Imperial College London

Queen Mary University of London

Multiparty Session C

Safe Parallel Programming with Message Optimisation

TOOLS 2012

Nicholas Ng, Nobuko Yoshida, Kohei Honda*

30 May, 2012

Motivation

- Parallel architectures
  - Efficient use of hardware resources
  - eg. Multicore processors, computer clusters
  - Difficult to program (correctly)
- One source of error
  - Communication mismatch (send-receive)
  - Communication deadlocks

Motivating example: Deadlock

```
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);
}
```

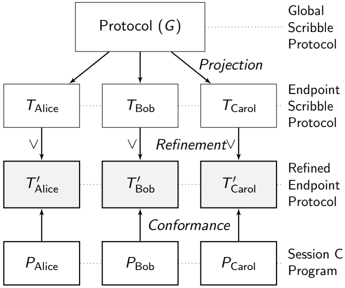
Program 0	Program 1
`Send a`	`Send b`
`Recv b`	`Recv a`

Contribution

- An intuitive programming framework and toolchain
  - For message-passing parallel programming
  - Based on formal and explicit interaction protocol
- First multiparty session-based programming environment for the low-level C language
  - Focussing on high performance, low latency
- Session type checker
  - Static checks
  - communication safety/deadlock freedom
  - Supports *asynchronous subtyping* for optimisation
- Evaluation with parallel algorithms implementation

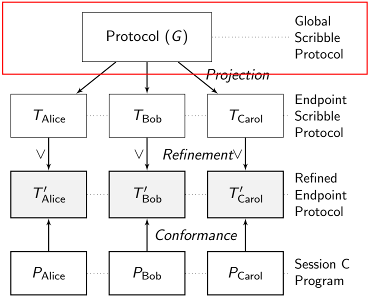
Session C programming: Overview

- Top down approach
- Based on multiparty session types (MPST) [Honda et al., POPL'08]
  - Communication should have a dual
  - Communication safety and deadlock freedom by typing



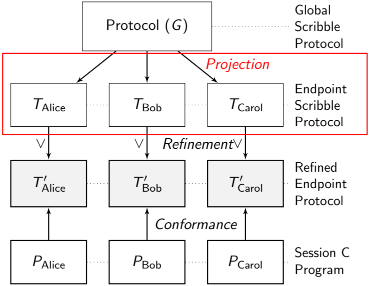
Session C programming: Key reasoning

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



Session C programming: 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

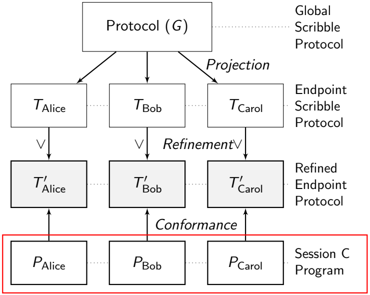


Imperial College London

Queen Mary University of London

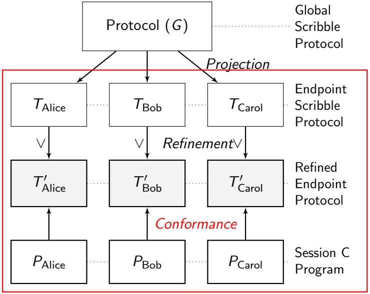
Session C programming: Key reasoning

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



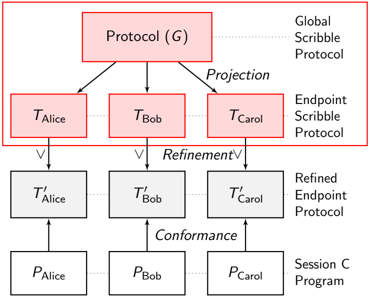
Session C programming: Key reasoning

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



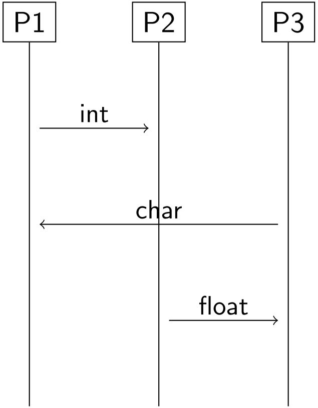
Scribble protocol specification language

- Developer friendly language to describe communication protocol (Red Hat, `www.scribble.org`)
- Interaction by message passing
- Captures flow-control elements of communication protocol



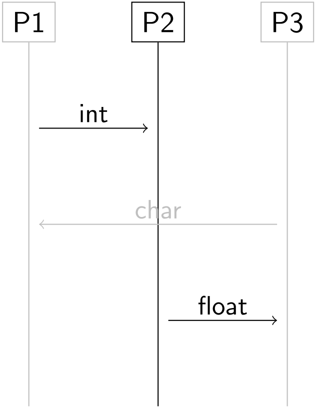
Scribble protocol specification language: Example

```
/* 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
}
```



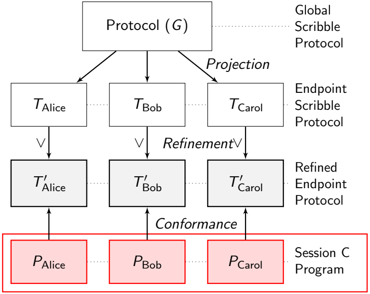
Scribble protocol specification language: Example

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

  float  to P3;
}
```



Session C runtime

- Message passing communication API
- Built on 0MQ socket library
- Aim: simple and lightweight



Session C runtime

- Basic primitives
  - Message passing
  - Iteration
  - Choice
- Advanced primitives
  - Multicast
  - Multi-channel iteration

- Primitives corresponds to protocol statements
- Most C features allowed with a few exceptions

Imperial College London

Queen Mary University of London

Session C runtime: Examples

Iteration and message passing

```
while (i<3) {
  send_int(A, 42);
}
```

```
while (i<3) {
  int val; recv_int(B, &val);
}
```

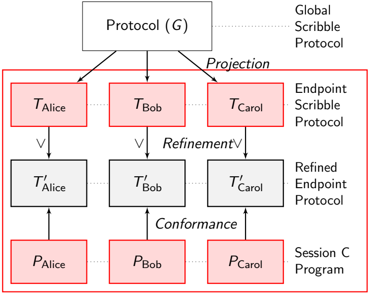
Directed choice

```
if (i<3) { // Choice from
  outbranch(B, LABEL0);
  send_int(B, 12);
} else {
  outbranch(B, LABEL1);
  send_char(B, 'A');
}
```

```
// Choice to
switch (inbranch(A, &label)) {
  case LABEL0:
    recv_int(A, &ival); break;
  case LABEL1:
    recv_char(A, &cval); break;
}
```

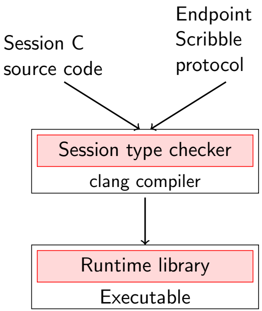
Session Type checking

- Static analyser
- Verify source code conforms with specification (endpoint protocol)



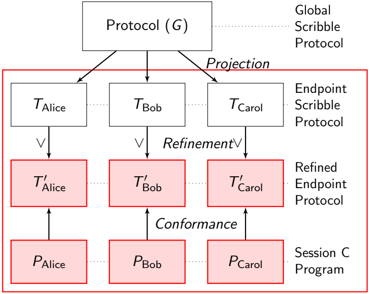
Session Type checking

- Implemented as a LLVM/clang compiler plugin
- Part of compilation process
- Session typing extracted based on usage of API



Session Type checking: Asynchronous optimisation

- Protocols designed safe, not necessarily efficient
- Asynchronous communication
  - non-blocking send
  - blocking receive
- Send/receive operation can overlap



Asynchronous optimisation

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

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

Figure: MPST.

Asynchronous optimisation

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

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

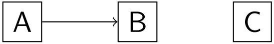


Figure: MPST.

Asynchronous optimisation

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

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

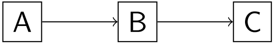


Figure: MPST.

Asynchronous optimisation

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

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

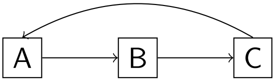


Figure: MPST.

Asynchronous optimisation

- Asynchronous operations can be **safely** permuted [Mostrous et al., ESOP'09]
- Pipelines inefficient in 'strict' multiparty session types
- Efficient pipelines with asynchronous subtyping of MPST

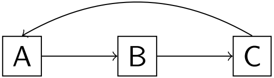
A B C

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

Figure: Asynchronous subtyping.

Asynchronous optimisation

- Asynchronous operations can be **safely** permuted [Mostrous et al., ESOP'09]
- Pipelines inefficient in 'strict' multiparty session types
- Efficient pipelines with asynchronous subtyping of MPST



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

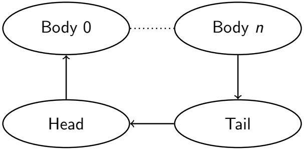
Figure: Asynchronous subtyping.

Parallel algorithms

- Demonstrating the expressiveness of MPST
- Performance evaluation: Static session types based approach do not degrade performance
- Example representative topologies
  1. N-body simulation: Ring topology
  2. Jacobi solution for the DPE: Mesh topology
  3. Linear equation solver: Warparound mesh topology
  4. Fast Fourier Transformation: Butterfly topology

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; }
}
```

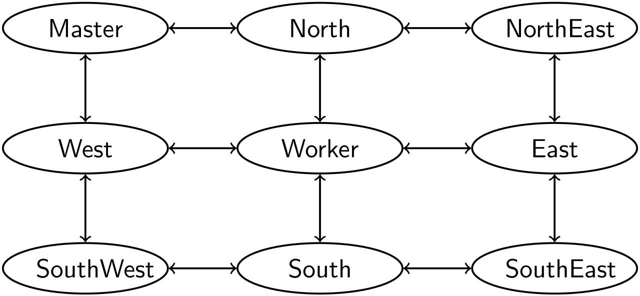
N-body simulation: Ring topology (2)

```
/* 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, ... );
}
```

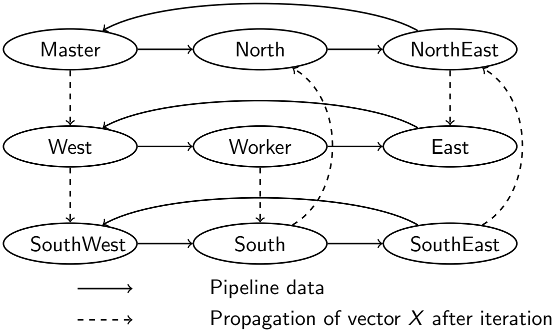
Jacobi solution for the DPE: Mesh topology

- Input segmented to 2D sub-grids
- Edge results exchanged between each neighbours
- Takes full advantage of asynchronous message optimisation



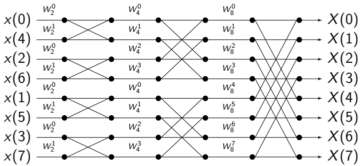
Linear equation solver: Wraparound mesh

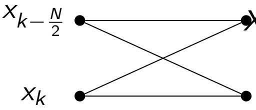
- Rows: Ring topology
- Columns: Diagonal propagates result to all in least distance



Fast Fourier Transformation: Butterfly topology

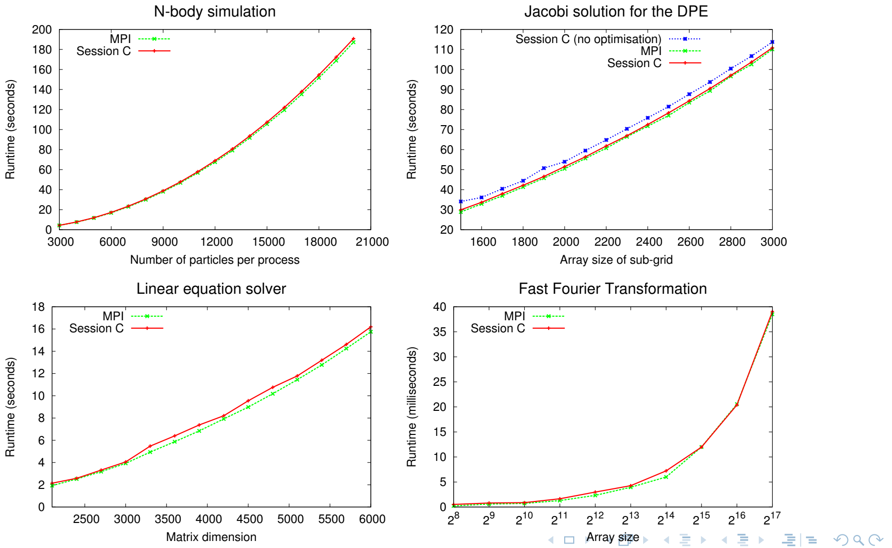
- Binary session types cannot efficiently represent
- Butterfly exchange: asynchronous optimisation





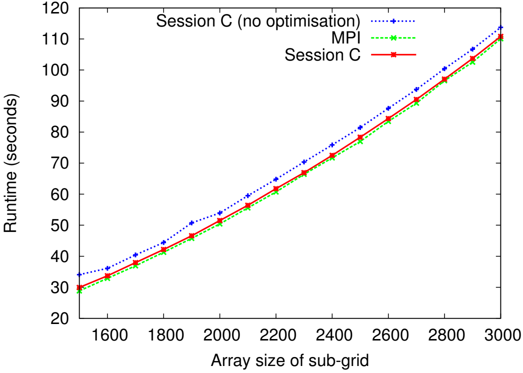
$$X_{k-\frac{N}{2}} = x_{k-\frac{N}{2}} + x_k * w_N^{k-\frac{N}{2}}$$

$$X_{k-\frac{N}{2}} = x_{k-\frac{N}{2}} + x_k * w_N^{k}$$


N-body simulation
Runtime (seconds)
Number of particles per process
MPI
Session C
Jacobi solution for the DPE
Runtime (seconds)
Array size of sub-grid
Session C (no optimisation)
MPI
Session C
Linear equation solver
Runtime (seconds)
Matrix dimension
MPI
Session C
Fast Fourier Transformation
Runtime (milliseconds)
Array size
MPI
Session C

Benchmark results: highlight

- Jacobi method for the Discrete Poisson Equation
- Mesh topology
- Asynchronous optimisation: 8% improvement



Related works

- MPI Deadlock detection by model checking
  - ISP/DAMPI [Vo et al., PPoPP'09/SC'10]
  - TASS [Siegel et al., PPoPP'11]
- Formally founded HPC languages
  - Pilot [Carter et al., IPDPSW'10] combines CSP and MPI
  - Occam-pi language : CSP and π-calculus
  - X10 [Lee et al., PPoPP'10] : PGAS
  - Session Java [Hu et al., ECOOP'08] applied in parallel programming setting [Ng et al., COORDINATION'11]

Conclusion

- Introduced a programming framework and toolchain for communication safe parallel programming in C
  - Based on formal and explicit interaction protocol
  - Low-level programming environment (C language)
  - Static type checker to verify implementation matches protocol
  - Type checker supports asynchronous subtyping for optimisation
  - Communicatoin safety and deadlock freedom by type checking
  - Static type checking does not degrade performance

Ongoing and future work

- Integrate with heterogeneous cluster with FPGA-acceleration [Ng et al., HEART'12]
- Parametrised processes for multiple replicated process [Denielou et al., FoSSaCS'10]
- Ongoing collaboration with Red Hat on Scribble project
- Memory/pointer safety by integrating Cyclone [Jim et al., USENIX ATC'02]

Try it!

Latest version on GitHub

`http://www.github.com/nickng/sessc`