Protocols by Default

Safe MPI Code Generation based on Session Types

Nicholas Ng, Jose G. F. Coutinho, and Nobuko Yoshida

Imperial College London

Abstract. This paper presents a code generation framework for type-safe and deadlock-free Message Passing Interface (MPI) programs. The code generation process starts with the definition of the global topology using a protocol specification language based on parameterised multiparty session types (MPST). An MPI parallel program backbone is automatically generated from the global specification. The backbone code can then be merged with the sequential code describing the application behaviour, resulting in a complete MPI program. This merging process is fully automated through the use of an aspect-oriented compilation approach. In this way, programmers only need to supply the intended communication protocol and provide sequential code to automatically obtain parallelised programs that are guaranteed free from communication mismatch, type errors or deadlocks. The code generation framework also integrates an optimisation method that overlaps communication and computation, and can derive not only representative parallel programs with common parallel patterns (such as ring and stencil), but also distributed applications from any MPST protocols. We show that our tool generates efficient and scalable MPI applications, and improves productivity of programmers. For instance, our benchmarks involving representative parallel and application-specific patterns speed up sequential execution by up to 31 times and reduce programming effort by an average of 39%.

1 Introduction

Message Passing Interface (MPI) [24] library is the most widely used API standard for programming high performance parallel applications using the message passing paradigm. MPI is a relatively low-level programming library, and according to a survey [12] the most common MPI programming error is the communication mismatch between senders and receivers. This type of error directly leads to lost messages, communication deadlocks and subtle calculation errors.

In this work, rather than directly verifying the correctness of a given piece of MPI code, we explore a compilation approach that automates the *generation* of a communication deadlock-free and type-safe MPI program, using as inputs the sequential code defining the algorithmic behaviour of the application and a language-independent interaction protocol. Code generation using abstractions of common parallel programming patterns (also known as *algorithmic skeletons*) is a well-developed field, and [15, 30] survey a number of existing tools and frameworks supporting high-level structured parallel programming. More recently, this code generation technique has been used to teach undergraduate students parallel programming, and is reported to reduce programming errors [14, 41], showing how accessible the technique is.

Our code generation framework is based on a novel approach which, in addition to common parallel programming patterns, supports general or application-specific communication patterns. The framework is driven by a theoretically-founded protocol language called Pabble [26]. Pabble is a protocol language based on the theory of multiparty session types (MPST) [19]. It is specifically designed for expressing indexed and grouped processes interaction patterns in parallel algorithms based on the theories in [11], and distributed applications including web services [25].

Writing a program using the Pabble language starts with the specification of the *global* communication protocol, which is translated automatically to endpoint protocols. The endpoint protocols are localised projection versions of the global protocol. Our previous work type-checks C distributed parallel applications written with a customised API [27] or MPI [26] by a programmer against endpoint protocols. This paper presents the first session-based approach to automatically guarantee (by construction), type-safety, communication-safety (i.e. no communication mismatch) and deadlock-freedom for MPI applications.

Because of the expressiveness of parameterised MPST [10, 11], our compilation framework can support parallel algorithms included in the Dwarf benchmarks [2] (i.e. algorithmic methods that capture common pattern of communication and computation). We can generate safe MPI programs using not only fixed topologies such as pipelines or stencils, but also any well-formed Pabble protocols. As a portable standard, MPI is being adapted as a common interface to different kinds of programming models, including FPGAs [31], stream programming [22] and fault tolerant [13]. General MPI applications exhibit more complex communication patterns than well-known, connected topologies found in scientific computing. The generality of MPST can provide a more flexible pattern programming approach based on code generation. In addition, structured session types can guide the optimisation process using MPI immediate operators, without compromising the safety properties of the original code. Through our Pabble-based workflow, snippets of sequential code are automatically combined to generate a distributed memory parallel application, exploiting the parallelism of multiple nodes and increasing programming productivity and reusability: the use of design patterns means that programmers do not need to write an application from scratch, and can reuse the same protocols and/or sequential code according to their needs.

Pabble code generation workflow. Fig. 1 shows the overview of our approach. **(a)** Programmers decide which Pabble communication protocol to use for code generation: **(a-1)** If a standard protocol such as a ring, stencil or matrix is used, programmers can reuse a protocol from the Pabble repository so that they do not have to write Pabble, or **(a-2)** If programmers wish to use a more specific protocol which is not provided in Pabble repository, they can write the intended protocol. In this case, the tool automatically checks whether the protocol is well-formed or not; **(b)** As the second step, the programmer needs to write sequential computation code (kernels) in C99 and annotate their code with pragmas to link the kernels with the protocol specification; **(c)** The tool generates an MPI backbone from the Pabble protocol in **(a)**; **(d)** The kernels are automatically injected into the

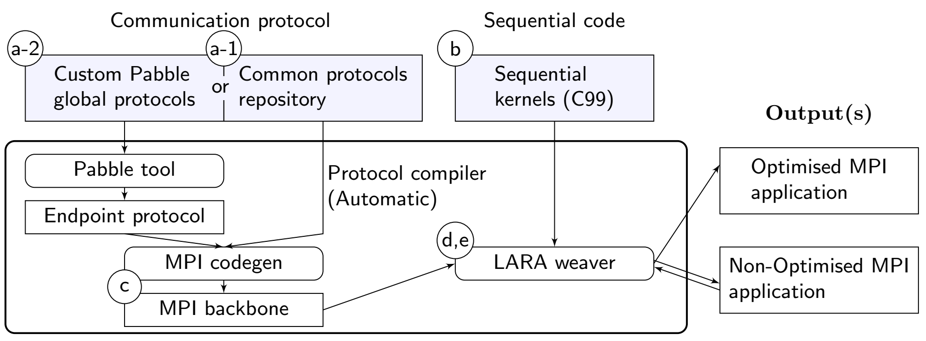

Fig. 1. Pabble-based code generation workflow. Shaded boxes indicate user inputs.

MPI backbone using the LARA [6] weaver, an aspect-oriented compilation tool, resulting in a complete MPI application **(e)** As part of the merging project, the LARA weaver can optionally perform optimisations against previously generated source code, such as overlapping communication and computation, to improve the runtime performance.

Challenges. The technical challenges of this work include bridging the gap between the high-level Pabble specification describing the global communication protocol, and the low-level C kernels and MPI calls that realise computation and communication, requiring several implementation details to be automatically inferred. We are cautious to avoid unnecessary assumptions between the Pabble specification and the C code defining the behaviour of the application, by providing a simple and minimally intrusive interface to interoperate between them. The use of session types to define communication patterns separately from computation means that data-dependent and non-deterministic protocols are not supported, but sufficient enough to generate safe representative algorithms (see Section 5).

Outline. Section 2 outlines the application development workflow through a running example; Section 3 explains the first of two stages of compilation, the generation of MPI backbone from protocol; Section 4 explains the second stage of compilation, merging the backbone with kernels and the optimisation; Section 5 gives a number of case studies including scientific computations and flexible grid computations, and performance evaluation of our framework showing the flexibility and productivity. The Pabble homepage [29] includes the code generation framework information, including the Pabble library and benchmark results.

2 Application Development Workflow

2.1 Interaction protocols with the Pabble protocol language

Pabble [26], or Parameterised Scribble [32],represents interaction types as parametric protocols, such that the protocols are scalable over the number of participants (i.e. compute nodes) given as parameters.

Listing 1 presents an example of a Pabble protocol which defines a 5-point stencil design pattern, where $N \times N$ processes are arranged in a 2-dimensional

Stencil Protocol

```
1  const N = 1..max;
2  global protocol Stencil(role P[1..N][1..N]) {
3   rec Steps {
4    LeftToRight(T) from P[r:1..N][c:1..N-1] to P[r][c+1];
5    RightToLeft(T) from P[r:1..N][c:2..N] to P[r][c-1];
6    UpToDown(T)  from P[r:1..N-1][c:1..N] to P[r+1][c];
7    DownToUp(T)  from P[r:2..N][c:1..N] to P[r-1][c];
8    continue Steps;
9   }
10 }
```

Listing 1. Pabble protocol for 5-point stencil.

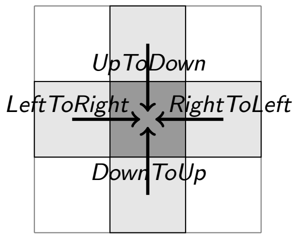

Fig. 2. Messages received by a process in a stencil protocol.

grid, and each participant exchanges messages with its 4 neighbours (except for edge participants). A Pabble protocol consists of a preamble and a definition. Line 1 defines `N` to be in the range between `1` and `max`, where `max` corresponds to the maximum integer. The concrete value of `N` is known only at run time, and stays constant in the duration of the instantiated protocol. `N` can be used in the protocol body as indices for role definition, which is the mechanism used by Pabble to support parameterised protocols. The protocol definition starts from Line 2, with the keywords `global protocol` followed by the protocol name `Stencil`. The parameters to the protocols are the role declarations, `role P[1..N][1..N]`, which declares a 2-dimensional role `P`, with $N \times N$ participants. Individual participants can be addressed by integer indices, e.g. `P[1][1]`, similar to an array access. A valid Pabble protocol ensures that all participants referenced in the protocol body are declared and within the index bounds ([26] provides a detailed list of well-formed conditions). For example, the following protocol is *not* well-formed because participants `P[5]` and `P[i+1]` are undefined when `i` is 3.

Non well-formed protocol

```
global protocol BadProtocol(role P[1..3]) {
  Msg(T) from P[1] to P[5];
  Msg(T) from P[i:1..3] to P[i+1]; }
```

Pabble protocols provide a guarantee of communication safety and deadlock freedom between participants in the protocol; this guarantee also extends to scalable protocols, where the number of participants are not known statically, and well-formed conditions ensure that the indexing of participants does not go beyond specified bounds. A Pabble protocol describes (1) the structured message interaction patterns of the application, and (2) the control-flow elements, excluding the logic related to actual computation, so that a Pabble protocol defining a parallel design pattern can be reused for different applications (see Section 5).

We provide a repository of common Pabble protocols describing common interaction patterns used by parallel applications. The `Stencil` protocol in Listing 1 is one example, and the other patterns in the repository include *ring pipeline*, *scatter-gather*, *master-worker* and *all-to-all*.

Our protocol body starts with a `rec` block, which stands for recursion, and is assigned with the label `Steps`. The recursion block does not specify the loop condition because a Pabble protocol only describes the interaction *structure* while implementation details are abstracted away. In the body of the recursion, we have

4 lines of interaction statements (Line 4-7), one for each direction. Interaction statements describe the sending of a message from one participant to another. For example, in Line 4 a message with label `LeftToRight` and with a generic payload type `T` is sent from `P[r:1..N][c:1..N-1]` to `P[r][c+1]`. The index expression `r:1..N` means that `r` is bound and iterated through the list of values in the range `1..N`, so the line encapsulates $N \times (N-1)$ individual interaction statements. The other interaction statements in Listing 1 can be similarly interpreted. Fig. 2 shows the messages received from neighbours for participant `P[2][2]` in a 3×3 grid, which is defined in the protocol as `role P[1..3][1..3]`.

2.2 Computation kernels

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

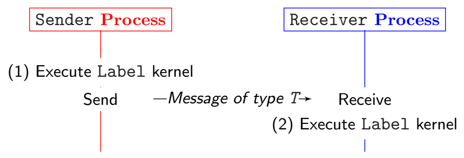

Fig. 3. Global view of `Label(T)from Sender to Receiver;`.

Fig. 3 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).

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

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

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

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

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

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

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

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

Each kernel has access to a send and receive queue local to the whole process, which holds pointers to the buffer to be sent and the buffer containing the received messages, respectively. The queues are the only mechanism for kernels to interface the MPI backbone. The simplest kernel is one that forwards incoming messages

Generated MPI Backbone

```
int main(int argc, char *argv[])
{ MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &meta.pid);
  MPI_Comm_size(MPI_COMM_WORLD, &meta.nprocs);
#pragma pabble type T
  typedef void T; ⇒ typedef double T;
  MPI_Datatype MPI_T; ⇒ MPI_Datatype MPI_T = MPI_DOUBLE;

  T *bufLeftToRight_r, *bufLeftToRight_s;
  /** Other buffer declarations **/
  /** Definitions of cond0, cond1, ... **/
#pragma pabble predicate Steps
  while (1) { ⇒ while(iter())
    if (cond0) { /*if P[i:0..(N-1)][j:1..(N-1)]*/
     bufLeftToRight_r = (T *)calloc(meta.buflen(LeftToRight), sizeof(T));
     MPI_Irecv(bufLeftToRight_r, meta.buflen(LeftToRight), MPI_T, /*P[i][(j-1)]*/...);
     MPI_Wait(&req[0], &stat[0]);
     pabble_recvq_enqueue(LeftToRight, bufLeftToRight_r);
#pragma pabble kernel LeftToRight ⇒ accumulate_LeftToRight(LeftToRight);
    }
    if (cond1) { /*if P[i:0..(N-1)][j:0..(N-2)]*/
#pragma pabble kernel LeftToRight  ⇒ accumulate_LeftToRight(LeftToRight);
     bufLeftToRight = pabble_sendq_dequeue();
     MPI_Isend(bufLeftToRight, meta.buflen(LeftToRight), MPI_T, /*P[i][(j+1)]*/...);
     MPI_Wait(&req[1], &stat[1]);
     free(bufLeftToRight);
    }
    /** similarly for RightToLeft, UpToDown and DownToUp **/
    MPI_Finalize();
  }
  return EXIT_SUCCESS; }
```

Listing 2. MPI backbone generated from the `Stencil` protocol.

from the receive queue directly to the send queue. In the above function, when the kernel function is called, it either consumes a message from the receive queue if it is not empty (i.e. after a receive), or produce a message for the send queue (i.e. before a send).

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

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

3 Compilation Step 1: Protocol to MPI backbone

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

3.1 MPI backbone generation from `Stencil` protocol

Based on the Pabble protocol (e.g. Listing 1), our code generation framework generates an *MPI backbone* code (e.g. Listing 2). First it automatically generates *endpoint protocols* from a global protocol as an intermediate step to make

MPI code generation more straightforward. For reference, Appendix A lists the endpoint protocol of the `Stencil` protocol in Listing 3.

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

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

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

3.2 MPI backbone generation from Pabble

Table 1 and 2 show how each Pabble construct is translated into MPI blocks for statements that involve P2P interactions and control-flow respectively (Appendix B lists additional cases, the internal iteration and choice constructs).

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

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

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

Table 1. Pabble interaction statements and their corresponding code.

1. Interaction

Global Protocol

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

Projected Endpoint Protocol

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

Generated MPI Backbone

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

2. Parallel interaction

Global Protocol

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

Projected Endpoint Protocol

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

Generated MPI Backbone

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

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

`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 in a systematic way.

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

3. Iteration and 4. For-loop. `rec` and `foreach` are iteration statements. Specifically `rec` is recursion, where the iteration conditions are not specified explicitly in the protocol, and translates to `while`-loops. The loop condition is the same in all processes. This may otherwise be known as *collective loops*. The loop generated

Table 2. Pabble statements and their corresponding code.

3. Iteration	**4. For-loop**
Global/Endpoint Protocol	Global/Endpoint Protocol
`rec LoopName { ... continue LoopName; }`	`foreach (i:0..N-1) { ... }`
Generated MPI Backbone	Generated MPI Backbone

`#pragma pabble predicate LoopName`
`while (1) {`
`  ... }`
	`for (int i=0; i<=N-1; i++) {`
	`  ...`
	`}`

by rec has a #pragma pabble predicate annotation, so that the loop condition can be later replaced by a kernel (see Section 4).

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

Table 3. MPI collective operations and their corresponding Pabble statements.

5. Scatter

Global Protocol

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

Generated MPI Backbone

```

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

6. Gather

Global Protocol

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

Generated MP Backbone

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

7. All-to-All

Global Protocol

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

Generated MPI Backbone

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

5. Scatter, 6. Gather and 7. All-to-all Collective operations are written in Pabble as multicast or multi-receive message interactions. While it is possible to convert these interactions into multiple blocks of MPI code following the rules in Table 2, we take advantage of the efficient and expressive collective primitives in MPI. Table 3 shows the conversion of Pabble statements into MPI collective operations. We describe only the most generic collective operations, i.e. `MPI_Scatter`, `MPI_Gather` and `MPI_Alltoall`.

Translating collective operations from Pabble to MPI considers both global Pabble protocol statements and endpoint protocol. If a statement involves the `__All` role as sender, receiver or both, it is a collective operation. Table 3 shows that translated blocks of MPI code do not use `if`-statements to distinguish between sending and receiving processes. This is because collective primitives in MPI are executed by *both* the senders and the receivers, and the runtime decides whether it is a sender or a receiver by inspecting the `rootRole` parameter (which is a process rank) in the `MPI_Scatter` or `MPI_Gather` call. Otherwise the conversion is similar to their point-to-point counterparts in Table 2.

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* [26]. A scalable constant is defined as:

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

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

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

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

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

4 Compilation Step 2: Aspect-Oriented Design-Flow

This section focuses on the final stage of our code generation framework, which merges two input components to derive the complete MPI program: (1) the communication safe MPI backbone derived automatically from a Pabble protocol (Section 3.1), and (2) the user supplied kernels capturing specific application functionality.

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

Our approach takes a similar path as OpenMP [8] and OpenACC [40], which parallelise sequential programs using non-invasive `#pragma` annotations. The difference is that while OpenMP operates on a shared memory architecture model and OpenACC operates via a host-directed execution (co-processor) model, our approach allows applications to target customised platform topologies defined by Pabble, since MPI works on both shared and distributed memory platforms.

LARA language. To support an automated merging process, our programming framework uses an aspect-oriented programming (AOP) language called LARA [6]. As far as we know, LARA is the only aspect-oriented approach that targets all stages of a development process allowing static code analysis and manipulation (e.g. source-level translation and code optimisation), toolchain execution (e.g. for design-space exploration) and application deployment (e.g. to extract dynamic behaviour). These various tasks, which are often performed manually and independently, can be described in a unified way as LARA aspects. These aspects can then drive LARA *weavers* to apply a particular strategy in a systematic and automated way. In our code generation framework, we use LARA's ability to analyse and manipulate C code to automate the merging process between the MPI backbone and the kernels sources (Section 4.1), and also to further optimise the MPI code by overlapping communication and computation (Section 4.2).

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

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

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

Table 4. 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())` `{ ... }`

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 4 shows an example.

Datatypes. `#pragma pabble type TypeName` annotates a generic type name in the backbone, and also annotates the concrete definition of the datatype in the kernels. In the second row of Table 4, the C datatype `T` is defined to be void since the protocol does not have any information to realise the type. The kernel defines `T` to be a concrete type of `double`, and hence our tool transforms the `typedef` in the backbone into `double` and infers the corresponding `MPI_Datatype` (MPI derived datatypes) to the built-in MPI integer primitive type, i.e. `MPI_Datatype MPI_T = MPI_DOUBLE`. Our tool also supports generating MPI datatypes for structures of primitive types, e.g. `struct { int x, int y, double m }` is transformed to its MPI-equivalent datatype.

Conditionals. `#pragma pabble predicate Label` is a pragma for annotating predicates, e.g. loop conditions or if-conditions, in the backbone. Since a Pabble communication protocol (and transitively, the MPI backbone) does not specify a loop condition, the default loop condition is `1`, i.e. always true. This annotation introduces a way to insert a conditional expression defined as a kernel function. It precedes the `while`-loop, as shown in the third row of Table 4, 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.

4.2 Performance optimisation for overlapping communication and computation by MPI immediate operators

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

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

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

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

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

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

Original

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

Optimised

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

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

Original

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

Optimised

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

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

5 Evaluation

In this section, we first demonstrate that our protocols can automatically generate MPI programs using different parallel patterns, including application-specific patterns (flexibility); and save efforts in the development of MPI applications (productivity and reusability). Then we measure the performance and efficiency of the generated MPI programs.

5.1 Productivity and reusability

Table 5. Comparing effort of implementing different applications using our framework.

	Protocol	Repo.	Dwarf	Pabble	Backbone	Kernels	Effort
heateq [3]	stencil	✓	SG	15	154	335	0.69
nbody	ring	✓	PM	15	93	228	0.71
wordcount	scatter-gather	✓		8	76	176	0.70
adpredictor [17]	scatter-gather	✓		8	76	182	0.71
montecarlo	scatter-gather	✓		8	76	70	0.48
montecarlo-mw	master-worker	✓		10	82	70	0.46
LEsovler [26]	wrapround mesh		SG	15	132	208	0.66
matvec	custom [28]		DM	15	130	117	0.41
fft64	6-step butterfly		S	11	64	134	0.68

Table 5 presents a comparison of different parallel algorithms developed using our approach. The second and third columns show the input Pabble protocol and whether it is available in our protocol repository. The Dwarf column denotes the categorisations of parallel computational and structural patterns defined in [2]; SG stands for 'Structured Grid', PM is 'Particle Methods'; DM is 'Dense Matrix'; and S is 'Spectral (FFT)'. The next three columns show lines of code in the input Pabble protocol, the generated backbone, and the input user kernel file. The final column shows the effort ratio of user written code against the total ($\frac{Kernels}{Backbone+Kernels}$ for protocols in repository or $\frac{Kernels+Pabble}{Backbone+Kernels}$). The higher the ratio, the more effort are needed to write an equivalent program from scratch.

heateq is an implementation of the heat equation based on [3], and uses the stencil protocol in our running example. nbody is a 2D N-body simulation implemented with a ring topology; it is optimised with the asynchronous messaging mode described in Section 4.2. wordcount is a simple application that counts the number of occurrences of each word in a given text, implemented using the scatter-gather pattern. adpredictor is an implementation of Microsoft's AdPredictor [17] algorithm for calculated click-through rate, also implemented in the same scatter-gather pattern, but with a different set of kernel functions. LEsolver is a linear equation solver parallelised with a custom wraparound mesh topology outlined in [26]. montecarlo is Monte-Carlo π simulation, implemented with two different patterns, scatter-gather and master-worker. A remarkable difference between the two patterns is that the former uses collective operations and all processes are involved in the main calculation, whereas with the master-worker pattern workers are coordinated by a central master process by P2P communication that does not perform the main calculation. Note that the kernels used for both implementations are the same (except with different kernel labels). matvec is matrix-vector multiplication parallelised using the `MatVec` protocol outlined in [28]. fft64 is an implementation of the Cooley-Tukey FFT between 64 processes using 6 steps of butterfly exchange between pairs of processes.

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

Our results show that our workflow saves development and debugging efforts for MPI parallel applications, especially for novice parallel programmers. The user can focus on developing and maintaining the functional behaviour of their application, knowing that the merging of updated kernels and the respective MPI backbones are correct.

5.2 Performance

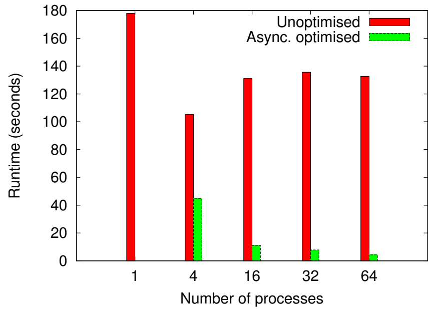

Fig. 4. N-body simulation (nbody).

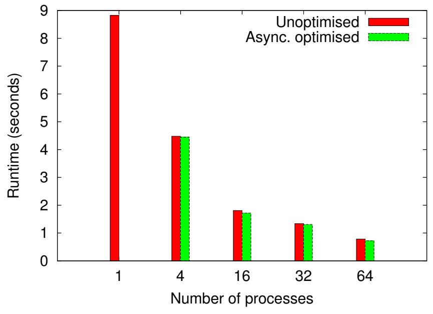

Fig. 5. Linear Equation Solver (LEsolver).

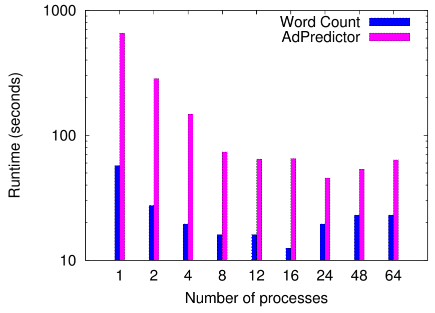

Fig. 6. Word Count (wordcount) and AdPredictor (adpredictor)

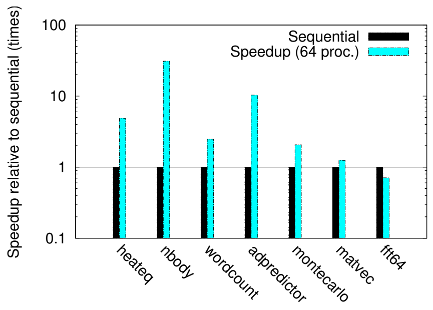

Fig. 7. Parallelisation speedup.

We evaluate our approach with 4 parallel applications which uses 3 different Pabble protocols. All implementations are evaluated on cx1[1], a general purpose multi-core cluster, and compiled with `icc` with optimisation level `-O3`, and tested using Intel's MPI library.

[1] http://www.imperial.ac.uk/ict/services/hpc/facilities

In Fig. 4 we compare the performance of nbody with and without asynchronous optimisation described in Section 4.2. The optimisation overlaps the main calculation with the communication, and the results show significant improvements over the unoptimised version. Fig. 5 presents the runtime performance of LEsolver which uses a custom wraparound mesh protocol with asynchronous optimisation. In comparison with nbody, the optimisation effect on LEsolver has less impact. This is partly because the asynchronous kernel implemented by nbody is more complex than the kernel implemented by LEsolver, so the time spent on communication is dominant. The asynchronous kernel in LEsolver also represents a smaller proportion of the total computations, hence it has the less effect on the overall runtime.

Fig. 6 shows that the two implementations – wordcount and adpredictor – both of which use the scatter-gather pattern and a different set of kernels follow a similar trend in scalability, which is dependent on the size of the input data.

Fig. 7 compares implementations in our framework running in 64 processes against sequential C versions. Results show speedup for all algorithms except fft64 due to communication overhead of the more complex butterfly topology.

6 Conclusion and Related Work

This paper presents a session-based framework for generating safe and scalable parallel applications based on flexible protocols that capture parallel design patterns. The framework consists of two parts: a compilation tool that derives a safe-by-construction parallel backbone from a Pabble protocol description, and an aspect-oriented compilation framework that mechanically inserts computation code into the backbone, and performs asynchronous optimisation. We demonstrate that our tool generates efficient and scalable MPI applications, and improves productivity of parallel application development with reusable patterns.

Pattern-based structured parallel programming. An algorithmic skeleton framework [15] is a high-level parallel programming approach which provides reusable parallel communication and interaction patterns programmers can parameterise to generate a specific parallel program. [15, 30] describe a number of tools that were developed in the past decade, and most of the tools target a similar set of skeletons, including farm (master-slave), pipeline, iterations and map. Our approach uses Pabble language to define the patterns of the skeletons, and is able to represent all common patterns above. In addition, custom patterns can be defined as Pabble protocols, and the formal MPST basis of Pabble ensures that valid protocols are guaranteed to be communication-safe and deadlock-free, and these properties hold for our generated MPI backbones (i.e. skeletons) by construction. Sklml [36], an implementation of P^3L language in OCaml supports the common patterns above but without extensibility. Recently, pattern programming was employed as a parallel programming teaching tool for undergraduate students [41, 14]. They used a pragma approach, and obtained positive feedback from the students. This motivated us to use the pragma annotation for sequential kernels for flexibility and preciseness. Other than teaching, most works in the field now target heterogeneous and embedded computing, for example, Fast Flow [20, 4] for CPU/GPU code generation, which can take advantage of the high-level abstraction of skeletons to target and coordinate between different hardware, each with different programming style.

Verification of MPI. The state-of-the-art in MPI program verification has been surveyed in [16]. Verification approaches in [16] are diverse and we focus on works that verify and detect deadlocks in MPI. ISP [39] is a runtime model checker based on in-situ partial order as a heuristic avoid state explosion. DAMPI [38] is a dynamic verifier for MPI based on ISP, but uses a distributed scheduling algorithm to allow scaling. Both of the tools suffer from interleaving explosion, where some execution schedule expands exponentially. MSPOE [33] improves on ISP's partial ordering algorithm to overcome the defect and detect orphaning deadlocks. All above tools are test-based and verify correctness with a fixed harness suite. MUST [18] is another scalable, MPI dynamic verification tool, which combines two MPI verification tools, Marmot [21] and Umpire [37], and overcomes scalability challenges in previous tools by comprehensive analysis of the semantics of the primitives. TASS [34] employs model checking and symbolic execution, but is also able to verify user-specified assertions for the interaction behaviour of the program and functional equivalence between MPI programs and sequential ones [35]. A user needs to specify the maximum number of processes (see [23] for further comparisons with protocol-based approaches). The concept of parallel control-flow graphs is proposed in [5] for static analysis of MPI programs, e.g., as a means to verify sender-receiver matching in MPI source code. An extension to dynamic analysis is presented in [1]. As far as we know, no other work focuses on communication deadlock-free MPI code generation based on types or backbones.

Session-based parallel programming. Session C [27] is a programming framework designed for parallel programming with multiparty session types. Users implement endpoint programs using session-based APIs and type-check them against its endpoint protocols. The framework differs from this work that it does not use a parameterised type for type-checking and the approach presented here are top-down code generation as opposed to type checking. Similarly, the work [26] introduces Pabble and type-checking MPI by Pabble, but it does not consider code generation. [23] proposes another type-checking tool for MPI based on multiparty session types. It treats a fine-grained index analysis by using VCC [7] where a program requires annotations for loops, which can be semi-automatically generated by the program annotator. All of these session-based works study type-checking endpoint programs written by developers. As far as we know, this work is the first to automatically generate a complete, communication-safe MPI code specified by a protocol specification language.

Future work includes extending our approach to generate MPI one-sided communication from the current point-to-point messaging abstraction in Pabble, which is more efficient in some categories of communication patterns; and supporting recursive, divide-and-conquer parallel pattern, which is possible with recent advances in session types on *sub-protocols* [9].

References

1. Aananthakrishnan, S., Bronevetsky, G., Gopalakrishnan, G.: Hybrid approach for data-flow analysis of MPI programs. In: ICS '13. pp. 455–456. ACM (2013)
2. Asanovic, K., Wawrzynek, J., Wessel, D., Yelick, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K.: A view of the parallel computing landscape. CACM 52(10), 56 (2009)
3. Balaji, P., Dinan, J., Hoefler, T., Thakur, R.: Advanced MPI Programming (Tutorial at SC'13). http://www.mcs.anl.gov/∼thakur/sc13-mpi-tutorial/
4. Boob, S., González-Vélez, H., Popescu, A.M.: Automated instantiation of heterogeneous fast flow CPU/GPU parallel pattern applications in clouds. In: PDP. pp. 162–169. IEEE (2014)
5. Bronevetsky, G.: Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In: CGO '09'. pp. 1–12. IEEE (2009)
6. Cardoso, J.a.M., Carvalho, T., Coutinho, J.G., Luk, W., Nobre, R., Diniz, P., Petrov, Z.: LARA: an aspect-oriented programming language for embedded systems. In: AOSD '12. pp. 179–190. ACM (2012)
7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs '09'. LNCS, vol. 5674, pp. 23–42 (2009)
8. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. Computational Science and Engineering 5(1), 46–55 (1998)
9. Demangeon, R., Honda, K.: Nested protocols in session types. In: CONCUR 2012. LNCS, vol. 7454, pp. 272–286. Springer (2012)
10. Deniélou, P.M., Yoshida, N.: Dynamic multirole session types. In: POPL '11. pp. 435–446. ACM (2011)
11. Denielou, P.M., Yoshida, N., Bejleri, A., Hu, R.: Parameterised Multiparty Session Types. Logical Methods in Computer Science 8(4), 1–46 (2012)
12. DeSouza, J., Kuhn, B., de Supinski, B.R., Samofalov, V., Zheltov, S., Bratanov, S.: Automated, scalable debugging of MPI programs with Intel Message Checker. In: SE-HPCS '05. pp. 78–82. ACM (2005)
13. Fagg, G.E., Dongarra, J.J.: FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1908, pp. 346–353. Springer (2000)
14. Ferner, C., Wilkinson, B., Heath, B.: Toward using higher-level abstractions to teach parallel computing. In: IPDPSW. pp. 1291–1296. IEEE (2013)
15. González-Vélez, H., Leyton, M.: A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. Softw. Pract. Exper. 40(12), 1135–1160 (2010)
16. Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., De Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs. CACM 54(12), 82–91 (2011)
17. Graepel, T., Candela, J.Q., Borchert, T., Herbrich, R.: Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsofts Bing Search Engine. In: ICML'10. pp. 13–20 (2010)
18. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In: SC '12. pp. 1–11. IEEE (2012)
19. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL '08. vol. 5201, pp. 273–284. ACM (2008)

20. Kolodziej, J., González-Vélez, H., Wang, L.: Advances in data-intensive modelling and simulation. Future Generation Comp. Syst. 37, 282–283 (2014)
21. Krammer, B., Bidmon, K., Müller, M.S., Resch, M.M.: MARMOT: an MPI analysis and checking tool. In: PARCO 2003. pp. 493–500 (2003)
22. Mancini, E.P., Marsh, G., Panda, D.K.: An MPI-stream hybrid programming model for computational clusters. In: CCGrid 2010. pp. 323–330. IEEE (2010)
23. Marques, E.R.B., Martins, F., Vasconcelos, V.T., Santos, C., Ng, N., Yoshida, N.: Protocol-based verification of C+MPI programs. DI-FCUL 13, University of Lisbon (2014)
24. Message Passing Interface. http://www.mcs.anl.gov/research/projects/mpi/
25. Ng, N., Yoshida, N.: Pabble: Parameterised Scribble. SOCA (2014), http://www.doc.ic.ac.uk/∼cn06/soca, to appear
26. Ng, N., Yoshida, N.: Pabble: Parameterised Scribble for Parallel Programming. In: PDP. pp. 707–714. IEEE (2014)
27. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe Parallel Programming with Message Optimisation. In: TOOLS 2012. LNCS, vol. 7304, pp. 202–218. Springer, Berlin, Heidelberg (2012)
28. Ng, N., Yoshida, N., Luk, W.: Scalable Session Programming for Heterogeneous High-Performance Systems. In: SEFM 2013 Collocated Workshops. LNCS, vol. 8368, pp. 82–98. Springer (2013)
29. Pabble project page. http://www.doc.ic.ac.uk/∼cn06/pabble
30. Rabhi, F., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer (2003)
31. Saldaña, M., Patel, A., Madill, C., Nunes, D., Wang, D., Chow, P., Wittig, R., Styles, H., Putnam, A.: MPI as a Programming Model for High-Performance Reconfigurable Computers. ACM TRETS 3(4), 1–29 (2010)
32. Scribble homepage. http://scribble.org/
33. Sharma, S., Gopalakrishnan, G., Bronevetsky, G.: A sound reduction of persistent-sets for deadlock detection in mpi applications. In: SBMF 2012. LNCS, vol. 7498, pp. 194–209. Springer (2012)
34. Siegel, S.F., Zirkel, T.K.: Collective assertions. In: VMCAI'11. pp. 387–402. LNCS (2011)
35. Siegel, S.F., Zirkel, T.K.: FEVS: A Functional Equivalence Verification Suite for High-Performance Scientific Computing. MSCS 5(4), 427–435 (2011)
36. Sklml webpage. http://sklml.inria.fr
37. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. In: SC '00. p. 51. IEEE (2000)
38. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for mpi programs. In: SC '10. pp. 1–10. IEEE (2010)
39. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPoPP '09. pp. 261–270. ACM (2008)
40. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC – First Experiences with Real-World Applications. In: Euro-Par 2012, LNCS, vol. 7484, pp. 859–870. Springer (2012)
41. Wilkinson, B., Villalobos, J., Ferner, C.: Pattern programming approach for teaching parallel and distributed computing. In: SIGCSE '13. pp. 409–414. ACM (2013)

A Appendix: Endpoint protocols

MPI processes are usually identified by their process id (called ranks), and the behaviour of each process is guarded by conditional tests on the ranks at runtime. Our tool automatically generates *endpoint protocols* as an intermediate step from a Pabble protocol to make MPI code generation more straightforward. The method, called *projection algorithm* in the literature [19], is described in detail in [26]. Projecting a Pabble protocol into an endpoint Pabble protocol preserves its succinctness, where each message interaction statements are guarded by a condition to check if a participant is part of a statement. Listing 3 shows the endpoint protocol of the stencil example. After generating the endpoint protocol, our tool automatically derives the MPI backbone from this description.

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

Listing 3. Automatically generated endpoint protocol based on the Stencil protocol.

B Appendix for Section 3.1

This appendix section presents omitted Pabble to MPI translation rules used in Section 3.1.

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.

Table 6. Pabble internal iteration and its corresponding code.

Internal interaction	
Internal() from __self to __self;	Global/Endpoint Protocol
#pragma pabble *Internal*	Generated MPI

Choice. Conditional branching in Pabble is performed by label branching and selection. We use the example given in Table 2 to explain. The deciding process, e.g. `P[master]`, makes a choice and executes the statements in the selected branch. Each branch starts by sending a unique label, e.g. `Branch0`, to the decision receiver, e.g. `P[worker]`. Hence for a well-formed Pabble protocol, the first line of each branch is from the deciding process to the same process but using a different label.

Note that the decision is only known between the two processes in the first statement, and other processes should be explicitly notified or use broadcast to propagate the decision. The MPI backbone is generated with a different structure as the endpoint protocol. First, the MPI backbone contains an outer if-then-else, splitting the deciding process (Line 1–9) and the decision receiver (Line 9–20). In the deciding process, a block of if-then-else-if code is generated to perform a send with different label (called MPI tag), e.g. Line 5. This statement is generated with all the queue and memory management code as described above for ordinary interaction statements. Each of the if-condition is annotated with `#pragma pabble predicate BranchLabel`, so that the conditions can be replaced by predicate kernels (see Section 4). For the decision receiver, `MPI_Probe` is used to peek the received label, then the `switch` statement is used to perform the correct receive (for different branches).

Choice

Global Protocol

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

Projected Endpoint Protocol

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

Generated MPI Backbone

```
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); switch (status.MPI_TAG) {
11      case Branch0:
12      // Ordinary block of recv.
13      if (rank==role_P(worker)) {
14        MPI_Recv(..., MPI_Type, role_P(master), Branch0, ...);
15        pabble_recvq_enqueue(Branch0, buf); }
16        ...  break;
17  #pragma pabble Branch1
18      case Branch1: ...
19    }
20  }
```