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

```c
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**

- Send a
-Recv b

**Program 1**

- Send 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. ⇒ 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
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. ⇒ 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. ⇒ 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;
    }

Nicholas Ng, Nobuko Yoshida and Kohei Honda
Multiparty Session C: Safe Parallel Programming with Message Optimisation
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
Session C runtime: Examples

Iteration and message passing

```c
while (i < 3) {
    send_int(A, 42);
}
while (i < 3) {
    int val;
    recv_int(B, &val);
}
```

Directed choice

```c
if (i < 3) { // Choice from
    outbranch(B, LABEL0);
    send_int(B, 12);
} else { // Choice to
    switch (inbranch(A, &label)) {
        case LABEL0:
            recv_int(A, &ival);
            break;
        case LABEL1:
            recv_char(A, &cval);
            break;
    }
    outbranch(B, LABEL1);
    send_char(B, 'A');
}
```
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

<table>
<thead>
<tr>
<th>Stage I</th>
<th>Stage II</th>
<th>Stage III</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:send</td>
<td>B:recv</td>
<td>C:recv</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Stage I</th>
<th>Stage II</th>
<th>Stage III</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:send</td>
<td>B:recv</td>
<td></td>
</tr>
<tr>
<td>B:send</td>
<td>C:recv</td>
<td></td>
</tr>
<tr>
<td>A:recv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Stage I</th>
<th>Stage II</th>
<th>Stage III</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:send</td>
<td>B:recv</td>
<td></td>
</tr>
<tr>
<td>B:send</td>
<td>C:recv</td>
<td></td>
</tr>
<tr>
<td>A:recv</td>
<td></td>
<td>C:send</td>
</tr>
</tbody>
</table>

![Figure: MPST.](image)

Nicholas Ng, Nobuko Yoshida and Kohei Honda
Multiparty Session C: Safe Parallel Programming with Message Optimisation
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

<table>
<thead>
<tr>
<th>Stage I</th>
<th>Stage II</th>
<th>Stage III</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:send</td>
<td>B:send</td>
<td>C:send</td>
</tr>
</tbody>
</table>

**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

---

**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
\textit{N}-body simulation: Ring topology (1)

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

\begin{verbatim}
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;
    }
  }
}
\end{verbatim}
/* 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 velocities
        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

Pipeline data

Propagation of vector $X$ after iteration
Fast Fourier Transformation: Butterfly topology

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

\[
X_{k - \frac{N}{2}} = X_{k - \frac{N}{2}} + x_k \ast w_N^{k - \frac{N}{2}}
\]

\[
X_k \ast w_N^{k - \frac{N}{2}} = X_{k - \frac{N}{2}} + x_k \ast w_N^{k - \frac{N}{2}}
\]
Nicholas Ng, Nobuko Yoshida and Kohei Honda

Multiparty Session C: Safe Parallel Programming with Message Optimisation
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 $\pi$-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
  - Communication 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