Motivation

- Parallel programming is non-trivial and error prone (e.g., deadlock)
- Session types theory guarantees communication safety between processes
- Parallel programming with a session-based programming language for safety (type safety, deadlock freedom) and performance
Contributions

1. Extended Session Java (SJ) with *multi-channel primitives* for parallel programming
2. Defined *multi-channel session calculus* with operational semantics and typing system
3. Showed the practical use of *multi-channel primitives* by implementing representative parallel algorithms in SJ
4. Evaluated performance of parallel algorithms implemented in SJ and compared against MPJ Express
Session types

- Typing system for $\pi$-calculus [Honda et al., ESOP'98]
- $\pi$-calculus models structured interactions between processes
- Communication should have a dual

Conventional types/sorts

- `int i = 9`
- `i` and `9` are both `int` datatype

Session types

- Program 1: `send(9)`
- Program 2: `int intValue = receive()`
- *Send int* and *Receive int* are duals
Session programming with SJ

Session Java (SJ) [Hu et al., ECOOP’08]
- An implementation of session types in Java
- Provides a socket programming interface
  - Session initiation: `accept()` `request()`
  - Communication: `send()` `receive()`
  - Iteration: `outwhile` statement `inwhile` statement
- Preliminary work in [Bejleri et al., PLACES’09]
- But lacks efficient mechanism to synchronise multiple sessions
Session programming with SJ: Workflow

1. Declare session type (called protocol) in source code
2. Local session type conformance by SJ compiler (ie. does program implement session as declared?)
3. Duality check between communicating programs at runtime (ie. are protocols compatible?)

```
protocol helloWorldSvr {
    sbegin. // start session
    ![ // Outwhile
        ![String> // Send ]*
}]
```

```
protocol helloWorldClnt {
    cbegin. // Join session
    ?[ // Inwhile
        ?(String) // Recv ]*
}]
```
Session programming with SJ: Workflow

1. Declare session type (called **protocol**) in source code
2. Local session type conformance by SJ compiler (ie. does program implement session as declared?)
3. Duality check between communicating programs at runtime (ie. are **protocols** compatible?)

```java
protocol helloWorldSvr
 { sbegin.![<!<String>>]* } }

SJSocket s = ss.accept();
s.outwhile(i++<3) {
    s.send("Hello World");
}

protocol helloWorldClnt
 { cbegin.?[?(String)]* } }

SJSocket c = cs.request();
c.inwhile {
    String str = c.receive();
}
```
inwhile and outwhile

- Powerful construct to connect two sessions
- Allow one process to control iteration of another

\[
P_1 \xrightarrow{s_{12}} P_2
\]

\(s_{12}: \text{session between } P_1 \text{ and } P_2\)

\(P_1\) \(s_{12}.\text{outwhile}(\text{true})\)\{ /*... */ \}

\(P_2\) \(s_{12}.\text{inwhile} \)\{ /*... */\}
Iteration chaining

How to synchronise multiple independent sessions?

\[ P_1 \xrightarrow{s_{12}} P_2 \xrightarrow{s_{23}} P_3 \]
Iteration chaining

How to synchronise multiple independent sessions?

\[ P_1 \xrightarrow{s_{12}} P_2 \xrightarrow{s_{23}} P_3 \]

Incorrect, non type-safe implementation of \( P_2 \):

```java
s12.inwhile {
    s23.outwhile(true) {
        // ...
    }
}
```
Iteration chaining

How to synchronise multiple independent sessions?

\[ P_1 \xrightarrow{s_{12}} P_2 \xrightarrow{s_{23}} P_3 \]

\(P_2\) with *iteration chaining* syntax:

```java
s23.outwhile(s12.inwhile) {
    // ...
    s12.send();
    s23.send();
}
```
Multi-channel primitives in SJ

How to write $P_1$ (again, incorrect and non type-safe):

```java
e = true;
s12.outwhile( e ) {
    s13.outwhile( e ) {
        // ...
    }
}
```
Multi-channel primitives

Multi-channel outwhile:

```java
<s12, s13>.outwhile(true) {
    // ...
}
```
Similarly for $P_4$, multi-channel inwhile:

```java
<s24, s34>.inwhile {
    // ...
}
```
Multi-channel primitives example: Jacobi solution
Multi-channel primitives example: Jacobi solution

```java
/** Master */
<right, down>.outwhile(e)
{
  // ...
}

/** North */
<right, down>.outwhile(
  left.inwhile)
{
  // ...
}

/** NorthEast */
<right, down>.outwhile(
  left.inwhile)
{
  // ...
}

/** West */
<down, right>.outwhile(
  up.inwhile)
{
  // ...
}

/** Worker */
<right, down>.outwhile(
  <left, up>.inwhile)
{
  // ...
}

/** East */
<left, up>.outwhile(
  <left, up>.inwhile)
{
  // ...
}

/** SouthWest */
<right.outwhile(
  up.inwhile)
{
  // ...
}

/** South */
<right.outwhile(
  <left, up>.inwhile)
{
  // ...
}

/** SouthEast */
<left, up>.inwhile
{
  // ...
}
```
Multi-channel primitives example: Jacobi solution

Worker process, chained multi-channel inwhile and outwhile

```java
<right, down>.outwhile(<left, up>.inwhile) {
    // ... calculation ...
    up.send(topRow);
    topRow = up.receive();
    right.send(rightCol);
    rightCol = right.receive();

    bottomRow_rcvd = down.receive();
    down.send(bottomRow);
    leftCol_rcvd = left.receive();
    left.send(leftCol);
}
```
Multi-channel primitives in SJ: summary

- More topologies can be expressed
- More intuitive to program and reason about
- Synchronises multiple sessions
Multi-channel session types: intuition

- Formalisation of multi-channel primitives
  - Correctness
  - Deadlock freedom
- `outwhile` multicasts loop condition to all channels
- `inwhile` collects loop conditions from all channels
Multi-channel session types: reduction rules (1)

Outwhile (true)

\[ E[\langle k_1 \ldots k_n \rangle.\text{outwhile}(e)\{ P \}] \quad (E[e] \rightarrow {}^*E'[\text{true}]) \]
\[ \rightarrow E[P;\langle k_1 \ldots k_n \rangle.\text{outwhile}(e')\{ P \}] | k_1 \uparrow[\text{true}] | \ldots | k_n \uparrow[\text{true}] \]

Outwhile (false)

\[ E[\langle k_1 \ldots k_n \rangle.\text{outwhile}(e)\{ P \}] \quad (E[e] \rightarrow {}^*E'[\text{false}]) \]
\[ \rightarrow E[0] | k_1 \uparrow[\text{false}] | \ldots | k_n \uparrow[\text{false}] \]

- Multichannel outwhile forwards loop condition to all session channels
Multi-channel session types: reduction rules (2)

**Inwhile (true)**

\[ E[\langle k_1 \ldots k_n \rangle.\text{inwhile}\{ P \}]} \mid k_1 \triangleright [\text{true}] \mid \ldots \mid k_n \triangleright [\text{true}] \]
\[ \rightarrow E[P;\langle k_1 \ldots k_n \rangle.\text{inwhile}\{ P \}] \]

**Inwhile (false)**

\[ E[\langle k_1 \ldots k_n \rangle.\text{inwhile}\{ P \}]} \mid k_1 \triangleright [\text{false}] \mid \ldots \mid k_n \triangleright [\text{false}] \]
\[ \rightarrow E[0] \]

- Multichannel `inwhile` collects loop conditions from all session channels
- Proceeds if conditions match
- Mismatch of conditions: runtime error
Well-formed topology: Example

\[
P_1 \\
\downarrow \\
P_2 \quad \rightarrow \quad P_3
\]

- `s23.outwhile(<s12, s23>.inwhile)`
- Valid inwhile outwhile topology construction
Well-formed topology: Example

- `s23.outwhile(<s12, s23>.inwhile)`
- Valid `inwhile` `outwhile` topology construction
- Cycle in the flow of control messages: **deadlock**
Well-formed topology

- Governs how multi-channel `outwhile` and `inwhile` are connected.
- Well-formed iff topology constructed as **uni-directed acyclic graph**
- All examples in paper conforms to well-formed topology:
  - *n*-Body simulation: ring topology
  - Jacobi solution of the discrete Poission equation: mesh topology
  - Linear equation solver: wraparound mesh topology
Theorem (Subject reduction)

*Multi-channel* `outwhile` and `inwhile` will not reduce to error

Theorem (Type and communication safety)

A typable process which forms a well-formed topology is type and communication safe.

Theorem (Deadlock freedom)

*If* \( P \) *forms a well-formed topology and* \( P \) *is well-typed, then* \( P \) *is deadlock free.*
Multi-channel session types and SJ programming

Workflow of a SJ program:

1. Declare session type (called protocol) in source code
2. Local session type conformance by SJ compiler
3. Well-formed topology verification on deployment config file
4. Program instantiated with verified config file
5. Duality check between communicating programs at runtime
Benchmark results

- Significant improvement over non multi-channel version
- Performs competitively against MPJ Express (MPI in Java)
Conclusions

- Multi-channel primitives increased the expressiveness of Session Java

- *multi-channel session type theory* and *well-formed topology* guarantees *communication safety* and *deadlock freedom*

- Benchmark result shows competitive performance against industry standard

- Parallel programming in multi-channel SJ is both *safe* and *efficient*
Future work

- Session-based low level, natively compiled language (eg. C) for low overhead HPC and systems programming
- Incorporate outwhile and inwhile primitives into multiparty session types

Full version:
http://www.doc.ic.ac.uk/~cn06/pub/2011/sj_parallel/
<table>
<thead>
<tr>
<th>Introduction</th>
<th>Multi-channel session programming with SJ</th>
<th>Multi-channel session types</th>
<th>Benchmarks</th>
<th>Conclusions</th>
</tr>
</thead>
</table>

Nicholas Ng  
Nobuko Yoshida  
Olivier Pernet  
Raymond Hu  
Yiannos Kryftis*  

Safe Parallel Programming with Session Java  
Imperial College
Syntax

(V) \[ \begin{align*} v & ::= a, b, x, y \quad \text{shared names} \\ & | \text{true, false} \quad \text{boolean} \\ & | n \quad \text{integer} \end{align*} \]

(E) \[ \begin{align*} e & ::= v \mid e + e \mid \text{not}(e) \ldots \quad \text{value, sum, not} \\ & | \langle k_1 \ldots k_n \rangle . \text{inwhile} \quad \text{inwhile} \end{align*} \]

(P) \[ \begin{align*} P & ::= 0 \quad \text{inaction} \\ & | T \quad \text{prefixed} \\ & | P : Q \quad \text{sequence} \\ & | P \mid Q \quad \text{parallel} \\ & | (\nu u) P \quad \text{hiding} \end{align*} \]

(D) \[ D ::= X(xk) = P \]

(T) \[ T ::= \text{request } a(k) \text{ in } P \\ & | \text{accept } a(k) \text{ in } P \\ & | k![\tilde{e}] \\ & | k?!(\tilde{x}) \text{ in } P \\ & | \text{throw } k[k'] \\ & | \text{catch } k(k') \text{ in } P \\ & | X[\tilde{e} \tilde{k}] \\ & | \text{def } D \text{ in } P \\ & | k < l \\ & | k > \{ l_1 : P_1 \| \ldots \| l_n : P_n \} \\ & | \text{if } e \text{ then } P \text{ else } Q \\ & | \langle k_1 \ldots k_n \rangle . \text{inwhile}\{ Q \} \\ & | k \uparrow [b] \\ & | \langle k_1 \ldots k_n \rangle . \text{outwhile}(e)\{ P \} \]
Operational Semantics

\[ \text{accept } a(k) \text{ in } P_1 \mid \text{request } a(k) \text{ in } P_2 \rightarrow (\nu k)(P_1 \mid P_2) \]

\[ k \triangleright \{ l_1 : P_1 \mid \cdots \mid l_n : P_n \} \mid k \triangleleft l_i ; \rightarrow P_i \quad (1 \leq i \leq n) \]

\[ \text{if } \text{true} \text{ then } P \text{ else } Q \rightarrow P \]

\[ \text{def } X(xk) = P \text{ in } X[ck] \rightarrow \text{def } X(xk) = P \text{ in } P\{c/x\} \]

\[ \langle k_1 \ldots k_n \rangle.\text{inwhile} \{ P \} \mid \Pi_i \in \{1..n\} k_i \triangleright \text{true} \rightarrow P; \langle k_1 \ldots k_n \rangle.\text{inwhile} \{ P \} \]

\[ \langle k_1 \ldots k_n \rangle.\text{inwhile} \{ P \} \mid \Pi_i \in \{1..n\} k_i \triangleright \text{false} \rightarrow 0 \]

\[ E[\langle k_1 \ldots k_n \rangle.\text{inwhile}] \mid \Pi_i \in \{1..n\} k_i \triangleright \text{true} \rightarrow E[\text{true}] \]

\[ E[\langle k_1 \ldots k_n \rangle.\text{inwhile}] \mid \Pi_i \in \{1..n\} k_i \triangleright \text{false} \rightarrow E[\text{false}] \]

\[ E[e] \rightarrow^* E'[\text{true}] \Rightarrow \]

\[ E[\langle k_1 \ldots k_n \rangle.\text{outwhile}(e) \{ P \}] \rightarrow E'[P; \langle k_1 \ldots k_n \rangle.\text{outwhile}(e) \{ P \}] \mid \Pi_i \in \{1..n\} k_i \triangleright \text{true} \]

\[ E[e] \rightarrow^* E'[\text{false}] \Rightarrow \]

\[ E[\langle k_1 \ldots k_n \rangle.\text{outwhile}(e) \{ P \}] \rightarrow E'[0] \mid \Pi_i \in \{1..n\} k_i \triangleright \text{false} \]

\[ P \equiv P' \text{ and } P' \rightarrow Q' \text{ and } Q' \equiv Q \Rightarrow P \rightarrow Q \]

\[ e \rightarrow e' \Rightarrow E[e] \rightarrow E[e'] \]

\[ P \mid Q \rightarrow P' \mid Q' \Rightarrow E[P] \mid Q \rightarrow E[P'] \mid Q' \]

In [Ow1] and [Ow2], we assume \( E = E' \mid \Pi_i \in \{1..n\} k_i \triangleright [b_i] \).
Type system

**Outwhile**

\[ \Gamma; \Delta \vdash e \triangleright \text{bool} \quad \Gamma \vdash P \triangleright \Delta \cdot k_1: \tau_1.\text{end} \ldots k_n: \tau_n.\text{end} \]

\[ \Gamma \vdash \langle k_1 \ldots k_n \rangle.\text{outwhile}(e)\{ P \} \triangleright \Delta \cdot k_1: ![\tau_1]^*.\text{end} \ldots k_n: ![\tau_n]^*.\text{end} \]

**Inwhile**

\[ \Gamma; \Delta \vdash Q \triangleright \Delta \cdot k_1: \tau_1.\text{end} \ldots k_n: \tau_n.\text{end} \]

\[ \Gamma \vdash \langle k_1 \ldots k_n \rangle.\text{inwhile}\{ Q \} \triangleright \Delta \cdot k_1: ?[\tau_1]^*.\text{end} \ldots k_n: ?[\tau_n]^*.\text{end} \]