
174     max_iter = atoi(argv[2]);
175
176     if (n <= 0 || n > MAX_DIM || n % p != 0 || max_iter <= 0 || max_iter
            >= MAX_ITER) {
177       return 1; // invalid arguments
178     }
179     _apply_(n);
180     Read_matrix("Enter the matrix", A_local, n, my_rank, p _ampi_arg);
```

```
181     Read_vector("Enter the right-hand side", b_local, n, my_rank, p
            _ampi_arg);
182
183     converged = Parallel_jacobi(A_local, x_local, b_local, n,
184         max_iter, p, my_rank _ampi_arg);
185     _(ghost \SessionType ta = choiceTrue(_type);)
186     _(ghost \SessionType tb = choiceFalse(_type);)
187     _(ghost \SessionType tc = next(_type);)
188     if (converged)
189     {
190       _(ghost _type = ta;)
191       Print_vector("The solution is", x_local, n, my_rank, p _ampi_arg);
192     }
193     else
194     {
195       _(ghost _type = tb;)
196       if (my_rank == 0) {
197 #ifndef _VCC_LIMITATIONS_
198         printf("Failed to converge in %d iterations\n", max_iter);
199 #endif
200       }
201     }
202     _(assert congruence(_type,skip()))
203     _(ghost _type = tc;)
204     MPI_Finalize();
205     _ampi_on_return
206 }  /* main */
207
208 #define Swap(x,y) {float* temp; temp = x; x = y; y = temp;}
209
210 int Parallel_jacobi(
211         MATRIX_T  A_local    /* in  */,
212         float*    x_local    /* out */,
213         float*    b_local    /* in  */,
214         int       n          /* in  */,
215         int       max_iter   /* in  */,
216         int       p          /* in  */,
217         int       my_rank    /* in  */
218         _ampi_arg_decl
219 )
220 {
221     int     i_local, i_global, j;
222     int     n_bar;
223     int     iter_num;
224     float   x_temp1[MAX_DIM];
225     float   x_temp2[MAX_DIM];
226     float*  x_old;
227     float*  x_new;
228
```

```
229     n_bar = n/p;
230
231     /* Initialize x */
232     MPI_Allgather(b_local, n_bar, MPI_FLOAT, x_temp1,
233         n_bar, MPI_FLOAT, MPI_COMM_WORLD);
234     x_new = x_temp1;
235     x_old = x_temp2;
236
237     iter_num = 0;
238     _(ghost \SessionType lb = loopBody(_type);)
239     _(ghost \SessionType lc = next(_type);)
240     do
241        _(invariant iter_num <= max_iter) // verification: to avoid
              overflow warning
242        _(invariant \thread_local_array(x_new, MAX_DIM))
243        _(invariant \thread_local_array(x_old, MAX_DIM))
244     {
245         _(ghost _type = lb;)
246         iter_num++;
247
248         /* Interchange x_old and x_new */
249         Swap(x_old, x_new);
250
251         Jacobi_iteration(A_local, x_local, b_local, x_old, n, p, my_rank
               , _ampi_arg_name);
252
253         MPI_Allgather(x_local, n_bar, MPI_FLOAT, x_new,
254                      n_bar, MPI_FLOAT, MPI_COMM_WORLD);
255         _(assert congruence(_type, skip()))
256     } while ((iter_num < max_iter) && converged(x_new,x_old,n));
257     _(ghost _type = lc;)
258   _ampi_on_return
259   return 1;
260 } /* Jacobi */
261
262 void Read_matrix(
263         char*     prompt   /* in  */,
264         MATRIX_T  A_local  /* out */,
265         int       n        /* in  */,
266         int       my_rank  /* in  */,
267         int       p        /* in  */
268         _ampi_arg_decl)
269 {
270
271     int       i, j;
272     MATRIX_DECL(temp);
273     int       n_bar;
274
275     n_bar = n/p;
```

```
276
277      /* Fill dummy entries in temp with zeroes */
278  #ifndef _VCC_LIMITATIONS_
279      for (i = 0; i < n; i++)
280        _(writes \array_range(temp, MAX_DIM * MAX_DIM)) {
281          for (j = n; j < MAX_DIM; j++)
282            _(writes \array_range(&val(temp,i,n), MAX_DIM-(unsigned) n)) {
283              val(temp,i,j) = 0.0f;
284          }
285      }
286
287      if (my_rank == 0) {
288          printf("%s\n", prompt);
289          for (i = 0; i < n; i++)
290              for (j = 0; j < n; j++)
291                  scanf("%f",&temp[i][j]);
292      }
293  #endif
294      MPI_Scatter(temp, n_bar*MAX_DIM, MPI_FLOAT, A_local,
295          n_bar*MAX_DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
296
297      _ampi_on_return
298  }  /* Read_matrix */
299
300  void Read_vector(
301           char*  prompt     /* in  */,
302           float  *x_local   /* out */,
303           int    n          /* in  */,
304           int    my_rank    /* in  */,
305           int    p          /* in  */
306           _ampi_arg_decl)
307  {
308
309      int   i;
310      float temp[MAX_DIM];
311      int   n_bar;
312
313      n_bar = n/p;
314
315      if (my_rank == 0) {
316  #ifndef _VCC_LIMITATIONS_
317          printf("%s\n", prompt);
318          for (i = 0; i < n; i++)
319              scanf("%f", &temp[i]);
320  #endif
321      }
322
323      MPI_Scatter(temp, n/p, MPI_FLOAT, x_local, n/p, MPI_FLOAT,
324          0, MPI_COMM_WORLD);
```

```
325
326     _ampi_on_return
327 }  /* Read_vector */
328
329 void Print_matrix(
330          char*      title      /* in */,
331          MATRIX_T   A_local    /* in */,
332          int        n          /* in */,
333          int        my_rank    /* in */,
334          int        p          /* in */
335          _ampi_arg_decl)
336 {
337
338     int       i, j;
339     MATRIX_DECL(temp);
340     int       n_bar;
341
342     n_bar = n/p;
343
344     _(assume \arrays_disjoint(A_local, MAX_DIM*MAX_DIM, temp, MAX_DIM*
           MAX_DIM))
345     MPI_Gather(A_local, n_bar*MAX_DIM, MPI_FLOAT, temp,
346          n_bar*MAX_DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
347
348     if (my_rank == 0) {
349 #ifndef _VCC_LIMITATIONS_
350         printf("%s\n", title);
351         for (i = 0; i < n; i++) {
352             for (j = 0; j < n; j++)
353                 printf("%4.1f ", temp[i][j]);
354             printf("\n");
355         }
356 #endif
357     }
358     _ampi_on_return
359 }  /* Print_matrix */
360
361 void Print_vector(
362          char*  title      /* in */,
363          float  x_local[]  /* in */,
364          int    n          /* in */,
365          int    my_rank    /* in */,
366          int    p          /* in */
367          _ampi_arg_decl)
368 {
369
370     int   i;
371     float temp[MAX_DIM];
372     int   n_bar;
```

```
373
374     n_bar = n/p;
375
376     MPI_Gather(x_local, n_bar, MPI_FLOAT, temp, n_bar, MPI_FLOAT,
377         0, MPI_COMM_WORLD);
378
379     if (my_rank == 0) {
380 #ifndef _VCC_LIMITATIONS_
381         printf("%s\n", title);
382         for (i = 0; i < n; i++)
383             printf("%4.1f ", temp[i]);
384         printf("\n");
385 #endif
386     }
387     _ampi_on_return
388 }  /* Print_vector */
```

A.4 Parallel dot product

A.4.1 Protocol projection

```
1  _(ghost _(pure) \SessionData anyFloat(\integer len)
2       _(ensures \result == floatRef(\lambda float v; \true, len));
3  )
4  _(ghost _(pure) \SessionType ftype (\integer rank)
5  _(ensures \result ==
6    seq(action(size(), intRef(\lambda \integer p; p == 4, 1)),
7    seq(abs(body(\lambda \integer p;
8      seq(action(val(),intRef(\lambda \integer n; n > 0 && n % p == 0, 1))
         ,
9      seq(abs(body(\lambda \integer n;
10       seq(action(bcast(0), intRef(\lambda \integer v; v == n, 1)),
11       seq(foreach(1,p-1, body(\lambda \integer q;
12       seq(message(0,q,anyFloat(n/p))[rank],skip()))),
13       seq(foreach(1,p-1, body(\lambda \integer q;
14         seq(message(0,q,anyFloat(n/p))[rank],skip()))),
15       seq(action(allreduce(MPI_SUM), anyFloat(1)),
16       seq(foreach(1,p-1, body(\lambda \integer q;
17         seq(message(q,0,anyFloat(1))[rank],skip()))),
18       skip()))))))),
19     skip()))))),
20   skip()))
21   );
22 )
```

A.4.2 Program code

```
1  /* parallel_dot1.c -- Computes a parallel dot product.  Uses
       MPI_Allreduce.
```

```
2   *
3   * Input:
4   *     n: order of vectors
5   *     x, y:  the vectors
6   *
7   * Output:
8   *     the dot product of x and y as computed by each process.
9   *
10  * Note:  Arrays containing vectors are statically allocated.  Assumes
       that
11  *     n, the global order of the vectors, is evenly divisible by p, the
12  *     number of processes.
13  *
14  * See Chap 5, pp. 76 & ff in PPMPI.
15  */
16 #include <stdio.h>
17 #include <mpi.h>
18
19 #define MAX_LOCAL_ORDER 100
20 #define FAKE_USER_INPUT 60
21
22 #include "parallel_dot.h"
23
24 void Read_vector(char* prompt, float local_v[], int n_bar, int p,
25      int my_rank _ampi_arg_decl)
26 ;
27
28 float Parallel_dot(float local_x[], float local_y[], int n_bar
      _ampi_arg_decl)
29   _ampi_func
30   _(requires n_bar > 0)
31   _(requires \thread_local_array(local_x, (unsigned) n_bar))
32   _(requires \thread_local_array(local_y, (unsigned) n_bar))
33
34   _(requires
35     first(_type) == action(allreduce(MPI_SUM),anyFloat(1))
36     )
37   _(ensures
38     next(_type) == _type_out
39     )
40 ;
41
42 void Print_results(float dot, int my_rank, int p _ampi_arg_decl)
43   _ampi_func
44 _(requires p == _ampi_procs)
45 _(requires my_rank == _ampi_rank)
46 _(requires _ampi_procs == 4)
47
48 _(requires
```

```
49    _ampi_rank == 0 ==>
50    first(_type) == action(recv(1), anyFloat(1)) &&
51    first(next(_type)) == action(recv(2), anyFloat(1)) &&
52    first(next(next(_type))) == action(recv(3), anyFloat(1))
53    )
54  _(ensures
55    _ampi_rank == 0 ==> next(next(next(_type))) == _type_out
56    )
57
58  _(requires
59    _ampi_rank > 0 ==> first(_type) == action(send(0),anyFloat(1))
60  )
61  _(ensures
62    _ampi_rank > 0 ==> next(_type) == _type_out
63  )
64  ;
65
66  float  Serial_dot(float x[], float y[], int n)
67    _(requires n > 0)
68    _(requires \thread_local_array(x, (unsigned) n))
69    _(requires \thread_local_array(y, (unsigned) n))
70  ;
71
72  void Scan_vector(float v[], int len)
73    _(writes \array_range (v, (unsigned) len))
74  ;
75
76  int
77  main(int argc, char* argv[] _ampi_arg_decl)
78  {
79    float  local_x[MAX_LOCAL_ORDER];
80    float  local_y[MAX_LOCAL_ORDER];
81    int    n;
82    int    n_bar;  /* = n/p */
83    float  dot;
84    int    p;
85    int    my_rank;
86
87    _(assume _ampi_procs == 4)
88    MPI_Init(&argc, &argv);
89    MPI_Comm_size(MPI_COMM_WORLD, &p);
90    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
91
92    if (my_rank == 0) {
93  #ifndef _VCC_LIMITATIONS_
94        printf("Enter the order of the vectors\n");
95        scanf("%d", &n);
96  #else
97      n = FAKE_USER_INPUT;
```

```
98  #endif
99    }
100
101   _(assume n > 0 && n < MAX_LOCAL_ORDER)
102   _(assume n % p == 0)
103   _apply_(n);
104   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
105
106   n_bar = n/p;
107
108   _(assert
109   my_rank == 0 ==> first(_type) == action(send(1),anyFloat(n_bar))
110   )
111   Read_vector("the first vector", local_x, n_bar, p, my_rank  _ampi_arg)
        ;
112   Read_vector("the second vector", local_y, n_bar, p, my_rank  _ampi_arg
        );
113   dot = Parallel_dot(local_x, local_y, n_bar  _ampi_arg);
114   Print_results(dot, my_rank, p _ampi_arg);
115
116   MPI_Finalize();
117   _ampi_on_return
118   return 0;
119 }  /* main */
120
121 void Scan_vector(float v[], int len) {
122 #ifndef _VCC_LIMITATIONS_
123   printf("Enter %s\n", prompt);
124   int i;
125   for (i = 0; i < len; i++)
126     scanf("%f", &v[i]);
127 #else
128   int i;
129   for (i = 0; i < len; i++)
130     v[i] = 27;
131 #endif
132 }
133
134
135 /*****************************************************************/
136 void Read_vector(
137          char*  prompt     /* in  */,
138          float  local_v[]  /* out */,
139          int    n_bar      /* in  */,
140          int    p          /* in  */,
141          int    my_rank    /* in  */
142    _ampi_arg_decl)
143   _ampi_func
144   _(requires n_bar > 0 && n_bar < MAX_LOCAL_ORDER)
```

```
145   _(requires \thread_local_array(local_v, (unsigned) n_bar))
146   _(requires p == _ampi_procs)
147   _(requires my_rank == _ampi_rank)
148   _(writes \array_range (local_v, (unsigned) n_bar))
149   _(requires _ampi_procs == 4)
150
151   _(requires
152     _ampi_rank == 0 ==>
153     first(_type) == action(send(1),anyFloat(n_bar)) &&
154     first(next(_type)) == action(send(2), anyFloat(n_bar)) &&
155     first(next(next(_type))) == action(send(3), anyFloat(n_bar))
156     )
157   _(ensures
158     _ampi_rank == 0 ==> next(next(next(_type))) == _type_out
159     )
160
161   _(requires
162     _ampi_rank > 0 ==> first(_type) == action(recv(0),anyFloat(n_bar))
163     )
164   _(ensures
165     _ampi_rank > 0 ==> next(_type) == _type_out
166     )
167
168
169 {
170     int i, q;
171     float temp[MAX_LOCAL_ORDER];
172     MPI_Status status;
173
174     if (my_rank == 0) {
175       Scan_vector (local_v, n_bar);
176
177       // send to 1
178       Scan_vector (temp, n_bar);
179       MPI_Send(temp, n_bar, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
180
181       // send to 2
182       Scan_vector (temp, n_bar);
183       MPI_Send(temp, n_bar, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
184
185       // send to 3
186       Scan_vector (temp, n_bar);
187       MPI_Send(temp, n_bar, MPI_FLOAT, 3, 0, MPI_COMM_WORLD);
188     } else {
189         MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
190             &status);
191     }
192     _ampi_on_return
193 }  /* Read_vector */
```

```
194
195
196 /*****************************************************************/
197 float Serial_dot(
198           float  x[]  /* in */,
199           float  y[]  /* in */,
200           int    n    /* in */) {
201     int    i;
202     float  sum = 0.0f;
203
204     for (i = 0; i < n; i++)
205         sum = sum + x[i]*y[i];
206     return sum;
207 } /* Serial_dot */
208
209
210 /*****************************************************************/
211 float Parallel_dot(
212           float  local_x[]  /* in */,
213           float  local_y[]  /* in */,
214           int  n_bar        /* in */
215     _ampi_arg_decl) {
216
217     float  local_dot;
218     float  dot = 0.0f;
219
220     local_dot = Serial_dot(local_x, local_y, n_bar);
221     MPI_Allreduce(&local_dot, &dot, 1, MPI_FLOAT,
222         MPI_SUM, MPI_COMM_WORLD);
223     _ampi_on_return
224     return dot;
225 } /* Parallel_dot */
226
227
228 /*****************************************************************/
229 void Print_results(
230          float  dot      /* in */,
231          int    my_rank  /* in */,
232          int    p        /* in */
233    _ampi_arg_decl) {
234     int         q;
235     float       temp;
236     MPI_Status  status;
237
238     if (my_rank == 0) {
239 #ifndef _VCC_LIMITATIONS_
240         printf("dot = \n");
241         printf("Process 0 > %f\n", dot);
242         MPI_Recv(&temp, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD,
```

```
243                 &status);
244         printf("Process %d > %f\n", 1, temp);
245         MPI_Recv(&temp, 1, MPI_FLOAT, 2, 0, MPI_COMM_WORLD,
246                 &status);
247         printf("Process %d > %f\n", 2, temp);
248         MPI_Recv(&temp, 1, MPI_FLOAT, 3, 0, MPI_COMM_WORLD,
249                 &status);
250         printf("Process %d > %f\n", 3, temp);
251 #else
252   MPI_Recv(&temp, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD,
253      &status);
254         MPI_Recv(&temp, 1, MPI_FLOAT, 2, 0, MPI_COMM_WORLD,
255      &status);
256         MPI_Recv(&temp, 1, MPI_FLOAT, 3, 0, MPI_COMM_WORLD,
257      &status);
258 #endif
259     } else {
260         MPI_Send(&dot, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
261     }
262     _ampi_on_return
263 }  /* Print_results */
```

Bibliography

[1] Eclipse C/C++ Development Tooling (CDT). http://www.eclipse.org/projects/project.php?id=tools.cdt.

[2] G. Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *CGO*, pages 1–12. IEEE Computer Society, 2009.

[3] C.Barrett, A.Stump, and C.Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *WSMT*, 2010.

[4] The LLVM Team. Clang: a C Language Family Frontend for LLVM. `http://clang.llvm.org/`, 2012.

[5] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[6] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard — Version 3.0*. 2012.

[7] I. Foster. *Designing and building Parallel programs*. Addison-Wesley, 1995.

[8] G.Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *ISCGP*, CGO '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[9] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky. Formal analysis of MPI-based Parallel programs. *CACM*, 54(12):82–91, 2011.

[10] 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. *CACM*, 54(12):82–91, December 2011.

[11] G.R.Luecke, Y.Zou, J.Coyle, J.Hoekstra, and M.Kraeva. Mpi-check: a tool for checking fortran 90 mpi programs. In *Concurrency and Computation: Pratice and Experience*, pages 93–100, 2003.

[12] G.R.Luecke, Y.Zou, J.Coyle, J.Hoekstra, M.Kraeva, and H.Chen. Deadlock Detection in MPI Programs. In *Concurrency and Computation: Pratice and Experience*, pages 911–932, 2002.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable Parallel Programming with the message passing interface*, volume 1. MIT press, 1999.

[14] T. Hilbrich and J. Protze. MPI Runtime Error Detection with MUST and MARMOT. In *VI-HPS Tuning*. ZIH, Technische Universitat Dresden, September 2010.

[15] Tobias Hilbrich, Martin Schulz, Bronis R. Supinski, and Matthias S. Muller. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In *Tools for High Performance Computing 2009*, pages 53–66. Springer Berlin Heidelberg, 2010.

[16] K. Honda, E. Marques, F. Martins, N. Ng, V. Vasconcelos, and N. Yoshida. Verification of MPI programs using session types. *Recent Advances in the Message Passing Interface*, pages 291–293, 2012.

[17] K. Honda, A. Mukhamedov, G. Brown, T.C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.

[18] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. *Programming Languages and Systems*, pages 122–138, 1998.

[19] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *SPPL*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM.

[20] B. Jacobs, J. Smans, and F. Piessens. Verifast. http://people.cs.kuleuven.be/ bart.jacobs/verifast/.

[21] J.S.Vetter and B.R.de Supinski, . Dynamic Software Testing of MPI Applications with Umpire. In *Supercomputing, ACM/IEEE Conference*, page 51, nov. 2000.

[22] B. Krammer, K. Bidmon, M.S. Muller, and M.M. Resch. MARMOT: An MPI analysis and checking tool. In F.J. Peters G.R. Joubert, W.E. Nagel and W.V. Walter, editors, *Parallel Computing Software Technology, Algorithms, Architectures and Applications*, volume 13 of *Advances in Parallel Computing*, pages 493 – 500. North-Holland, 2004.

[23] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. 1998.

[24] The LLVM Compiler Infrastructure. `http://llvm.org/`, 2012.

[25] E.R.B. Marques, F. Martins, V.T. Vasconcelos, N. Ng, and N. Martins. Towards deductive verification of MPI programs against session types. In *PLACES*, 2013. (to appear).

[26] N.D. Martins, C. Santos, E.R.B. Marques, F. Martins, and V.T. Vasconcelos. Especificaąão e Verificação de Protocolos para Programas MPI. In *INFORUM*, 2013. (in portuguese; to appear).

[27] M.Barnett, B.E.Chang, R.DeLine, B.Jacobs, and K.R.M.Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, pages 364–387. Springer, 2006.

[28] Microsoft. Hyper-v server, 2012. http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx.

[29] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1 – 40, 1992.

[30] L.De Moura and N.Bjørner. Z3: an efficient SMT solver. In *TPS*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] N. Ng, N. Yoshida, and K. Honda. Multiparty session c: safe parallel programming with message optimisation. In *ICOMCP*, TOOLS'12, pages 202–218, Berlin, Heidelberg, 2012. Springer-Verlag.

[32] Nicholas Ng, N. Yoshida, O. Pernet, R. Hu, and Y. Kryftis. Safe parallel programming with session java. In *COORDINATION*, COORDINATION'11, pages 110–126, Berlin, Heidelberg, 2011. Springer-Verlag.

[33] R.P.Feynman P, R.B Leighton, M. Sands, and E.M. Hafner. The Feynman Lectures on Physics; Vol. I. *American Journal of Physics*, 33:750, 1965.

[34] P.S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[35] P.Cuoq, F.Kirchner, N.Kosmatov, V.Prevosto, J.Signoles, and B.Yakobowski. Frama-c: a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] P.Krusina and G.R.Luecke. MPI-CHECK for C/C++ MPI Programs. Iowa State University, USA, November 2003.

[37] S.Aananthakrishnan, G.Bronevetsky, and G.Gopalakrishnan. Hybrid approach for data-flow analysis of MPI programs. In *ICS*, pages 455–456. ACM, 2011.

[38] Martin Schulz and Bronis R. de Supinski. PNMPI tools: a whole lot greater than the sum of their parts. In *Conference on Supercomputing*, pages 30:1–30:10, New York, NY, USA, 2007. ACM.

[39] S.F. Siegel and G. Gopalakrishnan. Formal Analysis of Message Passing. In *VMCAI*, volume 6538 of *LNCS*, pages 2–18, 2011.

[40] S.F. Siegel, A. Mironova, G.S. Avrunin, and L.A. Clarke. Combining symbolic execution with model checking to verify Parallel numerical programs. *ACM TSEM*, 17(2):10:1–10:34, May 2008.

[41] S.F. Siegel and L.F. Rossi. Analyzing BlobFlow: A Case Study Using Model Checking to Verify Parallel Scientific Software. In *EuroPVM/MPI*, volume 5205 of *LNCS*, pages 274–282, 2008.

[42] Stephen F. Siegel and Timothy K. Zirkel. In *PPoPP*, pages 309–310, New York, NY, USA, 2011. ACM.

[43] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: a tool for model checking MPI programs. In *PPoPP*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[44] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *HPCNSA*, November 2010.

[45] M. Weiser. Program slicing. In *Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[46] Xtext—language development made easy! `http://www.eclipse.org/Xtext/`.