15

Protocol-Driven MPI Program Generation

Nicholas Ng and Nobuko Yoshida

Imperial College London, UK

Abstract

This chapter presents Parameterised Scribble (Pabble), an extension of the Scribble language to capture scalable protocols, and a top-down, code generation framework of Message-Passing Interface (MPI) programs.

The code generation process begins with defining a Pabble protocol for the topology of the MPI application. An MPI parallel program skeleton is automatically generated from the protocol, which can then be merged with code kernels defining their behaviours. The merging process is fully automated through the use of an aspect-oriented compilation tool.

Pabble protocols are parameterised over the number of roles at runtime, and are grounded on theories of parameterised multiparty session types (MPST) where valid Pabble protocols can ensure safety and progress of communication in the generated MPI programs. Using the framework, programmers only need to supply the intended Pabble protocol and provide code kernels to obtain parallelised programs. Since the skeleton generation and the merging process are automatic, the framework not only simplifies the development of MPI programs, the output programs are efficient and scalable MPI applications, that are guaranteed, free from communication mismatch, type errors or deadlocks by construction, improving productivity of programmers.

15.1 Introduction

The Message Passing Interface (MPI) [8] is the de-facto standard for parallel programming on high-performance computing systems. Despite the advances in novel techniques and models such as the Partitioned Global Address Space

(PGAS) used by X10 [3, 10, 17] for simplifying parallel programming, MPI is still by far the most widely used parallel programming library in the scientific community. However, parallel programming with the MPI library is a well-documented difficult task, in which reasoning about interactions between distributed processes is difficult at scale, and communication mismatches are amongst the most common pitfalls by MPI users [6].

To apply behavioural types in safe, scalable parallel programming, this chapter presents a parallel programming workflow based on a protocol language Pabble, which we will explain in more details in Section 15.5. Figure 15.1 shows the overview of our approach, and this chapter explains the core use case of the approach highlighted in the figure. A Pabble protocol is an abstract representation of the communication topology, or *parallel communication patterns* of a parallel application. We consider every application a coupling between sequential, computation code that defines functional behaviours of processes in the application, and a communication topology that connects the processes together as a coherent application. Hence, to build a parallel application, we first define the communication protocol, written in Pabble. A valid Pabble protocol is guaranteed free of interactions and patterns that introduce communication errors and deadlocks. The Pabble protocol is used to generate an annotated MPI program backbone (Section 15.6), specifying the interactions between parallel processes. Based on the Pabble protocol, computation kernels are written in C language (C99), using queues to pass data locally between the kernels. The kernels are then merged with the MPI backbone by LARA [2], an aspect-oriented compilation tool, to transform the backbone and the kernels into a complete MPI application (Section 15.7).

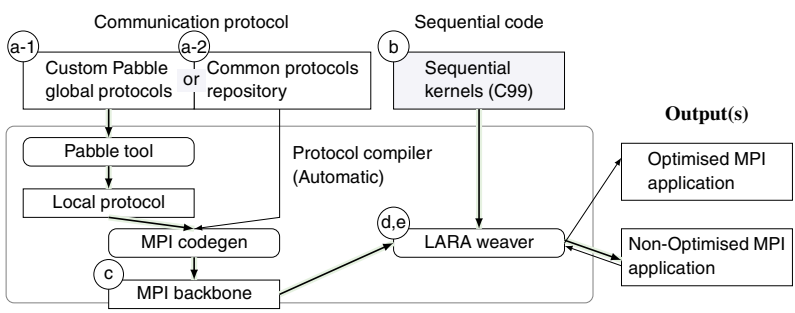

Figure 15.1 Pabble-based MPI program generation workflow (core flow highlighted).

In addition to the merge, LARA can also perform pragma directed optimisations on the source code to overlap communication and computation, improving the runtime performance. The details of the optimisations, rigorous evaluations of the approach, and a pre-generated repository of common protocols are omitted from this chapter, but can be found in the original paper [12].

15.2 Pabble: Parameterised Scribble

In this section we introduce *Parameterised Scribble* (Pabble) [14, 15], a developer friendly notation for specifying application level interaction protocol based on the theory of *parameterised* multiparty session types [5]. As the name suggests, Pabble is a parametric evolution of Scribble [16, 18], which itself is based on the theory of multiparty session types [1, 9]. We begin with an example Scribble protocol to explain the basic syntax of Pabble and the Scribble family of protocol languages, and why parameterisation is important for protocols describing scalable, parallel program topologies.

Scribble

```
module example;
global protocol Ring(role Worker1, role Worker2, role Worker3) {
  rec LOOP {
    Data(T) from Worker1 to Worker2;
    Data(T) from Worker2 to Worker3;
    DataLast(T) from Worker3 to Worker1;
    continue LOOP; }
}
```

Listing 15.1 Ring protocol in Scribble.

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

This greatly enhances the expressive power and modularity of the protocols. Here 'parameterised' refers to the number of participants in a role that can be changed by parameters.

Pabble

```
module example;
const N = 3;
global protocol Ring(role Worker[1..N]) {
  rec LOOP {
    Data(T) from Worker[i:1..N-1] to Worker[i+1];
    DataLast(T) from Worker[N] to Worker[1];
    continue LOOP; }
}
```

Listing 15.2 Parametrised Ring protocol in Pabble.

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

Pabble (at Worker)

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

The above code shows the *local protocol* of `Ring`, which is a localised version of Listing 15.2 at the `Worker` role. It represents the `Worker[1..N]` parameterised role, and corresponds to multiple endpoints in the same logical grouping. A Pabble local protocol is automatically generated from its global protocol following the projection algorithm in [14], and programmers only need to define the global protocol to use Pabble for MPI development.

Above servers as a primer on the Pabble language, sufficient for our introductory example; Later, the full syntax and explanations of the Pabble language will be given in Section 15.5.

15.3 MPI Backbone

A typical MPI program follows a Single Program, Multiple Data (SPMD) parallel programming model, where a single source code is executed by multiple parallel processors. This model shares a lot of similarities with the parameterised local protocols in Pabble which groups together similar roles in a single protocol, except that local protocols can be generated from global protocols which are easier to express overall communication or topologies. As a running example, we use the Pabble protocol presented earlier to demonstrate the framework and implement a ring accumulator that calculates a sum of values from each `Worker` and distribute to all.

C/MPI Backbone

```
1  int main(int argc, char *argv[])
2  { MPI_Init(&argc, &argv);
3    MPI_Comm_rank(MPI_COMM_WORLD, &meta.pid);
4    MPI_Comm_size(MPI_COMM_WORLD, &meta.nprocs);
5  #pragma pabble type T
6    typedef void T; ⇒ typedef double T;
7    MPI_Datatype MPI_T; ⇒ MPI_Datatype MPI_T = MPI_DOUBLE;
8
9    T *bufData_r, *bufData_s;
10   /** Other buffer declarations **/
11   /** Definitions of cond0, cond1, ... **/
12 #pragma pabble kernel Init ⇒ init(Init, "input.txt")
13 #pragma pabble predicate Ring
14   while (1) { ⇒ while(iter())
15     if (cond0) { /*if Worker[i:2..N]*/
16       bufData_r = (T *)calloc(meta.buflen(Data), sizeof(T));
17       MPI_Irecv(bufData_r, meta.buflen(Data), MPI_T, /*Worker[i-1]*/...);
18       MPI_Wait(&req[0], &stat[0]);
19       pabble_recvq_enqueue(Data, bufData_r);
20 #pragma pabble kernel Data ⇒ accumulate(Data);
21     }
22     if (cond1) { /*if Worker[i:1..(N-1)]*/
23 #pragma pabble kernel Data  ⇒ accumulate(Data);
24       bufData = pabble_sendq_dequeue();
25       MPI_Isend(bufData, meta.buflen(Data), MPI_T, /*Worker[i+1]*/...);
26       MPI_Wait(&req[1], &stat[1]);
27       free(bufData);
28     }
29     // Similarly for DataLast between Worker[1] and Worker[N]
30     MPI_Finalize();
31   }
32   return EXIT_SUCCESS; }
```

Listing 15.3 MPI backbone generated from the `Ring` protocol.

15.3.1 MPI Backbone Generation from `Ring` Protocol

Based on the Pabble `Ring` protocol in the introduction, our code generation framework generates an *MPI backbone* code (e.g. Listing 15.3). First it automatically generates *local protocols* from a global protocol as an intermediate step to make MPI code generation more straightforward. The MPI backbone generation procedure is described in details later in Section 15.6, here we focus on the generated MPI backbone code output.

An MPI backbone is a C99 program with boilerplate code for initialising and finalising the MPI environment of a typical MPI application (lines 2–4 and 30 respectively), and MPI primitives for message passing (e.g. `MPI_Isend`/`MPI_Irecv`[1]). Therefore the MPI backbone realises the interaction between participants as specified in the Pabble protocol, without supporting any specific application functionality. The backbone has three kinds of `#pragma` annotations as placeholders for kernel functions, types and program logic. The annotations are explained in Section 15.7.1. The boxed code in Listing 15.3 represents how the backbone are converted to code that calls the kernel functions in the MPI program.

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

Following the type declarations, other variable declarations including the buffers (line 9), and their allocation and deallocation are managed by the backbone. They are generated as guarded blocks of code, which come directly from the local protocol. lines 15–21 shows a guarded receive that correspond to `if Worker[i:2..N] Data(T)from Worker[i-1]` in the protocol and lines 22–28 for `if Worker[i:1..N-1] Data(T)to Worker[i+1]`.

Given the MPI backbone, we can then implement computation kernels for the MPI program.

15.4 Computation Kernels

Computation kernels are C functions that describe the algorithmic behaviour of the application. Conceptually, each message interaction defined in Pabble (e.g. `Label(T)from Sender to Receiver`), and – through the automatic MPI backbone generation – the MPI backbone, can be associated to a kernel by its label (e.g. `Label`).

[1] We use `MPI_Isend`/`MPI_Irecv` with `MPI_Wait` in place of the equivalent `MPI_Send`/`MPI_Recv` respectively. To simplify presentation we write `MPI_Send`/`MPI_Recv` in the rest of the chapter.

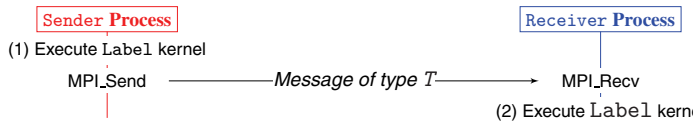

Figure 15.2 Global view of `Label(T)from Sender to Receiver;`.

Figure 15.2 shows how kernels are invoked in a message-passing statement between two processes named `Sender` and `Receiver` respectively. Since a message interaction statement involves two participants (e.g. `Sender` and `Receiver`), the kernel serves two purposes: (1) produce a message for sending and (2) consume a message after it has been received. The two parts of the kernel are defined in the same function, but runs on the sending process and the receiving process respectively. The kernels are top-level functions and do not send or receive messages directly through MPI calls. Instead, messages are passed between kernels and the MPI backbone (derived from the Pabble protocol) via a queue API: in order to send a message, the producer kernel (e.g. (1)) of the sending process enqueues the message to its send queue; and a received message can be accessed by a consumer kernel (e.g. (2)), dequeuing from its receive queue. This allows the decoupling between computation (as defined by the kernels) and communication (as described in the MPI backbone).

15.4.1 Writing a Kernel

We now explain how a user writes a kernel file, which contains the set of kernel functions related to a Pabble protocol for an application. As an example, we implement accumulator in a ring topology below.

A minimal kernel file must define a variable `meta` of `meta_t` type, which contains the process id (i.e. `meta.pid`), total number of spawned processes (i.e. `meta.nprocs`) and a callback function that takes one parameter (message label) and returns the send/receive size of message payload (i.e. `unsigned int meta.bufsize(int label)`). The `meta.buflen` function returns the buffer size for the MPI primitives based on the label given, as a lookup table to manage the buffer sizes centrally. Process id and total number of spawned processes will be populated automatically by the backbone code generated. The kernel file includes the definitions of the kernel functions, annotated with pragmas, associating the kernels with message labels. The pragmas that are allowed are detailed in Section 15.7. The kernels can use file (i.e. `static`) scope variables for local data storage. Our ring

accumulator kernel file starts with the following declarations for local data and `meta`:

Kernel file header User C Kernel

```
typedef struct {
  double* values; int N;
} local_data_t;
static local_data_t *local;

unsigned int buflen(int label) { return 1; } // 1-size buffer for all.
meta_t meta = {/*pid*/0, /*nprocs*/1, MPI_COMM_NULL, &buflen};
```

15.4.1.1 Initialisation

Most parallel applications require explicit partitioning of input data. In these cases, the programmer writes a kernel function for partitioning, such that each participant has a subset of the input data. Input data are usually partitioned with a layout similar to the layout of the participants. In our ring accumulator example, the processes are arranged linearly, and the input file contains an array of at least `meta.nprocs` elements, so `meta.nprocs` initial values are read into the `local->values` array. In our example initialisation function below, we also set the current accumulated value to be our initial value of `local->values[meta.pid]`.

Kernel: Init User C Kernel

```
#pragma pabble kernel Init
void init(int id, const char *filename)
{ FILE *fp = fopen(filename, "r");
  local = (local_data_t *)malloc(sizeof(local_data_t));
  local->values = NULL; local->N = 0;
  ... // allocate etc.
  int nprocs = meta.nprocs; // Number of processes (known at runtime).
  for (int i=0; i<nprocs; i++)
    fscanf(fp, "%f", &local->values[i]); // Copy data to local
  fclose(fp); local->N = nprocs;
  local->accumulated = local->values[meta.pid]; /* initial value on proc */
}
```

15.4.1.2 Passing data between backbone and kernel through queues

The kernels are `void` functions with at least one parameter, which is the label of the kernel. Inside the kernel, no MPI primitive should be used to perform message passing. Data received from another participant or data that need to be sent to another participant can be accessed using a receive queue

Kernel: Data User C Kernel

```
#pragma pabble kernel Data
void accumulate(int id)
{ double *rcvd_val; // Ptr to received value (temp).
  if (!pabble_recvq_isempty() && pabble_recvq_top_id() == id) {
    rcvd_val = (double *)pabble_recvq_dequeue(); // allocated by backbone
    local->accumulated += *rcvd_val;
  } else { // Allocate and send value
    accumulated_val = (double *)calloc(meta.buflen(id), sizeof(double));
    *accumulated_val = local->accumulated
    pabble_send_enqueue(id, accumulated_val);
  }
}
```

and send queue. Consider the following kernel for the label `Data` in the ring accumulator example:

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

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

15.4.1.3 Predicates

A predicate kernel is similar to a normal `void` kernel, but with a function signature that returns an `int` (as a boolean), it is used as a conditional variable, where the value of the variable is determined by the body of the kernel. In the `iter()` predicate kernel, we use the number of processes to determine when the ring protocol has completed a cycle (i.e. executed `meta.nproc` times) and terminate the `while`-loop.

Kernel: Ring User C Kernel

```
#pragma pabble predicate Ring
int iter() { static int i = 0; return i++ < meta.nprocs }
```

Aftere writing the computation kernels, we can then use the framework to merge the MPI backbones with the computation kernels, and we get a complete MPI program. The resulting MPI program is shown in Listing 15.3 (boxed code).

15.5 The Pabble Language

In this section, we present more details of the Pabble language, including its syntax, and the well-formedness conditions (i.e. syntactic restrictions to ensure protocol correctness) of the language.

15.5.1 Global Protocols Syntax

Figure 15.3 lists the core syntax of Pabble, which consists of two protocol declarations, *global* and *local*. A global protocol is declared with the protocol name (str denotes a string) with role and group parameters followed by the body G. Role R is a name with argument expressions. The argument expressions are ranges or arithmetic expressions h, and the number of arguments corresponds to the dimension of the array of roles: for example, `Worker[1..4][1..2]` denotes a 2-D array with size 4 and 2 in the two dimensions respectively, forming a 4-by-2 array of roles.

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

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

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

Apart from specifying ranges by constants, ranges can also be specified using expressions. Expression e consists of operators for numbers, logarithm, left and right logical shifts (`<<`, `>>`), numbers, variables (i, j, k), and constants (`M`, `N`). Constants are either *bound* outside the protocol declaration or are left *free* (unbound) to represent an arbitrary number. As in [11], when the constants are bound, they are declared by numbers outside the protocol, e.g. `const N = 10` or lower and upper bounds, e.g. `const N = 1..10`. We also allow leaving the declaration *free* (unbound), e.g. `const N`, as a shorthand to represent an arbitrary constant with lower and upper bounds `0` and `max` respectively, i.e. `const N = 0..max`, where `max` is a special

Global Pabble

```
global protocol str(para) { G }
```

Parameter

para	::=	`role` R_d`, ...,`	Role declaration
		`group` *str* `= {`R_d`, ...}, ...`	Group declaration

Global protocol body

G	::=	$l(T)$ `from` R `to` R`;`	Interaction
	|	`choice at` R `{` G_1 `}` `or ... or {` G_N `}`	Choice
	|	`foreach (`b`) {` G `}`	Foreach
	|	`allreduce` $op_c(T)$`;`	Reduction
	|	`rec` l `{` G `}`	Recursion
	|	`continue` l`;`	Continue
	|	G G	Sequential composition

Payload type

T	::=	`int` | `float` | ...	Data types

Expression

e	::=	e *op* e	Binary expressions
	|	*num*	Integers
	|	$i, j, k, \ldots$ | `N`	Variables, constants
op	::=	op_c | `-` | `/` | `%` | `<<` | `>>` | `log` | ...	Binary operations
op_c	::=	`+` | `*` | ...	Commutative operations

Role

R_d	::=	*str*	Role declaration
	|	*str*`[`e`..`e`]...[`e`..`e`]`	Param. role declaration
R	::=	*str*	Roles
	|	*str*`[`h`]...[`h`]`	Param. roles
	|	`All`	*All* group role
h	::=	b | e	Role parameter
b	::=	$i : e..e$	Binding range

Local Pabble

```
local protocol str at Rd(para) { L }
```

Local protocol body

L	::=	`[if` R`]` $l(T)$ `from` R`;`	(Conditional) Receive
	|	`[if` R`]` $l(T)$ `to` R`;`	(Conditional) Send
	|	`choice at` R `{` L_1 `}` `or ... or {` L_N `}`	Choice
	|	`foreach (`b`) {` L `}`	Foreach
	|	`allreduce` $op_c(T)$`;`	Reduction
	|	`rec` l `{` L `}`	Recursion
	|	`continue` l`;`	Continue
	|	L L	Sequential composition

Figure 15.3 Pabble syntax.

value representing the maximum possible value or practically unbounded. Binding range expression b takes the form of $i : e_1..e_n$ which means i is ranged from e_1 to e_n. Binding variables always bind to a range expression and not individual values. Indices in a Pabble protocol must be bound with

the binding range expression, the details are omitted here, please see *indices well-formed conditions* in [14].

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

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

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

There are restrictions on the indices on such as relative indices calculations and index bounds presented below. The restrictions ensure termination of the projection algorithm and safety of the communication topology at runtime.

15.5.1.1 Restriction on constants

In Pabble protocols, constants can be defined by

(1) A single numeric value (`const N = 3`); or
(2) Lower and upper bound constraints not involving the `max` keyword; or
(3) A range defined with the `max` keyword.

(1) sets a fixed value to a constant, as exemplified in Listing 15.2. (2) gives runtime constants a lower bound and an upper bound, e.g. the number of processes spawned in a scalable protocol, which is unknown at design time and will be defined and immutable once the execution begins. To ensure Pabble protocols are communication-safe in all possible values of constants, we must ensure that all parameterised role indices stay within their declared range.

Such conditions prevent sending or receiving from an invalid (non-existent) role which will lead to communication mismatch at runtime.

The following explains how to determine whether the protocol will be valid for all combinations of constants:

```
1 const M = 1..3;
2 const N = 2..5;
3 global protocol P(role R[1..N]) {
4   T from R[i:1..M] to R[i+1];
5 }
```

The basic constraints from the constants are:

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

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

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

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

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

By comparing this against the basic constraints on the constants, we can check that not all outcomes belong to the regions and thus this is not a communication-safe protocol (an example of a unsafe case is $\texttt{M} = 3$ and $\texttt{N} = 2$). On the other hand, if we alter line 4 to `T from R[i:1..N-1] to R[i+1];`, the constraints are unconditionally true and so we can guarantee all combinations of constants `M` and `N` will not cause communication errors.
(3) is a special case of (2), where the upper bound of a constant is set to the `max` keyword. We write `const N = 0..max` to represent a range without upper bound, here it means the constant `N` can be any integer value larger than 1. Since it is not possible to enumerate all values of `N`, we apply a more restrictive constraint on the expressions, allowing only range calculation that uses addition or subtractions on integers (e.g. `i+1`).

15.5.2 Local Protocols

As mentioned in Section 15.2, *local protocols* are localised versions of the global protocols at each role, and are used directly for skeleton generation. They are generated from a global protocol by a projection algorithm detailed

in [14]. Local protocol L consists of the same syntax of the global type except the input from R (receive) and the output to R (send). The main declaration `local protocol` *str* `at` R_e `(...) { ` L `}` means the protocol is located at role R_e. We call R_e *the endpoint role*. In Pabble, multiple local protocol instances can reside in the same parameterised local protocol. This is because each local protocol is a local specification for a participant of the interaction. When there are multiple participants with a similar interaction structure that fulfil the same *role* in the protocol, such as the `Worker` role from our `Ring` example from the introduction, the participants are grouped together as a single parameterised role. The local protocol for a collection of participants can be specified in a single parameterised local protocol, using *conditional statements* on the role indices to capture corner cases. For example, in a general case of a pipeline interaction, all participants receive from one neighbour and send to another neighbour, except the first participant which initiates the pipeline and is only a sender and the last participant which ends the pipeline and does not send. In these cases we use conditional statements to guard the input or output statements. To express conditional statements in local protocols, `if` R may be prepended to an input or output statement. `if` R input/output statement will be ignored if the local role does not match R. More complicated matches can be performed with a parameterised role, where the role parameter range of the condition is matched against the parameter of the local role. For example, `if Worker[1..3]` will match `Worker[2]` but not `Worker[4]`. It is also possible to bind a variable to the range in the condition, e.g. `if Worker[i:1..3]`, and `i` can be used in the same statement.

15.6 MPI Backbone Generation

Below we explain how Pabble statements are translated into MPI blocks.

15.6.1 Interaction

An interaction statement in a Pabble protocol is projected in the local protocol as two parts: receive and send. The correspondence is shown in Figure 15.4.

The first line of the local protocol shows a receive statement, written in Pabble as `if P[dstId] from P[srcId]`. The statement is translated to a block of MPI code in 3 parts. First, memory is dynamically allocated for the receive buffer (line 2), the buffer is of `Type` and its size fetched from the function `meta.bufsize(Label)`. The function is defined in the kernels and

Global Protocol

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

→ Projected Local Protocol

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

Interaction Output C/MPI Backbone

```
1  if (meta.pid == role_P(dstIdx)) {
2   buf = (Type *)calloc(meta.bufsize(Label), sizeof(Type));
3   MPI_Recv(buf, meta.bufsize(Label), MPI_Type, role_P(srcIdx), Label, ...);
4   pabble_recvq_enqueue(Label, buf);
5  #pragma pabble kernel Label
6  }
7  if (meta.pid == role_P(srcIdx)) {
8  #pragma pabble kernel Label
9   buf = pabble_recvq_dequeue();
10  MPI_Send(buf, meta.bufsize(Label), MPI_Type, dstIdx, Label, ...);
11  free(buf);
12 }
```

Figure 15.4 Pabble interaction statement and its MPI backbone.

returns the size of message for the given message label. Next, the program calls `MPI_Recv` to receive a message (line 3) from participant `P[srcRole]` in Pabble. `role_P(srcIdx)` is a lookup macro from the generated backbone to return the process id of the sender. Finally, the received message, stored in the receive buffer `buf`, is enqueued into a global receive queue with `pabble_recvq_enqueue()` (line 4), followed by the pragma indicating a kernel of label `Label` should be inserted. The block of receive code is guarded by an if-condition, which executes the above block of MPI code only if the current process id matches the receiver process id.

The next line in the local protocol is a send statement, converse of the receive statement, written as `if P[srcIdx] Label(Type)to P[dstIdx]`. The MPI code begins with the pragma annotation, then dequeuing the global send queue with `pabble_sendq_dequeue()` and sends the dequeued buffer with `MPI_Send`. After this, the send buffer, which is no longer needed, is deallocated. The block of send code is similarly guarded by an if-condition to ensure it is only executed by the sender. By allocating memory before receive and deallocating memory after send, the backbone manages memory for the user systematically. Since the protocol and the backbone makes no assumption about memory management on user's computation kernel, this mechanism helps the separation of concern between the protocol (i.e. the generated backbone) and the user kernels, and leaves open the possibility of optimal memory management during merge without breaking existing kernels.

15.6.2 Parallel Interaction

A Pabble parallel interaction statement is written as `Label(Type)from P[i:1..N-1] to P[i+1]`, meaning all processes with indices from `1` to `N-1` send a message to its next neighbour. `P[1]` initiates sending to `P[2]`, and `P[2]` receives from `P[1]` then sends a message to `P[3]`, and so on. As shown in Figure 15.5, the local protocol encapsulates the behaviour of all `P[1..N]` processes, and the statement is realised in the local as conditional receive followed by a conditional send, similar to ordinary interaction. The difference is the use of a range of process ids in the condition, and *relative* indices in the sender/receiver indices. The generated MPI code makes use of expression with `meta.pid` (current process id) to calculate the relative index.

15.6.3 Internal Interaction

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

Global Protocol

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

→ Projected Local Protocol

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

Parallel Interaction Output C/MPI Backbone

```
1  if (role_P(2)<=meta.pid&&meta.pid<=role_P(N)) {
2    buf = (Type *)calloc(meta.bufsize(Label), sizeof(Type));
3    MPI_Recv(..., /*prevRank:*/ meta.pid-1, Label, ...);
4    pabble_recvq_enqueue(Label, buf);
5  #pragma pabble kernel Label
6  }
7  if (role_P(1)<=meta.pid&&meta.pid<=role_P(N-1)) {
8  #pragma pabble kernel Label
9    buf = pabble_sendq_dequeue();
10   MPI_Send(..., /*nextRank:*/ meta.pid+1, Label, ...); free(buf);
11 }
```

Figure 15.5 Pabble parallel interaction statement and its MPI backbone.

Global/Local Protocol

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

Output C/MPI Backbone

```
1 #pragma pabble Internal
```

Figure 15.6 Pabble internal interaction statement and its MPI backbone.

15.6.4 Control-flow: Iteration and For-loop

`rec` and `foreach` are iteration statements. Specifically `rec`/`continue` is recursion, where the iteration conditions are not specified explicitly in the protocol, and translates to `while`-loops. The loop condition is the same in all processes, otherwise be known as *collective loops*. The loop generated by `rec` has a `#pragma pabble predicate` annotation, so that the loop condition can be later replaced by a kernel (see Section 15.7.1).

The `foreach` construct, on the other hand, specifies a counting loop, iterating over the integer values in the range specified in the protocol from the lower bound (e.g. `0`) to the upper bound value (e.g. `N-1`). This construct can be naturally translated into a C `for`-loop.

15.6.5 Control-flow: Choice

Conditional branching in Pabble is performed by label branching and selection. We use the example given in Figure 15.8 to explain. The deciding process, e.g. `P[master]`, makes a choice and executes the statements in the selected branch. Each branch starts by sending a unique label, e.g. `Branch0`, to the decision receiver, e.g. `P[worker]`. Hence for a well-formed Pabble protocol, the first line of each branch is from the deciding process to the same process but using a different label.

Note that the decision is only known between the two processes in the first statement, and other processes should be explicitly notified or use broadcast to propagate the decision. The MPI backbone is generated with a different structure as the local protocol. First, the MPI backbone contains an outer

Global/Local Protocol

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

Iteration Output C/MPI Backbone

```
1 #pragma pabble predicate LoopName
2 while (1) {
3   ... }
```

Global/Local Protocol

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

Foreach Output C/MPI Backbone

```
1 for (int i=0; i<=N-1; i++) {
2   ...
3 }
```

Figure 15.7 Control-flow: Pabble iteration statements and their corresponding MPI backbones.

Global Protocol

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

→ Projected Local Protocol

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

Choice Output C/MPI Backbone

```
1  if (rank==role_P(master)) { // Choice sender
2  #pragma pabble predicate Branch0
3    if (1) {
4      // Block of send.
5      MPI_Send(..., MPI_Type, role_P(worker), Branch0, ...);
6    } else
7  #pragma pabble predicate Branch1
8    if (1) { ... }
9  } else { // Choice receiver
10   MPI_Probe(role_P(master), MPI_ANY_TAG, comm, &status);
11   switch (status.MPI_TAG) {
12     case Branch0:
13     // Ordinary block of recv.
14     if (rank==role_P(worker)) {
15       MPI_Recv(..., MPI_Type, role_P(master), Branch0, ...);
16       pabble_recvq_enqueue(Branch0, buf); }
17       ... break;
18 #pragma pabble Branch1
19     case Branch1: ...
20   }
21 }
```

Figure 15.8 Control-flow: Pabble choice and its corresponding MPI backbone.

if-then-else, splitting the deciding process (lines 1–9) and the decision receiver (lines 9–21). In the deciding process, a block of if-then-else-if code is generated to perform a send with different label (called MPI tag), e.g. line 5. This statement is generated with all the queue and memory management code as described above for ordinary interaction statements. Each of the if-condition is annotated with `#pragma pabble predicate BranchLabel`, so that the conditions can be replaced by predicate kernels (see Section 15.7.1). For the decision receiver, `MPI_Probe` is used to peek the received label, then the `switch` statement is used to perform the correct receive (for different branches).

15.6.6 Collective Operations: Scatter, Gather and All-to-all

Collective operations are written in Pabble as multicast or multi-receive message interactions. While it is possible to convert these interactions into

multiple blocks of MPI code following the rules in Figure 15.7 (e.g. loop through receivers for scatter), we take advantage of the efficient and expressive collective primitives in MPI. Figure 15.9 shows the conversion of Pabble statements into MPI collective operations. We describe only the most generic collective operations, i.e. `MPI_Scatter`, `MPI_Gather` and `MPI_Alltoall`.

Translating collective operations from Pabble to MPI uses both global Pabble protocol statements and local protocol. If a statement involves the `All` role as sender, receiver or both, it is a collective operation. Figure 15.9 shows that translated blocks of MPI code do not use `if`-statements to distinguish between sending and receiving processes. This is because collective

Global Protocol

```
Label(Type) from P[rootRole] to All; // One-to-Many: (a) Scatter
Label(Type) from All to P[rootRole]; // Many-to-One: (b) Gather
Label(Type) from All to All;        // Many-to-Many: (c) All-to-All
```

Collective operation: (a) Scatter Output C/MPI Backbone

```
1  rbuf = (Type *)calloc(meta.buflen(Label), sizeof(Type));
2  #pragma pabble kernel Label
3  sbuf = pabble_sendq_dequeue();
4  MPI_Scatter(sbuf, meta.buflen(Label), MPI_Type,
5              rbuf, meta.buflen(Label), MPI_Type, role_P(rootRole), ...);
6  pabble_recvq_enqueue(Label, rbuf);
7  #pragma pabble kernel Label
8  free(sbuf);
```

Collective operation: (b) Gather Output C/MPI Backbone

```
1  rbuf = (Type *)calloc(meta.buflen(Label)*meta.nprocs, sizeof(Type));
2  #pragma pabble kernel Label
3  sbuf = pabble_sendq_dequeue();
4  MPI_Gather(sbuf, meta.buflen(Label), MPI_Type,
5             rbuf, meta.buflen(Label), MPI_Type, role_P(rootRole), ...);
6  pabble_recvq_enqueue(Label, rbuf);
7  #pragma pabble kernel Label
8  free(sbuf);
```

Collective operation: (c) All-to-All Output C/MPI Backbone

```
1  rbuf = (Type *)calloc(meta.buflen(Label)*meta.nprocs, sizeof(Type));
2  #pragma pabble kernel Label
3  sbuf = pabble_sendq_dequeue();
4  MPI_Alltoall(sbuf, meta.buflen(Label), MPI_Type,
5               rbuf, meta.buflen(Label), MPI_Type, ...);
6  pabble_recvq_enqueue(Label, rbuf);
7  #pragma pabble kernel Label
8  free(sbuf);
```

Figure 15.9 Collective operations: Pabble collectives and their corresponding MPI backbones.

primitives in MPI are executed by *both* the senders and the receivers, and the runtime decides whether it is a sender or a receiver by inspecting the `rootRole` parameter (which is a process rank) in the `MPI_Scatter` or `MPI_Gather` call. Otherwise the conversion is similar to their point-to-point counterparts in Figure 15.4.

15.6.7 Process Scaling

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

A Pabble protocol for MPI code generation can contain any number of constant values (e.g. `const M = 10`), which are converted in the backbone as C constants (e.g. `#define M 10`), but it can use at most one *scalable constant* [13], and will scale with the total number of spawned processes. A scalable constant, defined in Section 15.5.1.1 as constant type (3), is written:

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

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

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

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

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

15.7 Merging MPI Backbone and Kernels

15.7.1 Annotation-Guided Merging Process

To combine the MPI backbone with the kernels, our aspect-oriented design-flow inserts kernel function calls into the MPI backbone code. The insertion

points are realised as `#pragmas` in the MPI backbone code, generated from the input protocol as placeholders where functional code is inserted. There are multiple types of annotations whose syntax is given as:

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

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

15.7.2 Kernel Function

`#pragma pabble kernel Label` defines the insertion point of kernel functions in the MPI backbone code. `Label` is the label of the interaction statement, e.g. `Label(T)from Sender to Receiver`, and the annotation is replaced by the kernel function associated to the label `Label`. Programmers must use the same pragma to manually annotate the implementation of the kernel function. The first row in Table 15.1 shows an example.

15.7.3 Datatypes

`#pragma pabble type TypeName` annotates a generic type name in the backbone, and also annotates the concrete definition of the datatype in the kernels. In the second row of Table 15.1, the C datatype `T` is defined to be void since the protocol does not have any information to realise the type. The kernel defines `T` to be a concrete type of `double`, and hence our tool transforms the `typedef` in the backbone into `double` and infers the corresponding `MPI_Datatype` (MPI derived datatypes) to the built-in MPI integer primitive type, i.e. `MPI_Datatype MPI_T = MPI_DOUBLE`. From the

Table 15.1 Annotations in backbone and kernel

	Generated MPI backbone	User supplied kernel	Merged code

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

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

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

given type we can also generate MPI datatypes for structures of primitive types, e.g. `struct { int x, int y, double m }` is transformed to its MPI-equivalent datatype.

15.7.4 Conditionals

`#pragma pabble predicate Label` annotates predicates, e.g. loop conditions or if-conditions, in the backbone. Since a Pabble communication protocol (and transitively, the MPI backbone) does not specify a loop condition, the default loop condition is `1`, i.e. always true. This annotation introduces a way to insert a conditional expression defined as a kernel function. It precedes the `while`-loop, as shown in the third row of Table 15.1, to label the loop with the name `Label`. The kernel function that defines expressions must use the same annotation as the backbone, e.g. `#pragma pabble predicate Label`. After the merge, this kernel function is called when the loop condition is evaluated.

15.8 Related Work

The general approach of describing parallel patterns and reusing them with different computation modules can date back to [4] by Darlington et al., where parallel patterns are described as higher order *skeleton* functions, written in a functional language. Parallel applications are implemented as functions that combine with the skeletons and transformed. Their system targets specialised parallel machines, and our approach targets MPI, a standard for parallel programming in a range of hardware configurations. The approach, also known as *algorithmic skeleton frameworks* for parallel programming, is surveyed in [7]. Some of these tools also target MPI for high-level structured parallel programming, and only works with a limited set of parallel patterns. Our code generation workflow based on Pabble supports generic patterns written in Pabble and guarantees communication safety in the generated MPI code.

15.9 Conclusion

In this chapter we presented a protocol-based workflow for constructing safe and efficient parallel applications. The framework consists of two parts, a safe-by-construction parallel interaction backbone, generated from the Pabble protocol language, and an aspect-oriented compilation workflow

to mechanically insert computation code into the backbone. Our approach simplifies parallel programming by making use of parallel communication patterns, described with our Pabble protocol description language, and building independent kernel code around the patterns as sequential C code. This approach is flexible, where multiple sets of kernels can share a common parallel communication pattern, since the computation and the communication are maintained separately.

Acknowledgements This work is supported EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1 and EP/N027833/1; and by EU FP7 612985 (UpScale).

References

[1] L. Bettini, M. Coppo, L. DAntoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR 2008*, volume 5201 of *LNCS*. Springer, 2008.

[2] J. a. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov. LARA: an aspect-oriented programming language for embedded systems. In *AOSD '12*. ACM Press, 2012.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*. ACM Press, 2005.

[4] J. Darlington, A. Field, P. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE'93*, 1993.

[5] P.-M. Denielou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised Multiparty Session Types. *Logical Methods in Computer Science*, 8(4):1–46, October 2012.

[6] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *SE-HPCS '05*. ACM Press, 2005.

[7] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, 2010.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[9] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. *JACM*, 63(1):9:1–9:67, 2016.
[10] J. K. Lee and J. Palsberg. Featherweight X10. In *PPoPP '10*. ACM Press, 2010.
[11] J. Magee and J. Kramer. *Concurrency – state models and Java programs (2. ed.).* Wiley, 2006.
[12] N. Ng, J. G. Coutinho, and N. Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. volume 9031 of *LNCS*. Springer, 2015.
[13] N. Ng and N. Yoshida. Pabble: Parameterised Scribble for Parallel Programming. In *PDP 2014*, pages 707–714, 2014.
[14] N. Ng and N. Yoshida. Pabble: parameterised Scribble. *SOCA*, 9(3–4), 2015.
[15] Pabble project on GitHub. `https://github.com/pabble-lang`
[16] Scribble homepage. `http://scribble.org/`
[17] X10 homepage. `http://x10-lang.org`
[18] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble Protocol Language. In *TGC 2013*, volume 8358 of *LNCS*. Springer, 2013.