# Safe Parallel Programming with Session Java

Nicholas Ng*, Nobuko Yoshida*
Olivier Pernet*, Raymond Hu*, and Yiannos Kryftis†

*Imperial College London †National Technical University of Athens

**Abstract.** The session-typed programming language Session Java (SJ) has proved to be an effective tool for distributed programming, promoting structured programming for communications and compile-time safety. This paper investigates the use of SJ for session-typed parallel programming, and introduces new language primitives for *chained iteration* and *multi-channel communication*. These primitives allow the efficient coordination of parallel computation across multiple processes, thus enabling SJ to express the complex communication topologies often used by parallel algorithms. We demonstrate that the new primitives yield clearer and safer code for pipeline, ring and mesh topologies through implementations of representative parallel algorithms. We then present a semantics and session typing system including the new primitives, and prove type soundness and deadlock-freedom for our implementations. The benchmark results show that the new SJ is substantially faster than the original SJ and performs competitively against MPJ Express[1] used as reference.

## 1 Introduction

The current practice of parallel and distributed programming is fraught with errors that go undetected until runtime, manifest themselves as deadlocks or communication errors, and often find their root in mismatched communication protocols. The Session Java programming language (SJ) [12] improves this status quo. SJ is an extension of Java with *session types*, supporting statically safe distributed programming by message-passing. Session types were introduced as a type system for the $\pi$-calculus [8, 21], and have been shown to integrate cleanly with formal models of object-oriented programming. The SJ compiler offers two strong static guarantees for session execution: (1) *communication safety*, meaning a session-typed process can never cause or encounter a communication error by sending or receiving unexpected messages; and (2) *deadlock-freedom* — a session-typed process will never block indefinitely on a message receive.

Parallel programs often make use of complex, high-level communication patterns such as globally synchronised iteration over chained topologies like rings and meshes. Yet modern implementations are still written using low-level languages and libraries, commonly C and MPI [13]: implementations make the best use of hardware, but at the cost of complicated programming where communication is entangled with computation. There is no global view of inter-process communication, and no formal guarantees are given about communication correctness, which often leads to hard-to-find errors.

[1] MPJ Express [16] is a Java implementation of the MPI standard. Extensive benchmarks comparing MPJ Express to other MPI implementations are presented in [16]. The benchmarks show performance competitive with C-based MPICH2.

We investigate parallel programming in SJ as a solution to these issues. However, SJ as presented in [12] only guarantees progress for each session in isolation: deadlocks can still arise from the interleaving of multiple sessions in a process. Moreover, implementing chained communication topologies without additional language support requires temporary sessions, opened and closed on every iteration — a source of non-trivial inefficiencies (see § 3 for an example). We need new constructs, well-integrated with existing binary sessions, to enable lightweight *global* communication safety and deadlock-freedom, increase expressiveness to support *structured programming for communication topologies* and improve *performance*.

Our new *multi-channel* session primitives fit these requirements, and make it possible to safely and efficiently express parallel algorithms in SJ. The combination of new primitives and a well-formed topology check extension to SJ compilation [12] bring the benefits of type-safe, structured communications programming to HPC. The primitives can be chained, yielding a simple mechanism for structuring global control flow. We formalise these primitives as novel extensions of the *session calculus*, and the correctness condition on the shape of programs enforced by a simple extension of SJ compilation. This allows us to prove *communication safety* and *deadlock-freedom*, and offers a new, lightweight alternative to multiparty session types for global type-safety.

**Contributions.** This paper constitutes the first introduction to parallel programming in SJ, in addition to presenting the following technical contributions:

(§ **2**) We introduce SJ as a programming language for type-safe, efficient parallel programming, including our implementation of *multi-channel* session primitives, and the extended SJ tool chain for parallel programming. We show that the new primitives enable clearer, more readable code.

(§ **3**) We discuss SJ implementations of parallel algorithms using the Jacobi solution to the discrete Poisson equation (§ 3) as an example. The algorithm uses communication topology representative of a large class of parallel algorithms, and demonstrates the practical use of our multi-channel primitives.

(§ **4**) We define the *multi-channel session calculus*, its operational semantics, and typing system. We prove that processes conforming to a *well-formed communication topology* (Definition 4.1) satisfy the subject reduction theorem (Theorem 4.1), which implies *type and communication-safety* (Theorem 4.2) and *deadlock-freedom* across multiple, interleaved sessions (Theorem 4.3).

(§ **5**) Performance evaluation of *n*-Body simulation and Jacobi solution algorithms, demonstrating the benefits of the new primitives. The SJ implementations using the new primitives show competitive performance against an MPJ Express [14].

Related and future work are discussed in § **6**. Detailed definitions, proofs, benchmark results and source code can be found at the on-line Appendix [5].

**Acknowledgements.** We thank the referees for their useful comments and Brittle Tsoi and Wayne Luk for their collaborations. This work is partially supported by EPSRC EP/F003757 and EP/G015635.

## 2 Session-Typed Programming in SJ

This section firstly reviews the key concepts of session-typed programming using Session Java (SJ) [11, 12]. In (1), we outline the basic methodology; in (2), the protocol

structures supported by SJ. We then introduce the new session programming features developed in this paper to provide greater expressiveness and performance gains for *session-typed parallel programming*. In (3), we explain session *iteration chaining*; and in (4), the generalisation of this concept to the *multi-channel* primitives. Finally, (5) describes the *topology verification* for parallel programs.

**(1) Basic SJ programming.** SJ is an extension of Java for type-safe concurrent and distributed session programming. Session programming in SJ, as detailed in [12], starts with the declaration of the intended communication protocols as session types; we shall often use the terms *session type* and *protocol* interchangeably. A session is the interaction between two communicating parties, and its session type is written from the viewpoint of one side of the session. The following declares a protocol named P:

```
protocol P !<int>.?(Data)
```

Protocol P specifies that, at this side of the session, we first send (!) a message of Java type int, then receive (?) another message, an instance of the Java class Data, which finishes the session. After defining the protocol, the programmer implements the processes that will perform the specified communication actions using the SJ *session primitives*. The first line in the following code implements an Alice process conforming to the P protocol:

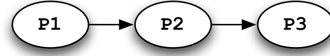
```
A: alice.send(42); Data d = (Data) alice.receive();//!<int>.?(Data)
B: int i = bob.receiveInt(); bob.send(new Data()); //?(int).!<Data>
```

The alice variable refers to an object of class SJSocket, called a *session socket*, which represents one endpoint of an active session. The session-typed primitives for session-typed communication behaviour, such as send and receive, are performed on the session socket like method invocations. SJSocket declarations associate a protocol to the socket variable, and the SJ compiler statically checks that the socket is indeed used according to the protocol, ensuring the *correct communication behaviour* of the process.

This simple session application also requires a counterpart Bob process to interact with Alice. For safe session execution, the Alice and Bob processes need to perform matching communication operations: when Alice sends an int, Bob receives an int, and so on. Two processes performing matching operations have session types that are *dual* to each other. The dual protocol to P is protocol PDual ?(int).!<Data>, and a dual Bob process can be implemented as in the second line of the above listing.

**(2) More complex protocol structures.** Session types are not limited to sequences of basic message passing. Programmers can specify more complex protocols featuring *branching*, *iteration* and *recursion*.

The protocols and processes in Fig.1 demonstrate session iteration and branching. Process *P1* communicates with *P2* according to protocol IntAndBoolStream; *P2* and *P3* communicate following protocol IntStream. Like basic message passing, iteration and branching are coordinated by *active* and *passive* actions at each side of the session. Process *P1* actively decides whether to continue the session iteration using outwhile (*condition*), and if so, selects a branch using outbranch(*label*). The former action implements the ![$\tau$]* type given by IntAndBoolStream, where $\tau$ is the !{Label1: $\tau_1$, Label2: $\tau_2$, ...} type implemented by the latter. Processes *P2* and *P3* passively follow




```
protocol IntAndBoolStream ![!{Label1: !<int>, Label2: !<boolean>}]*
protocol IntAndBoolDual   ?[?{Label1: ?<int>, Label2: ?(boolean)}]*
protocol IntStream        ![!<int>]*
protocol IntStreamDual    ?[?(int)]*
```

```
P1: s.outwhile(x < 10) {
      s.outbranch(Label1) {
        s.send(42);
    }}

P3: s.inwhile {
      int i = s.receiveInt();
    }
```

```
P2: s2.outwhile(s1.inwhile()) {
      s1.inbranch() {
        case Label1:
          int i = s1.receiveInt();
          s2.send(i);
        case Label2:
          boolean b = s1.receiveBool();
          s2.send(42);
    }}
```

Session socket *s* in *P1* follows *IntAndBoolStream*; *s1* and *s2* in *P2* follows *IntAndBoolDual* and *IntStream*; s in *P3* follows *IntStreamDual*.

**Fig. 1.** Simple chaining of session iterations across multiple pipeline process.

the selected branch and the iteration decisions (received as internal control messages) using `inbranch` and `inwhile`, and proceed accordingly; the two dual protocols show the passive versions of the above iteration and branching types, denoted by `?` in place of `!`.

So far, we have reviewed basic SJ programming features [12] derived from standard session type theory [8, 21]; the following paragraphs discuss new features motivated by the application of session types to parallel programming in practice.

**(3) Expressiveness gains from iteration chaining.** The three processes in Fig. 1 additionally illustrate session *iteration chaining*, forming a linear pipeline as depicted at the top of Fig. 1. The net effect is that *P1* controls the iteration of both its session with *P2* and transitively the session between *P2* and *P3*. This is achieved through the chaining construct `s2.outwhile(s1.inwhile())` at *P2*, which receives the iteration decision from *P1* and forwards it to *P3*. The flow of both sessions is thus controlled by the same master decision from *P1*.

Iteration chaining offers greater expressiveness than the individual iteration primitives supported in standard session types. Normally, session typing for ordinary `inwhile` or `outwhile` loops must forbid operations on any session other than the session channel that of loop, to preserve linear usage of session channels. This means that e.g. `s1.inwhile(){ s1.send(v); }` is allowed, whereas `s1.inwhile(){ s2.send(v); }` is not. With the iteration chaining construct, we can now construct a process containing two interleaved `inwhile` or `outwhile` loops on separate sessions. In fact, session iteration chaining can be further generalised as we explain below.

**(4) Multi-channel iteration primitives.** Simple iteration chaining allows SJ programmers to combine multiple sessions into linear pipeline structures, a common pattern in parallel processing. In particular, type-safe session iteration (and branching) along a pipeline is a powerful benefit over traditional stream-based data flow [18]. More complex topologies, however, such as rings and meshes, require iteration signals to be di-

```
Master:       <s1,s2>.outwhile(i < 42) {...}
Forwarder1:   s3.outwhile(s1.inwhile()) {...}
Forwarder2:   s4.outwhile(s2.inwhile()) {...}
End:          <s3,s4>.inwhile() {...}
```

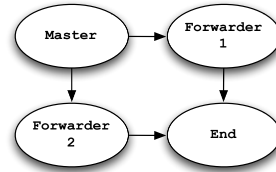



**Fig. 2.** Multi-channel iteration in a simple grid topology.

rectly forwarded from a given process to more than one other, and for multiple signals to be directed into a common sink; in SJ, this means we require the ability to send and receive multiple iteration signals over a set of session sockets. For this purpose, SJ introduces the generalised *multi-channel* primitives; the following focuses on multi-channel iteration, which extends the chaining constructs from above.

Fig. 2 demonstrates multi-channel iteration for a simple grid topology. Process *Master* controls the iteration on both the `s1` and `s2` session sockets under a single iteration condition. Processes *Forwarder1* and *Forwarder2* iterate following the signal from *Master* and forward the signal to *End*; thus, all four processes iterate in lockstep. Multi-channel `inwhile`, as performed by *End*, is intended for situations where multiple sessions are combined for iteration, but all are coordinated by an iteration signal from a common source; this means all the signals received from each socket of the `inwhile` will always agree — either to continue iterating, or to stop. In case this is not respected at run-time, the `inwhile` will throw an exception, resulting in session termination. Together, multi-channel primitives enable the type-safe implementation of parallel programming patterns like scatter-gather, producer-consumer, and more complex chained topologies. The basic session primitives express only disjoint behaviour within individual sessions, whereas the multi-channel primitives implement interaction across multiple sessions as a single, integrated structure.

**(5) The SJ tool chain with topology verification.** In previous work, the safety guarantees offered by the SJ compiler were limited to the scope of each independent *binary* (two-party) session. This means that, while any one session was guaranteed to be internally deadlock-free, this property may not hold in the presence of interleaved sessions in a process as a whole. The nodes in a parallel program typically make use of many interleaved sessions – with each of their neighbours in the chosen network topology. Furthermore, `inwhile` and `outwhile` in iteration chains must be correctly composed.

As a solution to this issue, we add a *topology verification* step to the SJ tool chain for parallel programs. Fig. 3 sum-

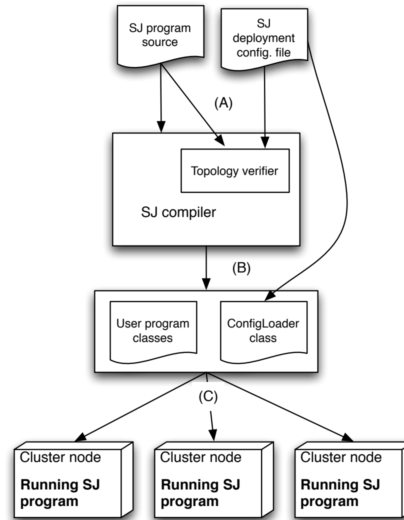



**Fig. 3.** The SJ tool chain.

marises the SJ tool chain for developing type-safe SJ parallel program on a distributed computing cluster. An SJ parallel program is written as a collection of SJ source files, where each file corresponds to a role in the topology. Topology verification (A) takes as input the source files and a *deployment configuration file*, listing the hosts where each process will be deployed and describing how to connect the processes. The sources and configuration files are then analysed statically to ensure the overall session topology of the parallel program conforms to a *well-formed topology* defined in Definition 4.1 in § 4, and in conjunction with session duality checks in SJ, precludes *global deadlocks* in parallel SJ programs (see Theorem 4.3). The source files are then compiled (B) to bytecode, and (C) deployed on the target cluster using details on the configuration file to instantiate and establish sessions with their assigned neighbours, ensuring the runtime topology is constructed according to the verified configuration file, and therefore safe execution of the parallel program.

## 3 Parallel Algorithms in SJ

This section presents the SJ implementation of a Jacobi method for solving the Discrete Poisson Equation and explains the benefits of the new multi-channel primitives. The example was chosen both as a representative real-world parallel programming application in SJ, and because it exemplifies a complex communication topology [7]. Implementations of other algorithms featuring other topologies, such as $n$-Body simulation (circular pipeline) and Linear Equation Solver (wraparound mesh), are available from [5].

**Jacobi solution of the discrete poisson equation: mesh topology.** Poisson's equation is a partial differential equation widely used in physics and the natural sciences. Jacobi's algorithm can be implemented using various partitioning strategies. An early session-typed implementation [1] used a one-dimensional decomposition of the source matrix, resulting in a linear communication topology. The following demonstrates how the new multi-channel primitives are required to increase parallelism using a two-dimensional decomposition, i.e. using a 2D mesh communication topology. The mesh topology is used in a range of other parallel algorithms [3].

The discrete two-dimensional Poisson equation $(\nabla^2 u)_{ij}$ for a $m \times n$ grid reads:

$$u_{ij} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - dx^2 g_{i,j})$$

where $2 \leq i \leq m-1$, $2 \leq j \leq n-1$, and $dx = 1/(n+1)$. Jacobi's algorithm converges on a solution by repeatedly replacing each element of the matrix $u$ by an adjusted average of its four neighbouring values and $dx^2 g_{i,j}$. For this example, we set each $g_{i,j}$ to 0. Then, from the $k$-th approximation of $u$, the next iteration calculates:

$$u_{ij}^{k+1} = \frac{1}{4}(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k)$$

Termination may be on reaching a target convergence threshold or on completing a certain number of iterations. Parallelisation of this algorithm exploits the fact that each element can be independently updated within each iteration. The decomposition divides the grid into subgrids, and each process will execute the algorithm for its assigned subgrid. To update the points along the boundaries of each subgrid, neighbouring processes need to exchange their boundary values at the beginning of each iteration.

```
protocol MasterToWorker
  cbegin.                 // Open a session with the Worker
  !<int>.!<int>.          // Send matrix dimensions
  ![                      // Main loop: checking convergence condition
    !<double[]>.          // Send our boundary values...
    ?(double[]).          // ..and receive our neighbour's
    ?(ConvergenceValues)  // Convergence data for neigbouring subgrid
  ]*                      // (end of main loop)
```

**Fig. 4.** The session type between the *Master* and *Workers* for the Jacobi algorithm.

A 2D mesh implementation is shown in Fig. 7. The *Master* node controls iteration from the top-left corner. Nodes in the centre of the mesh receive iteration control signals from their top and left neighbours, and propagate them to the bottom and right. Nodes at the edges only propagate iteration signals to the bottom or the right, and the final node at the bottom right only receives signals and does not propagate them further.

The session type for communication from the *Master* to either of the *Workers* under it or at its right is given in Fig. 4. The *Worker*'s protocol for interacting with the *Master* is the dual of *MasterToWorker*; the same protocol is used for interaction with other *Workers* at their right and bottom (except for *Workers* at the edges of the mesh).

As listed in Fig. 5, it is possible to express the complex 2D mesh using single-channel primitives only. However, this implementation suffers from a problem: without the multi-channel primitives, there is no way of sending iteration control signals both horizontally and vertically; the only option is to open and close a temporary session in every iteration (Fig. 7), an inefficient and counter-intuitive solution. Moreover, the continuous nature of the vertical iteration sessions cannot be expressed naturally.

Having noted this weakness, Fig. 6 lists a revised implementation, taking advantage of multi-channel `inwhile` and `outwhile`. The multi-channel `inwhile` allows each *Worker* to receive iteration signals from the two processes at its top and left. Multi-channel `outwhile` lets a process control both processes at the right and bottom. Together, these two primitives completely eliminate the need for repeated opening and closing of intermediary sessions in the single-channel version. The resulting implementation is clearer and also much faster. See § 5 for the benchmark results.

## 4 Multi-channel Session $\pi$-Calculus

This section formalises the new nested iterations and multi-channel communication primitives and proves correctness of our implementation. Our proof method consists of:

1. We first define programs (*i*.e. starting processes) including the new primitives, and then define operational semantics with running processes modelling intermediate session communications.
2. We define a typing system for programs and running processes.
3. We prove that if a group of running processes conforms to a *well-formed topology*, then they satisfy the subject reduction theorem (Theorem 4.1) which implies type and communication-safety (Theorem 4.2) and deadlock-freedom (Theorem 4.3).
4. Since programs for our chosen parallel algorithms conform to a well-formed topology, we conclude that they satisfy the above three properties.

*Master* :
```
right.outwhile(notConverged()) {
  under = chanUnder.request();
  sndBoundaryVal(right, under);
  rcvBoundaryVal(right, under);
  doComputation(rcvRight, rcvUnder);
  rcvConvergenceVal(right, under);
}
```
*Worker* :
```
right.outwhile(left.inwhile) {
  over = chanOver.accept();
  under = chanUnder.request();
  sndBoundaryVal(left,right,over,
      under);
  rcvBoundaryVal(left,right,over,
      under);
  doComputation(rcvLeft,rcvRight,
      rcvOver,rcvUnder);
  sndConvergenceVal(left,top);
}
```
*WorkerSE* :
```
left.inwhile {
  over = chanOver.request();
  sndBoundaryVal(left,over);
  rcvBoundaryVal(left,over);
  doComputation(rcvLeft,rcvOver);
  sndConvergenceVal(left,top);
}
```

**Fig. 5.** Initial 2D mesh implementation with single-channel primitives only.

*Master* :
```
<under,right>.outwhile(
    notConverged()) {
  sndBoundaryVal(right, under);
  rcvBoundaryVal(right, under);
  doComputation(rcvRight, rcvUnder
      );
  rcvConvergenceVal(right, under);
}
```
*Worker* :
```
<under,right>.outwhile
    (<over,left>.inwhile) {
  sndBoundaryVal(left,right,over,
      under);
  rcvBoundaryVal(left,right,over,
      under);
  doComputation(rcvLeft,rcvRight,
      rcvOver,rcvUnder);
  sndConvergenceVal(left,top);
}
```
*WorkerSE* :
```
<over,left>.inwhile {
  sndBoundaryVal(left,over);
  rcvBoundaryVal(left,over);
  doComputation(rcvLeft,rcvOver);
  sndConvergenceVal(left,top);
}
```

**Fig. 6.** Efficient 2D mesh implementation using multi-`outwhile` and multi-`inwhile`.

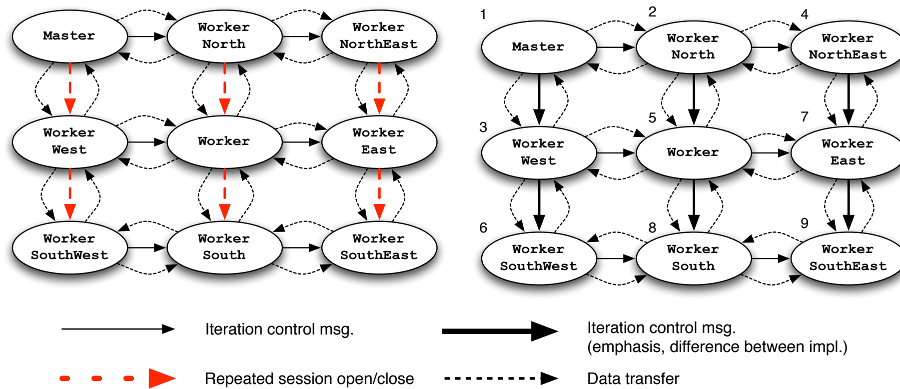



**Fig. 7.** Initial and improved communication patterns in the 2D mesh implementation.

### 4.1 Syntax

The session $\pi$-calculus we treat extends [8]. Fig. 8 defines its syntax. Channels $(u, u', \ldots)$ can be either of two sorts: *shared channels* $(a, b, x, y)$ or *session channels* $(k, k', \ldots)$. Shared channels are used to open a new session. In accepting and requesting processes, the name $a$ represents the public interaction point over which a session may commence. The bound variable $k$ represents the actual channel over which the session communications will take place. Constants $(c, c', \ldots)$ and expressions $(e, e', \ldots)$ of ground types (booleans and integers) are also added to model data. *Selection* chooses an available branch, and *branching* offers alternative interaction patterns; *channel send* and *channel receive* enable session delegation [8]. The *sequencing*, written $P; Q$, meaning that $P$ is executed before $Q$. This syntax allows for complex forms of synchronisation, joining, and forking since $P$ can include any parallel composition of arbitrary processes. The second addition is that of *multicast inwhile* and *outwhile*, following SJ syntax. Note that the definition of expressions includes multicast inwhile $\langle k_1 \ldots k_n \rangle.\mathtt{inwhile}$, in order to allow inwhile as an outwhile loop condition. The control message $k \dagger [b]$ created by outwhile appears only at runtime.

The precedence of the process-building operators is (from the strongest) "$\triangleleft, \triangleright, \{\}$", ".", ";" and "|". Moreover we define that "." associates to the right. The binders for channels and variables are standard.

(Values)

| | | |
|---|---|---|
| $v$ ::= | $a, b, x, y$ | shared names |
| \| | $\mathtt{true}, \mathtt{false}$ | boolean |
| \| | $n$ | integer |

(Expressions)

| | | |
|---|---|---|
| $e$ ::= | $v \mid e + e \mid \mathtt{not}(e) \ldots$ | value, sum, not |
| \| | $\langle k_1 \ldots k_n \rangle.\mathtt{inwhile}$ | inwhile |

(Processes)

| | | |
|---|---|---|
| $P$ ::= | $\mathbf{0}$ | inaction |
| \| | $T$ | prefixed |
| \| | $P \,;\, Q$ | sequence |
| \| | $P \mid Q$ | parallel |
| \| | $(\nu u)P$ | hiding |

(Declaration)

$D$ ::= $X(xk) = P$

(Prefixed processes)

| | | | | |
|---|---|---|---|---|
| $T$ ::= | $\overline{a}(k).P$ | request | $\mid \mathtt{def}\ D\ \mathtt{in}\ P$ | recursion |
| \| | $a(k).P$ | accept | $\mid k \triangleleft l$ | selection |
| \| | $\overline{k}\langle e \rangle$ | sending | $\mid k \triangleright \{l_1 : P_1 [] \cdots [] l_n : P_n\}$ | branch |
| \| | $k(x).P$ | reception | $\mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$ | conditional |
| \| | $k\langle k' \rangle$ | sending | $\mid \langle k_1 \ldots k_n \rangle.\mathtt{inwhile}\{Q\}$ | inwhile |
| \| | $k(k').P$ | reception | $\mid \langle k_1 \ldots k_n \rangle.\mathtt{outwhile}(e)\{P\}$ | outwhile |
| \| | $X[ek]$ | variables | $\mid k \dagger [b]$ | message |

**Fig. 8.** Syntax.

We formalise the reduction relation $\longrightarrow$ in Fig.8 up to the standard structural equivalence $\equiv$ with the rule $\mathbf{0}\,; P \equiv P$ based on [8]. Reduction uses the standard *evaluation contexts* defined as:

$$E ::= [\,] \mid E; P \mid E \mid P \mid (\nu u)E \mid \mathsf{def}\ D\ \mathsf{in}\ E \\ \mid \mathtt{if}\ E\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid \langle k_1 \ldots k_n \rangle.\mathtt{outwhile}(E)\{P\} \mid E + e \mid \cdots$$

We use the notation $\Pi_{i \in \{1..n\}} P_i$ to denote the parallel composition of $(P_1 \mid \cdots \mid P_n)$.

Rules [Link] is a session initiation rule where a fresh channel $k$ is created, then restricted because the leading parts now share the channel $k$ to start private interactions. Rule [Com] sends data. Rule [Lbl] selects the $i$-th branch, and rule [Pass] passes a session channel $k$ for delegation. The standard conditional and recursive agent rules [If1], [If2] and [Def] originate in [8].

Rule [Iw1] synchronises with $n$ asynchronous messages if they all carry $\mathtt{true}$. In this

$$
\begin{array}{lll}
a(k).P_1 \mid \overline{a}(k).P_2 \longrightarrow (\nu k)(P_1 \mid P_2) & \overline{k}\langle c\rangle \mid k(x).P_2 \longrightarrow P_2\{c/x\} & \text{[Link], [Com]} \\
k \rhd \{l_1 : P_1 [] \cdots [] l_n : P_n\} \mid k \lhd l_i \longrightarrow P_i \quad (1 \le i \le n) & k\langle k'\rangle \mid k(k').P_2 \longrightarrow P_2 & \text{[Lbl], [Pass]} \\
\texttt{if true then } P \texttt{ else } Q \longrightarrow P & \texttt{if false then } P \texttt{ else } Q \longrightarrow Q & \text{[If]} \\
\texttt{def } X(xk) = P \texttt{ in } X[ck] \longrightarrow \texttt{def } X(xk) = P \texttt{ in } P\{c/x\} & & \text{[Def]} \\
\langle k_1 \dots\ k_n\rangle.\texttt{inwhile}\{P\} \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] \longrightarrow P; \langle k_1 \dots\ k_n\rangle.\texttt{inwhile}\{P\} & & \text{[Iw1]} \\
\langle k_1 \dots\ k_n\rangle.\texttt{inwhile}\{P\} \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] \longrightarrow \mathbf{0} & & \text{[Iw2]} \\
E[\langle k_1 \dots k_n\rangle.\texttt{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] \longrightarrow E[\texttt{true}] & & \text{[IwE1]} \\
E[\langle k_1 \dots k_n\rangle.\texttt{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] \longrightarrow E[\texttt{false}] & & \text{[IwE2]} \\
E[e] \longrightarrow^* E'[\texttt{true}] \Rightarrow & & \\
\quad E[\langle k_1 \dots\ k_n\rangle.\texttt{outwhile}(e)\{P\}] \longrightarrow E'[P; \langle k_1 \dots\ k_n\rangle.\texttt{outwhile}(e)\{P\}] & & \\
\quad \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] & & \text{[Ow1]} \\
E[e] \longrightarrow^* E'[\texttt{false}] \Rightarrow & & \\
\quad E[\langle k_1 \dots\ k_n\rangle.\texttt{outwhile}(e)\{P\}] \longrightarrow E'[\,\mathbf{0}\,] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] & & \text{[Ow2]} \\
P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q' \equiv Q \Rightarrow P \longrightarrow Q & & \text{[Str]} \\
e \longrightarrow e' \Rightarrow E[e] \longrightarrow E[e'] & P \longrightarrow P' \Rightarrow E[P] \longrightarrow E[P'] & \\
P \mid Q \longrightarrow P' \mid Q' \Rightarrow E[P] \mid Q \longrightarrow E[P'] \mid Q' & & \text{[Eval]}
\end{array}
$$

In [Ow1] and [Ow2], we assume $E = E' \mid \Pi_{i\in\{1..n\}} k_i \dagger [b_i]$

**Fig. 9.** Reduction rules.

case, it repeats again. Rule [Iw2] is its dual and synchronises with $n$ `false` messages. In this case, it moves to the next command. On the other hand, if the results are mixed (i.e. $b_i$ is true, while $b_j$ is false), then it is stuck. In SJ, it will raise the exception, cf. § 2 (4). The rules for expressions are defined similarly. The rules for outwhile generates appropriate messages. Note that the assumption $E[e] \longrightarrow E'[\texttt{true}]$ or $E[e] \longrightarrow E'[\texttt{false}]$ is needed to handle the case where $e$ is an inwhile expression.

In order for our reduction rules to reflect SJ's actual behaviour, inwhile rules should have precedence over outwhile rules. Note that our algorithms do not cause an infinite generation of $k \dagger [b]$ by outwhile: this is ensured by the well-formed topology criteria described later, together with this priority rule.

### 4.2 Types, Typing System and Well-Formed Topologies

This subsection presents types and typing systems. The key point is an introduction of types and typing systems for asynchronous runtime messages. We then define the notation of a well-formed topology.

**Types.** The syntax of types, an extension of [8], follows:

$$
\begin{array}{lll}
\text{Sort} & S & ::= \mathsf{nat} \mid \mathsf{bool} \mid \langle \alpha, \overline{\alpha} \rangle \\
\text{Partial session} & \tau & ::= \varepsilon \mid \tau;\, \tau \mid ?[S] \mid ?[\alpha] \mid \&\{l_1\colon \tau_1, \dots, l_n\colon \tau_n\} \mid ![\tau]^* \mid \mathbf{x} \\
 & & \quad \mid ![S] \mid ![\alpha] \mid \oplus\{l_1\colon \tau_1, \dots, l_n\colon \tau_n\} \mid ?[\tau]^* \mid \mu\mathbf{x}.\tau \\
\text{Completed session} & \alpha & ::= \tau.\mathsf{end} \mid \bot \qquad \text{Runtime session} \quad \beta ::= \alpha \mid \alpha^\dagger \mid \dagger
\end{array}
$$

Sorts include a pair type for a shared channel and base types. The partial session type $\tau$ represents intermediate sessions. $\varepsilon$ represents inaction and $\tau;\tau$ is a sequential composition. The rest is from [8]. The types with ! and ? express respectively the sending

and reception of a value $S$ or session channel. The selection type $\oplus$ represents the transmission of the label $l_i$ followed by the communications described by $\tau_i$. The branching type & represents the reception of a label $l_i$ chosen in the set $\{l_1,\ldots,l_n\}$ followed by the communications described by $\tau_i$. Types $![\tau]^*$ and $?[\tau]^*$ are types for outwhile and inwhile. The types are considered up to the equivalence: $\&\{l_1 \colon \tau_1,\ldots,l_n \colon \tau_n\}.\mathsf{end} \equiv \&\{l_1 \colon \tau_1.\mathsf{end},\ldots,l_n \colon \tau_n.\mathsf{end}\}$. This equivalence ensures all partial types $\tau_1 \ldots \tau_n$ of selection ends, and are compatible with each other in the completed session type (and vice versa). $\varepsilon$ is an empty type, and it is defined so that $\varepsilon;\tau \equiv \tau$ and $\tau;\varepsilon \equiv \tau$.

Runtime session syntax represents partial composed runtime message types. $\alpha^\dagger$ represents the situation inwhile or outwhile are composed with messages; and $\dagger$ is a type of messages. The meaning will be clearer when we define the parallel composition.

**Judgements and environments.** The typing judgements for expressions and processes are of the shape:

$$\Gamma;\Delta \vdash e \triangleright S \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta$$

where we define the environments as $\Gamma ::= \emptyset \mid \Gamma \cdot x \colon S \mid \Gamma \cdot X : S\alpha$ and $\Delta ::= \emptyset \mid \Delta \cdot k \colon \beta$. $\Gamma$ is the *standard environment* which associates a name to a sort and a process variable to a sort and a session type. $\Delta$ is the *session environment* which associates session channels to running session types, which represents the open communication protocols. We often omit $\Delta$ or $\Gamma$ from the judgement if it is empty.

Sequential and parallel compositions of environments are defined as:

$$\Delta;\Delta' = \Delta \setminus \mathrm{dom}(\Delta') \cup \Delta' \setminus \mathrm{dom}(\Delta) \cup \{k \colon \Delta(k) \setminus \mathsf{end};\Delta'(k) \mid k \in \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta')\}$$
$$\Delta \circ \Delta' = \Delta \setminus \mathrm{dom}(\Delta') \cup \Delta' \setminus \mathrm{dom}(\Delta) \cup \{k \colon \Delta(k) \circ \Delta'(k) \mid k \in \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta')\}$$

where $\Delta(k) \setminus \mathsf{end}$ means we delete $\mathsf{end}$ from the tail of the types (e.g. $\tau.\mathsf{end} \setminus \mathsf{end} = \tau$). Then the resulting sequential composition is always well-defined. The parallel composition of the environments must be extended with new running message types. Hence $\beta \circ \beta'$ is defined as either (1) $\alpha \circ \overline{\alpha} = \bot$; (2) $\alpha \circ \dagger = \alpha^\dagger$ or (3) $\alpha \circ \overline{\alpha}^\dagger = \bot^\dagger$. Otherwise the composition is undefined. Here $\overline{\alpha}$ denotes a dual of $\alpha$ (defined by exchanging ! to ? and & to $\oplus$; and vice versa). (1) is the standard rule from session type algebra, which means once a pair of dual types are composed, then we cannot compose any processes with the same channel further. (2) means a composition of an iteration of type $\alpha$ and $n$-messages of type $\dagger$ becomes $\alpha^\dagger$. This is further composed with the dual $\overline{\alpha}$ by (3) to complete a composition. Note that $\bot^\dagger$ is different from $\bot$ since $\bot^\dagger$ represents a situation that messages are not consumed with inwhile yet.

**Typing rules.** We explain the key typing rules for the new primitives (Fig. 10). Other rules are similar with [8] and left to [5].

[EInwhile] is a rule for inwhile-expression. The iteration session type of $k_i$ is recorded in $\Delta$. This information is used to type the nested iteration with outwhile in rule [Outwhile]. Rule [Inwhile] is dual to [Outwhile]. Rule [Message] types runtime messages as $\dagger$. Sequential and parallel compositions use the above algebras to ensure the linearity of channels.

**Well-formed topologies.** We now define the well-formed topologies. Since our multi-channel primitives offer an effective, structured message passing synchronisation mechanism, the following simple definition is sufficient to capture deadlock-freedom in representative topologies for parallel algorithms. Common topologies in parallel algorithms such as circular pipeline, mesh and wraparound mesh all conform to our well-

$$\frac{\Delta = k_1 : ?[\tau_1]^*.\mathsf{end},...,k_n : ?[\tau_n]^*.\mathsf{end}}{\Gamma;\Delta \vdash \langle k_1 \dots k_n\rangle.\mathtt{inwhile} \triangleright \mathsf{bool}} \quad \frac{\Gamma \vdash b \triangleright \mathsf{bool}}{\Gamma \vdash k\dagger[b] \triangleright k : \dagger} \quad \text{[EINWHILE],[MESSAGE]}$$

$$\frac{\Gamma;\ \Delta \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \cdot k_1 :\ \tau_1.\mathsf{end} \cdot \dots \cdot k_n :\ \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \dots\ k_n\rangle.\mathtt{outwhile}(e)\{P\} \triangleright \Delta \cdot k_1 :\ ![\tau_1]^*.\mathsf{end},...,k_n :\ ![\tau_n]^*.\mathsf{end}} \quad \text{[OUTWHILE]}$$

$$\frac{\Gamma \vdash Q \triangleright \Delta \cdot k_1 :\ \tau_1.\mathsf{end} \cdot \dots \cdot k_n :\ \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \dots\ k_n\rangle.\mathtt{inwhile}\{Q\} \triangleright \Delta \cdot k_1 :\ ?[\tau_1]^*.\mathsf{end},...,k_n :\ ?[\tau_n]^*.\mathsf{end}} \quad \text{[INWHILE]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P;\ Q \triangleright \Delta;\Delta'} \quad \frac{\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \quad \text{[SEQ],[CONC]}$$

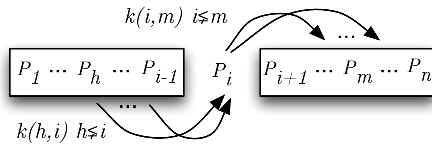
**Fig. 10.** Key typing rules

formed topology definition below [5]. Below we call $P$ is a *base* if $P$ is either $\mathbf{0}$, $\overline{k}\langle e\rangle$, $k(x).\mathbf{0}$, $k \triangleleft l$ or $k \triangleright \{l_1 : \mathbf{0}[] \cdots []l_n : \mathbf{0}\}$.

**Definition 4.1 (Well-formed topology.).** Suppose a group of $n$ parallel composed processes $P = P_1 \mid \dots \mid P_n$ such that $\Gamma \vdash P \triangleright \Delta$ with $\Delta(k) = \bot$ for all $k \in \mathrm{dom}(\Delta)$; and $k_{(i,j)}$ denotes a free session channel from $P_i$ to $P_j$. We say $P$ conforms to a *well-formed topology* if $P$ inductively satisfies one of the following conditions:

1. (inwhile and outwhile)
   $P_1 = \langle\vec{k}_1\rangle.\mathtt{outwhile}(e)\{Q_1\}$ $P_i = \langle\vec{k}_i\rangle.\mathtt{outwhile}(\langle\vec{k}'_i\rangle.\mathtt{inwhile})\{Q_i\}$ $(2 \le i < n)$
   $P_n = \langle\vec{k}'_n\rangle.\mathtt{inwhile}\{Q_n\}$ $\quad \vec{k}_i \subset k_{(i,i+1)} \cdots k_{(i,n)}, \vec{k}'_i \subset k_{(1,i)} \cdots k_{(i-1,i)}$
   and $(Q_1 \mid \cdots \mid Q_n)$ conforms to a well-formed topology.
2. (sequencing) $P_i = Q_{1i};...;Q_{mi}$ where $(Q_{j1} \mid Q_{j2} \mid \cdots \mid Q_{jn})$ conforms to a well-formed topology for each $1 \le j \le m$.
3. (base) (1) session actions in $P_i$ follow the order of the index (e.g. the session actions at $k_{(i,j)}$ happens before $k_{(h,g)}$ if $(i,j) < (h,g)$), then the rest is a base process $P'_i$; and (2) $P_i$ includes neither shared session channels, inwhile nor outwhile.

The figure below explains condition (1) of the above definition, ensuring consistency of control flows within iterations. Subprocesses $P_i$ are ordered by their process index $i$. A process $P_i$ can only send outwhile control messages to processes with a higher index via $\vec{k}_i$ (denoted by $k_{(i,m)}$), while it can receive messages from those with a lower index via $\vec{k}'_i$ (denoted by $k_{(h,i)}$). This ordering guarantees absence of cycles of communications.




There is only one source $P_1$ (only sends outwhile control messages) and one sink $P_n$ (only receives those messages). (2) says that a sequential composition of well-formed topologies is again well-formed. (3) defines base cases which are commonly found in the algorithms: (3-1) means that since the order of session actions in $P_i$ follow the order of the indices, $\Pi_i P_i$ reduces to $\Pi_i P'_i$ without deadlock; then since $\Pi_i P'_i$ is a parallel composition of base processes where each channel $k$ has type $\bot$, $\Pi_i P'_i$ reduces to $\mathbf{0}$ without deadlock. (3-2) ensures a single global topology.

### 4.3 Subject Reduction, Communication Safety and Deadlock Freedom

We state here that process groups conforming to a well-formed topology satisfy the main theorems. The full proofs can be found in [5].

**Theorem 4.1 (Subject reduction)** *Assume $P$ forms a well-formed topology and $\Gamma \vdash P \triangleright \Delta$. Suppose $P \longrightarrow^* P'$. Then we have $\Gamma \vdash P' \triangleright \Delta'$ with for all $k$ (1) $\Delta(k) = \alpha$ implies $\Delta'(k) = \alpha^\dagger$; (2) $\Delta(k) = \alpha^\dagger$ implies $\Delta'(k) = \alpha$; or (3) $\Delta(k) = \beta$ implies $\Delta'(k) = \beta$.*

(1) and (2) state an intermediate stage where messages are floating; or (3) the type is unchanged during the reduction. The proof requires to formulate the intermediate processes with messages which are started from a well-formed topology, and prove they satisfy the above theorem.

We say process has a *type error* if expressions in $P$ contains either a type error of values or constants in the standard sense (e.g. $\texttt{if}\ 100\ \texttt{then}\ P\ \texttt{else}\ Q$).

To formalise communication safety, we need the following notions. Write $\texttt{inwhile}(Q)$ for either $\texttt{inwhile}$ or $\texttt{inwhile}\{Q\}$. We say that a processes $P$ is a *head subprocess* of a process $Q$ if $Q \equiv E[P]$ for some evaluation context $E$. Then *$k$-process* is a head process prefixed by subject $k$ (such as $\overline{k}\langle e \rangle$). Next, a *$k$-redex* is the parallel composition of a pair of $k$-processes. i.e. either of form of a pair such that $(\overline{k}\langle e \rangle, k(x).Q)$, $(k \lhd l, k \rhd \{l_1 : Q_1 [] \cdots [] l_n : Q_n\})$, $(\overline{k}\langle k' \rangle, k(k').P)$, $(\langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{P\}, \langle k'_1 \ldots k'_m \rangle.\texttt{inwhile}(Q))$ with $k \in \{k_1, .., k_n\} \cap \{k'_1, .., k'_m\}$ or $(k \dagger [b] \mid \langle k'_1 \ldots k'_m \rangle.\texttt{inwhile}(Q))$ with $k \in \{k_1, .., k_n\}$. Then $P$ is a *communication error* if $P \equiv (\nu \tilde{u})(\texttt{def}\ D\ \texttt{in}\ (Q \mid R))$ where $Q$ is, for some $k$, the parallel composition of two or more $k$-processes that do not form a $k$-redex. The following theorem is direct from the subject reduction theorem [21, Theorem 2.11].

**Theorem 4.2 (Type and communication safety)** *A typable process which forms a well-formed topology never reduces to a type nor communication error.*

Below we say $P$ is *deadlock-free* if for all $P'$ such that $P \longrightarrow^* P'$, $P' \longrightarrow$ or $P' \equiv \mathbf{0}$ . The following theorem shows that a group of typable multiparty processes which form a well-formed topology can always move or become the null process.

**Theorem 4.3 (Deadlock-freedom)** *Assume $P$ forms a well-formed topology and $\Gamma \vdash P \triangleright \Delta$. Then $P$ is deadlock-free.*

Now we reason Jacobi algorithm in Fig. 6. We only show the master $P_1$ and the worker in the middle $P_5$ (the indices follow the right picture of Fig. 7).

$$
\begin{aligned}
P_1 &= \langle k_{(1,2)}, k_{(1,4)} \rangle.\texttt{outwhile}(e)\{\overline{k_{(1,2)}}\langle \texttt{d}[] \rangle; k_{(1,2)}(x).\overline{k_{(1,4)}}\langle \texttt{d}[] \rangle; k_{(1,4)}(y).\ \mathbf{0}\ \} \\
P_5 &= \langle k_{(5,7)}, k_{(5,8)} \rangle.\texttt{outwhile}(\langle k'_{(2,5)}, k'_{(3,5)} \rangle.\texttt{inwhile})\{ \\
&\qquad k'_{(2,5)}(w).\overline{k'_{(2,5)}}\langle \texttt{d}[] \rangle; k'_{(3,5)}(x).\overline{k'_{(3,5)}}\langle \texttt{d}[] \rangle; \overline{k_{(5,7)}}\langle \texttt{d}[] \rangle; k_{(5,7)}(y).\overline{k_{(5,8)}}\langle \texttt{d}[] \rangle; k_{(5,8)}(z).\ \mathbf{0}\ \}
\end{aligned}
$$

where $\texttt{d}[]$ denotes the type of array with double. We can easily prove they are typable and forms the well-formed topology satisfying the conditions (1) and (3) in Definition 4.1. Hence it is type and communication-safe and deadlock-free. [5] lists the full definition and more complex algorithms which conform to a well-formed topology.

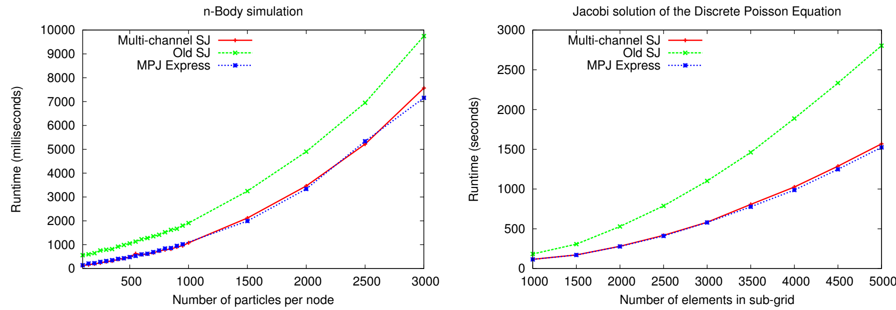



**Fig. 11.** SJ with and without multi-channel primitives and MPJ Express (left: 3-nodes $n$-Body simulation, right: 9-nodes Jacobi solution)

## 5 Performance Evaluation

This section presents performance results for several implementations of the $n$-Body simulation (details in [5, § A.1]), and Jacobi solution presented in § 3. We evaluated our implementations on a 9-node cluster for our benchmark, and each of the points is an average of 4 runs of the benchmark. All of them comprise an AMD PhenomX4 9650 2.30GHz CPU with 8GB RAM. The main objectives of these benchmarks is (1) to investigate the benefits of the new multi-channel primitives, comparing Old SJ (without the new primitives) and Multi-channel SJ (with the new primitives); and (2) compare those with MPJ Express [14] for reference. Fig. 11 shows a clear improvement when using the new multi-channel primitives in SJ. Multi-channel SJ also performs competitively against MPJ Express in both benchmarks. Hence SJ can be a viable alternative to MPI programming in Java, with the additional assurances of communication-safety and deadlock-free.

## 6 Related and Future Work

Due to space limitations, we focus on comparisons with extensions of Java with session types and MPI. Other related work, including functional languages with session types as well as HPC and PGAS languages can be found in the full version [5].

**Implementations of session types in Java.** SJ was introduced in [12] as the first general-purpose session-typed distributed programming language. Another recent extension of SJ added event-based programming primitives [11], for a different target domain: scalable and type-safe event-driven implementation of applications that feature a large number of concurrent but independent threads (e.g. Web servers). The preliminary experiments with parallel algorithms in SJ were reported in a workshop paper [1]. This early work considered only simple iteration chaining without analysis of deadlock-freedom, and without the general multi-channel primitives required for efficient representation of the complex topologies tackled here. The present paper also presents the formal semantics, type system, and proofs for type soundness and deadlock-freedom for the new primitives, which have not been studied in [1].

The Bica language [6] is an extension of Java also implementing binary sessions, which focuses on allowing session channels to be used as fields in classes. Bica does not support multi-channel primitives and does not guarantee deadlock-freedom across multiple sessions. See [10, 11] for more comparisons with [6]. A recent work [17] extends SJ-like primitives with multiparty session types and studies type-directed optimisations for the extended language. Their design is targeted at more loosely-coupled distributed applications than parallel algorithms, where processes are tightly-coupled and typically communicate via high-bandwidth, low-latency media; their optimisations, such as message batching, could increase latency and lower performance. It does not support features such as session delegation, session thread programming and transport independence [10, § 4.2.3], which are integrated into SJ. The latter in particular, together with SJ alias typing [10, § 3.1] (for session linearity), offers transparent portability of SJ parallel algorithm code over TCP and shared memory with zero-copy optimisations.

**Message-based parallel programming.** The present paper focuses on language and typing support for communications programming, rather than introducing a supplementary API. In comparison to the standard MPI libraries [7, §4], SJ offers structured communication programming from the natural abstraction of typed sessions and the associated static assurance of type and protocol safety. Recent work [19] applies model-checking techniques to standard MPI C source code to ensure correct matching of sends and receives using a pre-existing test suite. Their verifier, ISP, exploits independence between thread actions to reduce the state space of possible thread interleavings of an execution, and checks for deadlocks in the remaining states. In contrast, our session type-based approach does not depend on external testing, and a valid, compiled program is guaranteed communication-safe and deadlock-free in a matter of seconds. SJ thus offers a performance edge even in the cases of complex interactions (cf. [5]). The MPI API remains low-level, easily leading to synchronisation errors, message type errors and deadlocks [7]. From our experiences, programming message-based parallel algorithms with SJ are much easier than programming based on MPI functions, which, beside lacking type checking for protocol and communication safety, often requires manipulating numerical process identifiers and array indexes (e.g. for message lengths in the *n*-Body program) in tricky ways. Our approach gives a clear definition of a class of communication-safe and deadlock-free programs as proved in Theorems 4.2 and 4.3, which have been statically checked without exploring all execution states for all possible thread interleavings. Finally, benchmark results in §5 demonstrate how SJ programs can deliver the above benefits and perform competitively against a Java-based MPI [14].

**Future work.** Our previous work [20] shows type-checking for parallel algorithms based on parameterised multiparty sessions requires type equality, so type checking is undecidable in the general case. The method developed in this paper is not only decidable, but also effective in practice as we can reuse the existing binary SJ language, type-checker and runtime, with extensions to the new multi-channel inwhile and outwhile primitives for structuring message-passing communications and iterations. To validate more general communication topologies beyond the well-formed condition and typical parallel algorithms, we plan to incorporate new primitives into multiparty session types [9, 17] by extending the end-point projection algorithm based on roles [4]. Preliminary results from a manual SJ-to-C translation have shown large performance

gains for FPGA implementations [15]. Future implementation efforts will include a natively compiled, C-like language targeted at low overheads and efficiency for HPC and systems programming. We also plan to incorporate recent, unimplemented theoretical advances, including logical reasoning [2] to prove the correctness of parallel algorithms.

## References


1. A. Bejleri, R. Hu, and N. Yoshida. Session-Based Programming for Parallel Algorithms. In *PLACES*, EPTCS, 2009.
2. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 162–176.
3. H. Casanova, A. Legrand, and Y. Robert. *Parallel Algorithms*. Chapman & Hall, 2008.
4. P.-M. Deniélou and N. Yoshida. Dynamic Multirole Session Types. In *POPL '11*, pages 435–446. ACM, 2011.
5. On-line appendix. `http://www.doc.ic.ac.uk/~cn06/pub/2011/sj_parallel/`.
6. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular Session Types for Distributed Object-Oriented Programming. In *POPL '10*, pages 299–312. ACM, 2010.
7. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.
8. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP '98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
9. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008.
10. R. Hu. *Structured, Safe and High-level Communications Programming with Session Types*. PhD thesis, Imperial College London, 2010.
11. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-Safe Eventful Sessions in Java. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 329–353, 2010.
12. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In J. Vitek, editor, *ECOOP*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
13. Message Passing Interface. `http://www.mcs.anl.gov/research/projects/mpi/`.
14. MPJ Express homepage. `http://mpj-express.org/`.
15. N. Ng. High Performance Parallel Design based on Session Programming. MEng thesis, Department of Computing, Imperial College London, 2010.
16. A. Shafi, B. Carpenter, and M. Baker. Nested Parallelism for Multi-core HPC Systems using Java. *Journal of Parallel and Distributed Computing*, 69(6):532 – 545, 2009.
17. K. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient Session Type Guided Distributed Interaction. In *COORDINATION*, volume 6116 of *LNCS*, pages 152–167. Springer, 2010.
18. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-Throughput Stream Programming in Java. In *OOPSLA '07*, pages 211–228. ACM, 2007.
19. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal Verification of Practical MPI Programs. In *PPoPP '09*, pages 261–270. ACM, 2009.
20. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised Multiparty Session Types. In C.-H. L. Ong, editor, *FOSSACS*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.
21. N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *ENTCS*, 171(4):73–93, 2007.