Scalable Session Programming for Heterogeneous High-performance Systems

Nicholas Ng, Nobuko Yoshida, and Wayne Luk

Imperial College London

Abstract. This paper introduces a programming framework based on the theory of session types for safe and scalable parallel designs. Session-based languages can offer a clear and tractable framework to describe communications between parallel components and guarantee communication-safety and deadlock-freedom by compile-time type checking and parallel MPI code generation. Many representative communication topologies such as ring or scatter-gather can be programmed and verified in session-based programming languages. We use a case study involving N-body simulation, dense and sparse matrix multiplication to illustrate the session-based programming style. Finally, we outline a proposal to integrate session programming with heterogeneous systems for efficient and communication-safe parallel applications by a combination of code generation and type checking.

1 Introduction

Software programs that utilises parallelism to increase performance is no longer an exclusive feature of high performance applications. Modern day hardware, from multicore processor in smartphones to multicore multi-graphics card gaming systems, all take advantage of parallelism to improve performance. Message-passing is a scalable programming model for parallel programming, where the user has to make communication between components explicit using the basic primitives of message *send* and *receive*.

However, writing correct parallel programs is far from straightforward – blindly parallelising components with data dependencies might leave the overall program in an inconsistent state; arbitrary interleaving of parallel executions combined with complex flow control can easily lead to unexpected behaviour, such as blocked access to resources in a circular chain (i.e. deadlock) or mismatched send-receive pairs. These unsafe communications are a source of non-termination or incorrect execution of a program. Thus tracking and avoiding communication errors of parallel programs is as important as ensuring their functional correctness.

This work focuses on a programming framework which can automatically ensure deadlock-freedom and communication-safety i.e. matching communication pairs, for message-passing parallel programs based on the theory of *session types* [6, 7]. Towards the end of this paper, we discuss how this session-based programming framework can fit in heterogeneous computing environments with

reconfigurable acceleration hardware such as Field Programmable Gate Arrays (FPGAs).

To illustrate how session types can track communication mismatches, consider the parallel program in Fig. 1 that exchanges two values between two processes.

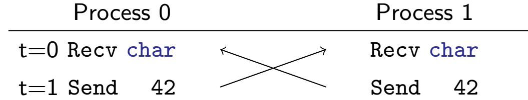

Fig. 1. Mismatched communication.

In this notation, the arrow points from the sender of the message to the intended receiver. Both `Process0` and `Process1` start by waiting to receive a value from the other processes, hence we have a typical deadlock situation.

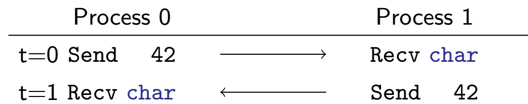

Fig. 2. Communication order swapped.

A simple solution is to swap the order of the receive and send commands for one of the processes, for example, Process 0, shown in Fig. 2.

However, the above program still has mismatched communication pairs and causing type error. Parallel programming usually involves debugging and resolving these communication problems, which is often a tedious task.

Using the session programming methodology, we can not only statically check that the above programs are incorrect, but can also encourage programmers to write safe designs from the beginning, guided by the information of types. Session types [6, 7] have been actively studied as a high-level abstraction of structured communication-based programming, which are able to accurately and intelligibly represent and capture complex interaction patterns between communicating parties.

The two examples above have session types shown in Fig. 3 and Fig. 4 respectively.

Process 0: Recv `char`; Send `int`
Process 1: Recv `char`; Send `int`

Fig. 3. Session types for original example.

Process 0: Send `int`; Recv `char`
Process 1: Recv `char`; Send `int`

Fig. 4. Session types for swapped example.

In the session types above, Send `int` stands for output with type `int` and Recv `int` stands for input with type `int`. The session types are used to check

that the communications between `Process 0` and `Process 1` are *incompatible* (i.e. incorrect) because one process must have a *dual type* of the other.

On the other hand, the following program is correct, having neither deadlock nor type errors, since it has a *mutually dual* session types shown on the right hand side:

In the session types theory, Recv `type` is dual to Send `type`, hence the type of `Process 0` is dual of the type of `Process 1`.

The above compatibility checking is simple and straightforward in the case of two parties. We can extend this idea to multiparty processes (i.e. more than two processes) based on multiparty session type theory [7]. Type-checking for parallel programs with multiparty processes is done statically and is efficient, with a polynomial-time bound with respect to the size of the program.

Below we list the contributions of this paper.

- Novel programming languages for communications in parallel designs and two session-based approaches to guarantee communication-safety and deadlock freedom (§ 2)
- Implementations of advanced communication topologies for parallel computer clusters by session types (§ 3)
- Case studies including N-body simulation, dense and sparse matrix multiplication to illustrate session programming for parallel computers (§ 4)

2 Session-based language design

2.1 Overview

As a language independent framework for communication-based programming, session types can be applied to different programming languages and environments. Previous work on Session Java (SJ) [8, 14] integrated sessions into the object-oriented programming paradigm as an extension of the Java language, and was applied to parallel programming [14]. Session types have also been implemented in different languages such as OCaml, Haskell, F#, Scala and Python. This section explains session types and their applications, focussing on an implementation of sessions in the C language (Session C) as a parallel programming framework. Amongst all these different incarnations of session types, the key idea remains unchanged. A session-based system provides (1) a set of predefined primitives or interfaces for session communication and (2) a session typing system which can verify, at compile time, that each program conforms to its session type. Once the programs are type checked, they run correctly without deadlock nor communication errors.

2.2 Multiparty session programming

Session C [12, 21] implements a generalised session type theory, *multiparty session types* (MPST) [7]. The MPST theory extends the original binary session types [6] by describing communications across multiple participants in the form of *global protocols*. Our development uses a Java-like protocol description language Scribble [5, 16] for describing the multiparty session types. Fig. 5 explains two design flows of Session C programming. In the type checking approach, the programmer writes a global protocol starting from the keyword `protocol` and the protocol name. In the first box of Fig. 5, the protocol named as `P` contains one communication with a value typed by `int` from participant `A` to participant `B`. For Session C implementation, the programmer uses the *endpoint protocol* generated by the projection algorithm in Scribble. For example, the above global protocol is projected to `A` to obtain `int to B` (as in the second box) and to `B` to obtain `int from A`. Each endpoint protocol gives a template for developing safe code for each participant and as a basis for static verification. Since we started from a correct global protocol, if endpoint programs (in the third box) conform to the induced endpoint protocols, it automatically ensures deadlock-free, well-matched interactions. This endpoint projection approach is particularly useful when many participants are communicating under complex communication topologies. Due to space limitation, this paper omits the full definition of global protocols, and will explain our framework and examples using only endpoint protocols introduced in the next subsection.

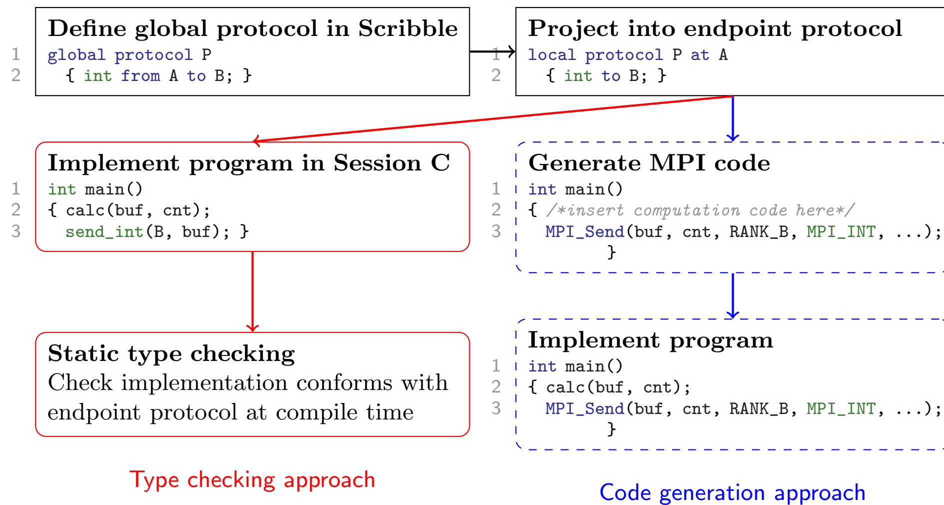

Fig. 5. Session C design flows.

2.3 Protocols for session communications

The endpoint protocols include types for basic message-passing and for capturing control flow patterns. We use the endpoint protocol description derived from Scribble to algorithmically specify high-level communication of distributed parallel programs as a library of network communications. A protocol abstracts away the contents but keeps the high level structures of communications as a series of type primitives.

The syntax of Scribble is described in details in [5,16], and can be categorised to three types of operations: *message-passing*, *choice* and *iteration*.

Message passing. It represents that messages (or data) being communicated from one process to another; in the language it is denoted by the statements `datatype to P1` or `datatype from P0` which stands for sending/receiving data of `datatype` to the participant identified by `P0`/`P1` respectively. Notice that the protocol does not specify the value being sent/received, but instead designate the datatype (which could be primitive types such as `int` or composite types), indicating its nature as a high-level abstraction of communication.

Choice. It allows a communication to exhibit different behavioural flows in a program. We denote a choice by a pair of primitives, `choice from` and `choice to`, meaning a distributed choice receiver and choice maker, respectively. A choice maker first decides a branch to take, identified by its `label`, and executes its associated block of statements. The chosen label is sent to the choice receiver, which looks up the label in its choices and execute the its associated block of statements. This ensures the two processes are synchronised in terms of the choice taken.

Iteration. It can represent repetitive communication patterns. We represent recursion by the `rec` primitive (short for recursion), followed by the block of statements to be repeated, enclosed by braces. The operation does not require communication as it is a local recursion. However two communicating processes have to ensure both of their endpoint protocols contains recursion, otherwise their protocols will not be compatible.

2.4 Session C

We present two approaches to session programming in C, using the Session C framework. The first approach is by type checking of user written code, using a simple session programming API we provided. The second approach is by MPI code generation from protocols.

Type checking approach In the type checking approach, a user implements a parallel program using the simple API provided by the library, following communication protocols stipulated in Scribble. Once a program is complete, the type checker verifies that the program code matches that of the endpoint protocol

description in Scribble to ensure that the program is safe. The core runtime API corresponds the endpoint protocol as described below.

Message passing primitives in Session C are written as `send_datatype(participant, data)` for message send, which is `datatype to participant` in the protocol, and `recv_datatype (participant, &data)` for message receive (`datatype from participant` in the protocol).

Choice in Session C is a combination of ordinary C control-flow syntax and session primitives. For a choice maker, each if-then or if-else block in a session-typed choice starts with `outbranch(participant, branchLabel)` to mark the beginning of a choice. `inbranch(participant, &branchLabel)` is a choice receiver, used as the argument of a switch-case statement, and each case-block is distinguished by the `branchLabel` corresponding to a choice in the `choice from` block in the protocol.

Iteration in Session C corresponds to `while` loops in C. As no communication is required, the implementation simply repeats a block of code consisting of above session primitives in a `rec` recursion block.

Code generation approach In the code generation approach, given a Scribble protocol, we generate an MPI parallel program skeleton. The program skeleton contains all the MPI code needed, the user inserts code that performs computation on the input data (e.g. for scientific calculation) between the MPI primitives, completing the program.

This approach is part of a larger extension of the Scribble language to support parameterised session types [2]. The extension, *Parameterised Scribble*, or Pabble [11], uses indices to parameterise participants. Participants can be defined and accessed in an array-like notation, in order to denote logical groupings of related participants. For example, a parallel algorithm that uses many parallel workers, can define a group of participants using `role participant[1..N]`, and a pipeline of message passing is written in Pabble as `datatype from participant [i:1..N-1] to participant[i+1]`. Pabble protocols can be written once, and a protocol with different number of participants can be instantiated by changing the value of `N`. MPI code generated from Pabble protocols can also take advantage of this feature and will be scalable over different number of processes.

These two approaches to session programming complement each other and cover different use cases: critical applications can use the type checking approach to ensure that the written program is communication and type safe; whereas scalable and parametric applications can use the MPI code generation capability to create communication safe and type safe parallel programs.

3 Advanced communication topologies for clusters

This section shows how session endpoint protocols introduced in § 2.3 can be used to specify advanced, complex communications for clusters. Consider a het-

erogeneous cluster with multiple kinds of acceleration hardware, such as GPUs or FPGAs, as Processing Elements (PEs). To allow a safe and high performance collaborative computation on the cluster, we can describe communications between PEs by our communication primitives. The PEs can be abstracted as small computation functions with a basic interface for data input and result output, hence we can easily describe high-level understanding of the program by the session types.

We list some widely used structured communication patterns that form the backbones of implementations of parallel algorithms. These patterns were chosen because they exemplify representative communication patterns used in clusters. Computation can interleave between statements if no conflict in the data dependencies exists. The implementation follows the theory of the optimisation for session types developed in [10], maximising overlapped messaging.

$\text{Node}_{0\leq i\leq n-1}$:
```
rec LOOP { // Repeat shifting ring
  datatype to Node[i+1]; // Next
  datatype from Node[i-1]; // Prev
LOOP }
```
Node_n:
```
rec LOOP { // Repeat shifting ring
  datatype from Node[N-1]; // Prev
  datatype to Node[0]; // Initial
LOOP }
```

Fig. 6. Endpoint protocols of Ring.

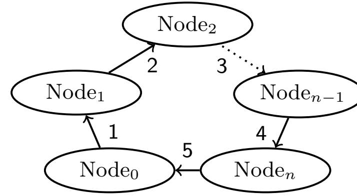

Fig. 7. n-node ring pipeline.

Ring topology. In a ring topology, as depicted in Fig. 7, processes or PEs are arranged in a pipeline, where the end of the node of the pipeline is connected to the initial node. Each of the connections of the nodes is represented by an individual endpoint session. We use N-body simulation as an example for ring topology. Note that the communication patterns between the middle $n-1$ Nodes are identical. The endpoint protocol is shown in Listing 6.

Map-reduce pattern. Map-reduce is a common scatter-gather pattern used to parallelise tasks that can be easily partitioned with few dependencies between the partitioned computations. The topology is shown in Fig. 9. It combines the map pattern which partitions and distributes data to parallel workers by a Master coordination node, and the reduce pattern which collects and combines completed results from all parallel workers. At the end of a map-reduce, the Master coordination node will have a copy of the final results combined into a single datum. All Workers in a map-reduce topology share a simple communication pattern, where they only interact with the Master coordination node. The Master node will have a communication pattern containing all known Workers.
The MPI operation `MPI_Alltoall` is a communication-only instance of the map-reduce pattern for all of the nodes, and only applies memory concatenation

to the collected set of data. Our endpoint types can represent this topology with more fine-grained primitives so that we can obtain performance gain by communication-computation overlap. Although collective operations are more efficient in cases where the implementations take advantage of the underlying architectures, fine-grained primitives can more readily allow partial data-structures to be distributed, without the need to create new copies of data or calculating offsets (as in `MPI_Alltoallv`) for transmission.

```
Master : rec LOOP {
           // Map phase
           datatype to Worker[0], Worker[1];
           // Reduce phase
           datatype from Worker[0], Worker[1];
         LOOP }
Worker_{0≤i≤n} : rec LOOP {
           datatype from Master; // Map phase
           datatype to Master; // Reduce phase
         LOOP }
```

Fig. 8. Endpoint protocols of Map-reduce.

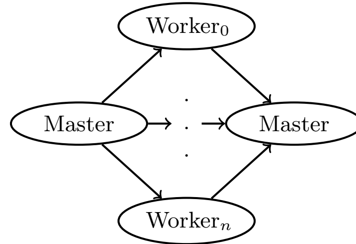

Fig. 9. Map-reduce pattern.

4 Case Studies

This section presents case studies of using Scribble protocols in parallel programming. All of these examples are representative patterns from common parallel patterns known as Dwarfs [1]. The Dwarf evaluation metric was proposed as a collection of high-level, core parallel communication patterns from important scientific and engineering methods. Each of these patterns is called a *Dwarf*, which represents one category of patterns and covers a broad range of concrete algorithm implementations. Dwarfs are used to evaluate our session-based protocol language and our programming methodology because they are not language or optimisation specific, being able to express the Dwarfs confirms that our approach is general enough to be extended to more practical use cases.

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

4.1 N-body simulation

We implemented a 2-dimension N-body simulation using a ring topology. Each `Worker` is initially assigned to a partition of the input data. In every round of the ring propagation, each `Worker` receives a set of partitioned input from a neighbour, and pipelines the input data received from the previous round to the

other neighbour. This propagation continues until the set of particles have been received by all `Workers` once. The algorithm will then perform one step of global update to calculate the new positions of the particles after one time step of the simulation.

Listing 1 is the protocol specification of the `Worker` participant of our N-body simulation implementation, and Listing 2 is the automatically generated endpoint version, both written in the syntax of parameterised Scribble.

```
1  global protocol Nbody(role Worker[0..N] {
2   rec RING {
3    // Workers 0 to N: Worker[i] -> [i+1]
4    int from Worker[i:1..N-1] to Worker[i+1];
5
6    // Data from Worker[N] -> [0]
7    int from Worker[N] to Worker[0];
8
9    continue RING;
10  }
11 }
```

Listing 1. Protocol of N-body simulation.

```
1  local protocol Nbody at Worker[0..N] {
2   rec RING {
3    // Workers 0 to N: Worker[i] -> [i+1]
4    if Worker[i:1..N] int from Worker[i-1];
5    if Worker[i:0..N-1] int to Worker[i+1];
6    // Data from Worker[N] -> r[0]
7    if Worker[0] int from Worker[N];
8    if Worker[N] int to Worker[0];
9    continue RING;
10  }
11 }
```

Listing 2. Worker endpoint of N-body protocol.

The block `rec RING { }` means recursion, and represents the repeating ring propagation in the algorithm. The line `if Worker[i:1..N] int from Worker[i-1]` stands for receiving a message from my previous neighbour `Worker[i-1]` with a message of type `int`, given that the current participant is one of `Worker[1]`, ..., or `Worker[N]`. The protocol generates MPI code equivalent to Listing 3.

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

Listing 3. MPI implementation of Worker endpoint.

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

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

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

4.2 Dense matrix-vector multiplication

Dense matrix-vector multiplication takes a $M \times N$ matrix and multiply it by a N dimensional vector to get a N dimensional vector result. The multiplication can be parallelised by partitioning the input matrix to N segments by row-wise block striping shown in Fig. 10 and distributed to N processes. Each process gets a copy of the vector, and each elements in the vector can be calculated by the processes in parallel.

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

```
1  global protocol DenseMatVec(role Worker[0..N]){
2    // Scatter Matrix A
3    foreach (i:1..N) {
4      LBound(int) from Worker[0] to Worker[i];
5      UBound(int) from Worker[0] to Worker[i];
6      Data(double) from Worker[0] to Worker[i];
7    }
8    // Scatter Vector B
9    (double) from Worker[0] to Worker[1..N];
10
11   // --- Perform calculation ---
12
13   // Gather data
14   (double) from Worker[1..N] to Worker[0];
15 }
```

Listing 4. Global protocol of dense matrix-vector multiplication.

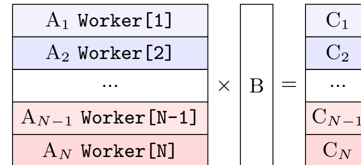

Fig. 10. Partitioning of input matrix.

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

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

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

```
1  double A[A_ROWS][A_COLS]; // Matrix A
2  double B[B_COLS]; // Vector B
3  double C[B_COLS]; // Partial result
4  ...
5  if (rank == 0) {
6    for (i = 1; i < size; i++) { // Calculate then send to each Worker
7      // Calculate LowerBound and UpperBound for each Worker
8      lbound = (i - 1) * partition_size;
9      ubound = lbound + partion_size;
10
11     MPI_Send(&lbound, 1, MPI_INT, Worker[i], LBound, ...);
12     MPI_Send(&ubound, 1, MPI_INT, Worker[i], UBound, ...);
13
14     // Send partition of matrix A
15     MPI_Send(&A[lbound][0], (ubound-lbound) * A_COLS, MPI_DOUBLE, Worker[i], Data, ...);
16   }
17 } else if (rank > 0) { // Workers, receiving work
18   MPI_Recv(&lbound, 1, MPI_INT, Worker[0], LBound, ...);
19   MPI_Recv(&ubound, 1, MPI_INT, Worker[0], UBound, ...);
20
21   MPI_Recv(&A[lbound][0], (ubound-lbound) * A_COLS, MPI_DOUBLE, Worker[0], Data, ...);
22 }
23
24 // All Workers receive the vector B
25 MPI_Bcast(&B, B_ROWS, MPI_DOUBLE, Worker[0], ...);
26 ...
27 // Calculate matrix multiplication
28 mat_vec_mul(A, B, lbound, ubound, C);
29 ...
30 // ... Gather results to Worker[0] ...
31 MPI_Gather(C, 1, MPI_DOUBLE, Result, 1, MPI_DOUBLE, Worker[0], ...);
```

Listing 5. MPI implementation of dense matrix-vector multiplication.

4.3 Sparse matrix-vector multiplication

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

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

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

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

The protocol to perform a sparse matrix-vector multiplication is shown in Listing 6. In the protocol, the partitioned matrix rows in CSR format are sent to each worker as separate row, col and values arrays (Line 3, 4 and 5). The N dimensional vector is then sent to all workers. The results of the calculation by each `Workers` are sent back to `Worker[0]` (Line 8).

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

Listing 6. Global protocol of sparse matrix-vector multiplication.

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

```
1  MPI_Comm_size(MPI_COMM_WORLD, &size);
2  int nr_of_rows = MATRIX_ROWS/size;
3  ...
4  MPI_Scatter(row_ptr, nr_of_rows, MPI_INT, ..);
5  ...
6  // calculate number of indices for each process
7  ...
8  MPI_Scatterv(col_ind, nr_of_elems, MPI_INT, ...);
9  MPI_Scatterv(vals, nr_of_elems, MPI_DOUBLE, ...);
10 ...
11 MPI_Bcast(vector, MATRIX_ROWS, MPI_DOUBLE, Worker[0], ...); // Distribute vector
12 ...
13 // Calculate matrix multiplication
14 mat_vec_mul(row_ptr, col_ind, vals, vector, C);
15 ...
16 MPI_Gather(C, 1, MPI_DOUBLE, Result, 1, MPI_DOUBLE, Worker[0], ...);
```

Listing 7. MPI implementation of sparse matrix-vector multiplication.

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

`nr_of_elems` is an array containing the number of elements to be sent to each worker. Since in a sparse matrix the number of elements in each row is not fixed,

the `nr_of_elements` array contains the number of matrix elements each worker receives. The indices of the array correspond to the rank of the workers and the column index `col_ind` is distributed to each worker process by `MPI_Scatterv` (Line 8), a variant of the `MPI_Scatter`, where the `v` stands for variable size as opposed to fixed size in `MPI_Scatter`. Similarly, the actual matrix element values are distributed to all workers by a call to `MPI_Scatterv` on Line 9, using the same `nr_of_elems` to specify the number of elements for each worker.

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

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

5 Related work and conclusion

ISP [3, 20] and the distributed DAMPI [19] are formal dynamic verifiers which apply model-checking techniques to standard MPI C source code to detect deadlocks using a test harness. The tool exploits independence between thread actions to reduce the state space of possible thread interleavings of an execution, and checks for potentially violating situations. TASS [3, 17] is another suite of tools for formal verification of MPI-based parallel programs by model-checking. It constructs an abstract model of a given MPI program and uses symbolic execution to evaluate the model, which is checked for a number of safety properties including potential deadlocks and functional equivalences.

Compared to the test-based and model-checking approaches which may not be able to cover all possible states of the model, the session type-based approach does not depend on external testing or extraction of models from program code for safety. It encourages designing communication-correct programs from the start, especially given the high level communication structure which session types captures.

Recent works [4, 9] used annotated MPI code and a software verifier to check the annotated MPI code for compliance against session types. Their bottom-up approach focusses on accurately representing MPI primitives and datatypes, whereas Session C treats them as high level abstractions, ignoring details such as send/receive data payload size.

There are a lot of challenges of verifying real-world MPI source code. MPI is a standardised and platform independent message-passing API, the ubiqui-

tous nature in supercomputing makes it a convenient abstraction layer between software and underlying hardware. In cases such as [15], it was used as a programming model for FPGAs. Hence its specification is intentionally vague, in order to allow different implementations to take advantage of any platform-specific optimisations. For example, there are a number of message transport modes such as the more commonly used `MPI_Send`/`MPI_Recv` (standard mode) or `MPI_Isend` /`MPI_Irecv` (immediate/non-blocking). The modes do not correspond directly to standard synchronous or asynchronous communication modes as one would expect. The different communication modes in MPI have subtle differences in their semantics. Care must be taken when making assumptions and correspondences with high-level Scribble protocols. In addition to standard point-to-point communication primitives, MPI also includes a huge number of primitives such as collective operators, topology construction and process management. A complete session type checking framework will be able to consider these additional information to extract the session types from the source code. Combining the flexibility of the host language (C) and the large number of MPI primitives makes our approach more challenging compared to model checking based approaches. This is because MPI model checkers work by observing the behaviours of the programs, which the same behaviour can be implemented in many different ways; whereas our type based approach requires us to understand the consequences of each primitive because we construct a type model without executing the program.

Furthermore, to apply our approach on low-level host languages, it is important to define a concise and simple correspondence between a Scribble and Pabble protocol to practical implementation, but offer enough flexibility to cope with conventional programs. This correspondence is important to both type checking and code generation: for type checking, the ability to support different programming styles would enable the type checker to check more existing code, and for code generation, the generated code will have a more natural style. For example, MPI uses process IDs (or ranks) to identify processes, and it is valid to perform numeric operations on the ranks to efficiently calculate target processes. A more concrete example is instead of conventional conditional statements, `MPI_Send(buf , cnt, MPI_INT, rank%2 ? rank+1: rank-1, ...);` may be used and the process ID, `rank`, is being used as a boolean to perform a choice, thus a straightforward analysis of `rank` usages would not be sufficient. These are valid programs that exploit the C language features and will require much more extensive analysis.

This paper is an extension of our previous works on Session C [12,13]. In both of the works, parametric protocols and MPI code generation were not explored, this work is a short insight into the benefits of using parametric protocols and potentials of integrating with specialised accelerators, as the framework was evaluated on [18], a heterogeneous cluster with FPGAs.

6 Future work

Integration with heterogeneous workflow. Immediate future works includes refining our MPI code generation tool to better integrate with APIs of specialised hardware. This includes streamlining the data received/sent from MPI directly into input/output buffers of acceleration hardware.Tighter integration between MPI and acceleration hardware will achieve better overall performance of the heterogeneous system.

Type-directed optimisations. Extending our type checker to support inferring parameterised MPST from MPI code is a prerequisite for type-directed optimisations. Once parameterised MPST can be extract from MPI code, Session C framework can then extend the support of asynchronous message optimisation [10] described in Session C framework [12] to expressive parameterised protocols. The theoretical and engineering challenges of this future work will be keeping type checking process decidable and representing most, if not all, of the common MPI primitives in Scribble.

Assertion and error recovery. We propose the use of runtime assertions for session-based programming in the Session C framework. Assertions are properties that are expected to hold during runtime, and they can complement static type checking. Error recovery is also a topic of interest, as large scale high performance parallel applications often need to gracefully handle unexpected errors such as hardware failures. Type-based approach to error handling and recovery will be explored as part of ongoing research on Scribble.

Adapting to other programming models. Our session-based approach is based on the message passing communication model, which can be used for coordination between heterogeneous nodes. Heterogeneous accelerators all use different programming and communication models, for example, General Purpose computing on GPU (GPGPU) uses a streaming model, and some reconfigurable hardware uses data-flow programming model. Adapting the high-level Scribble and Pabble language to these models will enable session types to be a common language to describe communication behaviour for parallel applications. We are aiming to achieve this by generalising our code generation to generate different target code.

Acknowledgements. The research leading to these results has received funding from EPSRC EP/F003757/01, EP/G015635/01 and the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521. The support by the HiPEAC NoE, the Maxeler University Program, and Xilinx is gratefully acknowledged.

References

1. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM 52(10), 56–67 (Oct 2009)
2. Deniélou, P.M., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. LMCS 8(4), 1–47 (2012)
3. 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)
4. Honda, K., Marques, E., Martins, F., Ng, N., Vasconcelos, V., Yoshida, N.: Verification of MPI programs using session types. In: Proc. EuroMPI 2012. LNCS, vol. 7940, pp. 291–293. Springer (2012)
5. Honda, K., Mukhamedov, A., Brown, G., Chen, T.C., Yoshida, N.: Scribbling interactions with a formal foundation. In: ICDCIT. LNCS, vol. 6536, pp. 55–75. Springer (2011)
6. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer-Verlag (1998)
7. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL'08. vol. 5201, p. 273 (2008)
8. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: ECOOP. LNCS, vol. 5142, pp. 516–541 (2008)
9. Marques, E., Martins, F., Vasconcelos, V., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: Proc. PLACES 2013 (2013), to appear
10. Mostrous, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: ESOP. LNCS, vol. 5502, pp. 316–332 (2009)
11. Ng, N., Yoshida, N.: Pabble: Parameterised scribble for parallel programming. In: PDP 2014 (2014), to appear
12. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe parallel programming with message optimisation. In: TOOLS. pp. 203–219 (2012)
13. Ng, N., Yoshida, N., Niu, X.Y., Tsoi, K.H., Luk, W.: Session types: towards safe and fast reconfigurable programming. SIGARCH Comput. Archit. News 40(5), 22–27 (Mar 2012), http://doi.acm.org/10.1145/2460216.2460221
14. Ng, N., Yoshida, N., Pernet, O., Hu, R., Kryftis, Y.: Safe Parallel Programming with Session Java. In: COORDINATION. LNCS, vol. 6721, pp. 110–126 (2011)
15. 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)
16. Scribble homepage, http://www.jboss.org/scribble
17. Siegel, S.F., Zirkel, T.K.: Automatic formal verification of MPI-based parallel programs. In: PPoPP'11. p. 309. ACM Press (2011)
18. Tsoi, K.H., Luk, W.: Axel: A Heterogeneous Cluster with FPGAs and GPUs. In: FPGA'10. pp. 115–124. ACM (2010)
19. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B.R., Schulz, M., Bronevetsky, G.: A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In: SC'10. pp. 1–10. IEEE (2010)
20. 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 (2009)
21. Session C homepage, http://www.doc.ic.ac.uk/~cn06/sessionc/