Session Types

Towards safe and fast reconfigurable programming

HEART 2012

Nicholas Ng, Nobuko Yoshida,
Xin Yu Niu, Kuen Hung Tsoi, Wayne Luk
Department of Computing, Imperial College London

Imperial College London

May 31, 2012

Motivation

- Parallel and heterogeneous architectures
  - Combines parallelism and specialisation, eg FPGA
  - Efficient use of computing resources
  - Difficult to program (correctly)
- One source of error of parallelising
  - Communication mismatch (send-receive)
  - Communication deadlocks

Message passing communication

- Message passing communication
  - scalable, commonly used
- MPI (Message-Passing Interface)
  - common for communication in parallel computers
- Communication mismatch and deadlocks
  - lead to program error

Motivating example

```
if (rank == 0) { // Program 0
    MPI_Send(a, 5, MPI_INT, 1 , TAG, MPI_COMM_WORLD);
    MPI_Recv(b, 5, MPI_INT, 1 , TAG, MPI_COMM_WORLD);
} else if (rank == 1) { // Program 1
    MPI_Send(b, 5, MPI_INT, 0 , TAG, MPI_COMM_WORLD);
    MPI_Recv(a, 5, MPI_INT, 0 , TAG, MPI_COMM_WORLD);
}
```

Program 0	Program 1
Send a	Send b
Recv b	Recv a

Figure: Interaction of two processes with a deadlock.

Contributions

- An intuitive programming framework and toolchain
  - For message-passing parallel programming
  - Based on formal and explicit interaction protocol
- Advanced communication topologies for computer clusters
- Case study comparing framework with existing tools

Proposal: Session types

- **Session Types** [Honda et al. ESOP'98, POPL'08]
  - Typing system for communication
  - Ensure compatible communication (send-receive) by typing
    - Sequence of send and receive
    - Also types flow-control constructs (eg. loops, if-then-else)
- Ideal to integrate into programming language

Program 0	Program 1
`Send a`	`Send b`
`Recv b`	`Recv a`

Figure: Incompatible.

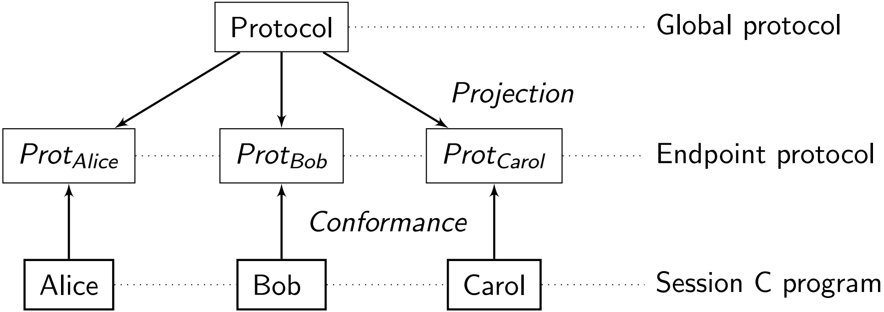
Program 0		Program 1
`Send a`	$\longrightarrow$	`Recv a`
`Recv b`	$\longleftarrow$	`Send b`

Figure: Compatible.

Session C programming framework: Aims

- Minimal extension of C language to support Session Types
- General communication-based framework
  - Safe distributed parallel programming
  - High performance applications
- Focussed on computer cluster communication

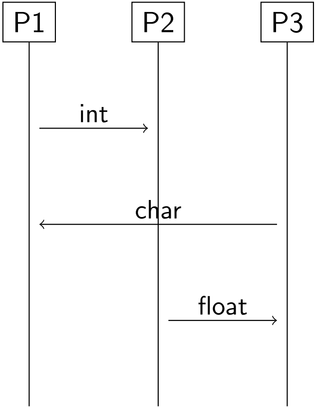
Session C toolchain and key reasoning



1. Design protocol in global view
2. Automatic *projection* to endpoint protocol, algorithm preserves safety
3. Write program according to endpoint protocol
4. Check program conforms to protocol
5. $\Rightarrow$ Safe program by design

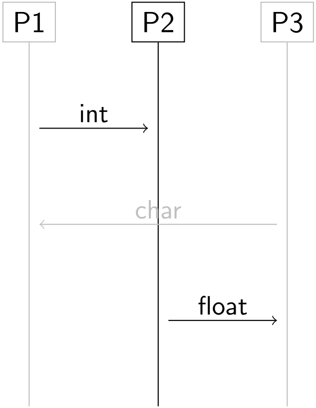
Scribble protocol specification language: Example

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



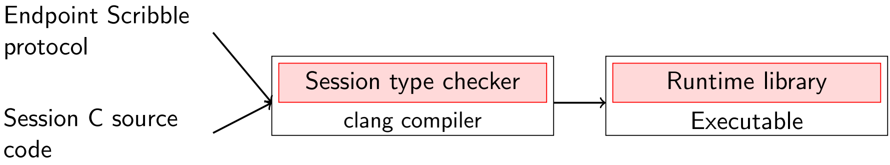
Scribble protocol specification language: Example

```
/* Endpoint protocol for P2 */
protocol Simple at P2
  (role P1, role P3) {
  int    from P1;

  float  to P3;
}
```



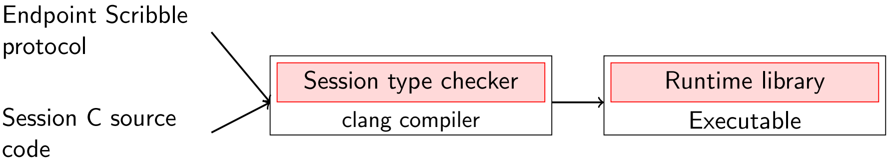
- Projection of Simple with respect to P2
- Endpoint protocol from perspective of P2

Session C Architecture



- User input *protocol* and *C source code*
- Session C framework
  - Runtime/communication API
  - Session Type checker

Session C Architecture: Session Type checker

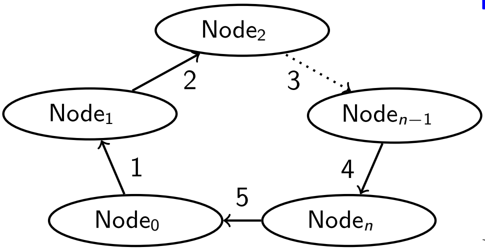


- Static analyser for source code
- Verify source code conforms with protocol specification
- Protocol extracted based on usage of API

Example topologies in the framework

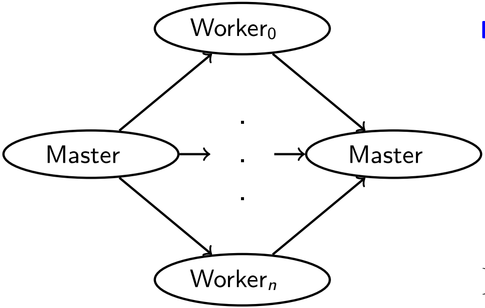
- Topology safe if can be described in framework
  - Subject to global protocol well-formedness conditions
- Examples: Ring topology and map-reduce

Topologies: Ring




```
protocol Ring {
  rec LOOP {
    datatype from Node0 to Node1;
    datatype from Node1 to NodeN;
    // Wrap back to Node0
    datatype from NodeN to Node0;
    LOOP;
  }
}
```

Topologies: Map-reduce

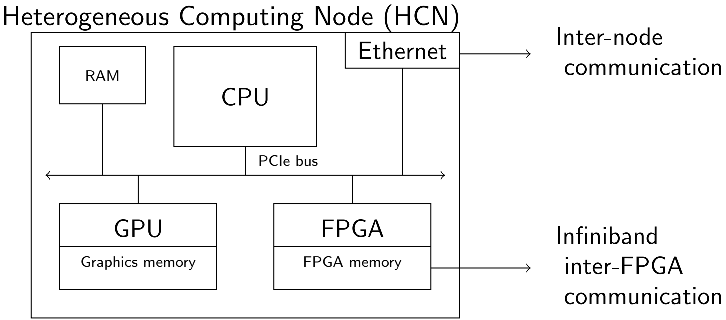



```
protocol MR {
  rec LOOP {
    datatype from Master to Worker0,
                                    Worker1;
    datatype from Worker0,
                        Worker1 to Master;
    LOOP;
  }
}
```

Evaluation

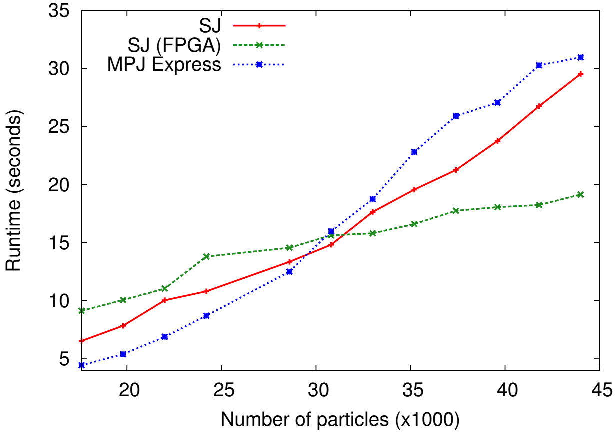
- Comparing session-enhanced programming with MPI
- Strength: Protocol known (and safe) at implementation time
- Asynchronous operation re-ordering
  - Optimisation applied to implementations
  - Correctness ensured by Session Type checker

Evaluation: environment



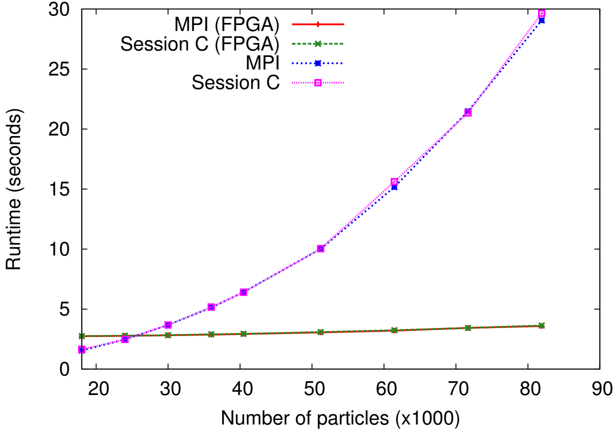
- Heterogeneous accelerators
  - Multicore CPU
  - GPU
  - FPGA
- Communication
  - Inter-node: Ethernet
  - Inter-component: PCI
  - Inter-FPGA: Infiniband

N-body simulation accelerated by FPGA (Java)



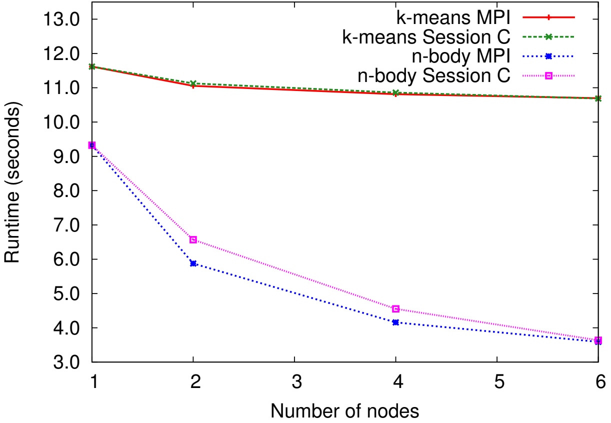
- Session Java comparable to MPJ Express (MPI in Java)
- FPGA overhead improves with larger input size

N-body simulation accelerated by FPGA (C)



- Session C performance same as MPI
- Significant improvement with FPGA acceleration

Scalability: N-body simulation and K-means clustering



- Performance improve with number of nodes
- MPI and Session C converge as nodes increase

Evaluation: Summary

- Session-enhanced languages (Java and C)
    - Communication safety ensured
    - Negligible performance cost
    - FPGA acceleration improves performance
    - Solution scalable

Future work

- Extending approach to include eg GPU or other hardware
- MPI-compatible runtime for multiparty session programming
- Integrate with customisable communication framework [Denholm et al., ASAP'11]

Conclusion

- Introduced a programming and verification framework for communication in C
- Shown advanced communication topologies for computer clusters
- Performance evaluation of framework against existing tools
    - Competitive performance

Try it!

Session C runtime and type-checker
`http://sesscc.googlecode.com`

Appendix

Related works

- MPI Deadlock detection by model checking techniques
  - ISP/DAMPI [Vo et al., PPoPP'09/SC'10]
  - TASS [Siegel et al., PPoPP'11]
- Our approach does not depend on testing or heuristics
  - Full guarantee of deadlock-freedom and communication-safety

Session C example: Protocol design

Description of protocol

```
protocol P (role A, role B) {
  int from A to B;
  int from B to A;
}
```

⇒ Description of a protocol for each endpoints

```
protocol P at A (role B) {
  int to B;
  int from B;
}

protocol P at B(role A) {
  int from A;
  int to A;
}
```

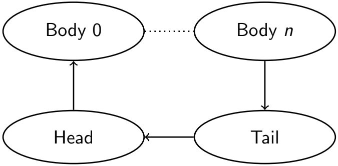
Session C example: Implementation

```
#include <libsess.h>
int main() { // A session C program
   session *s; int ival, sum = 0;
  // Start a session that follows the protocol "Protocol_Endpoint"
   join_session (&argc, &argv, &s, "Protocol_Endpoint.spr");
   role Bob = s->get_role(s, "Bob"); // Get role handle

   send_int (Bob, 42); // Send int to Bob
  while (i<3) {
     recv_int (Bob, &ival); // Recv int from Bob
    sum += ival;
  }
   send_int (Bob, sum); // Send int to Bob

   end_session (s); // End a session
  return 0;
}
```

Ring topology: full example

N-body simulation: Ring topology (1)

- Input segmented to n parts
- Results shifted right until all nodes worked on all segments



N-node ring topology

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

N-body simulation: Ring topology (2)

```
/* Endpoint Protocol */
protocol Nbody at Body
    (role Head, role Tail) {
  rec NrOfIters {
    rec SubCompute {
      particles from Head;
      particles to Tail;

      SubCompute;}

    NrOfIters;}
}
```

```
/* Implementation of Body worker */
particle_t *ps, *tmp_ps;
while ( iterations ++ < ITERS_NR) {
  while (rounds++ < NODES_NR) {
    send_particles(Tail, tmp_ps);
    // Update veclocities
    compute_forces(ps, tmp_parts ,...);
    recv_particles(Head, &tmp_ps);
  } // Update positions
    // by received velocities
  compute_positions(ps, pvs, ... );
}
```

Asynchronous reordering

Safe pipeline communication

- Some synchronous operations can be **safely** reordered [Mostrous et al., ESOP'09]
- Pipelines impossible in 'strict' multiparty session types, possible with asynchronous subtyping
- Safe pipeline improves performance, more scalable

Stage I	Stage II	Stage III
`A:send`	`B:recv`	
		`C:recv`

Figure: MPST.

Stage I	Stage II	Stage III
`A:send`	`B:send`	`C:send`

Figure: Asynchronous subtyping.

Safe pipeline communication

- Some synchronous operations can be **safely** reordered [Mostrous et al., ESOP'09]
- Pipelines impossible in 'strict' multiparty session types, possible with asynchronous subtyping
- Safe pipeline improves performance, more scalable

Stage I	Stage II	Stage III
`A:send` →	`B:recv`	
	`B:send`	`C:recv`
`A:recv`		

Figure: MPST.

Stage I	Stage II	Stage III
`A:send`	`B:send`	`C:send`

Figure: Asynchronous subtyping.

Safe pipeline communication

- Some synchronous operations can be **safely** reordered [Mostrous et al., ESOP'09]
- Pipelines impossible in 'strict' multiparty session types, possible with asynchronous subtyping
- Safe pipeline improves performance, more scalable

Stage I	Stage II	Stage III
A:send →	B:recv	
	B:send →	C:recv
A:recv		C:send

Figure: MPST.

Stage I	Stage II	Stage III
A:send	B:send	C:send

Figure: Asynchronous subtyping.

Safe pipeline communication

- Some synchronous operations can be **safely** reordered [Mostrous et al., ESOP'09]
- Pipelines impossible in 'strict' multiparty session types, possible with asynchronous subtyping
- Safe pipeline improves performance, more scalable

Stage I	Stage II	Stage III
`A:send` →	`B:recv`	
	`B:send` →	`C:recv`
⇢ `A:recv`		`C:send` ⇢

Figure: MPST.

Stage I	Stage II	Stage III
`A:send`	`B:send`	`C:send`

Figure: Asynchronous subtyping.

Safe pipeline communication

- Some synchronous operations can be **safely** reordered [Mostrous et al., ESOP'09]
- Pipelines impossible in 'strict' multiparty session types, possible with asynchronous subtyping
- Safe pipeline improves performance, more scalable

Stage I	Stage II	Stage III
`A:send`	`B:recv`	
	`B:send`	`C:recv`
`A:recv`		`C:send`

Figure: MPST.

Stage I	Stage II	Stage III
`A:send`	`B:send`	`C:send`
`A:recv`	`B:recv`	`C:recv`

Figure: Asynchronous subtyping.

Back to Evaluation