Safe Parallel Programming with Session Java

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

*Imperial College London
†Department of Information Technology Services, Ministry of Finance, Cyprus

Abstract. The session-typed programming language Session Java (SJ) has proved to be an effective tool for distributed programming, promoting productivity 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 with static safety guarantees. We demonstrate that the new primitives yield clearer and safer code for pipeline, ring and mesh topologies, and through implementations of a parallel Jacobi method and an n-Body simulation. We then present a semantics and session typing system including the new primitives, and prove type soundness and deadlock-freedom for our parallel algorithm implementations. Finally, we benchmark several implementations of the n-Body simulation on a hybrid computing cluster, demonstrating the performance gains due to the new primitives. The SJ implementation is also substantially faster than an MPJ Express[1] implementation 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) [21] 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 π-calculus [17, 39], and have been shown to integrate cleanly with formal models of object-oriented programming [11, 14]. The SJ compiler offers two strong static guarantees: (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 [24]: implementations make the best use of hardware, but at the cost of large, complicated programs 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.

We investigate parallel programming in SJ as a solution to these issues. However, SJ as presented in [21] only guarantees safety for each session in isolation: deadlocks can

[1] MPJ Express [31] is a Java implementation of the MPI standard.

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. We need new constructs, well-integrated with existing binary sessions, to enable lightweight global *communication safety* and *progress*, and increase expressiveness to support *productivity* and *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 an additional static verification tool 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 our verification tool. 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 a static topology verifier for SJ programs. We show that the new primitives enable clearer, more readable code.
- **(§ 3)** We discuss our implementations of the *n*-Body (§ 3.1) algorithm, and of the Jacobi solution to the discrete Poisson equation (§ 3.2). Both algorithms use communication topologies representative of a large class of parallel algorithms, and demonstrate the practical use of our multi-channel primitives.
- **(§ 4)** We formally 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* (Theorem 4.3).
- **(§ 5)** A performance evaluation of *n*-Body implementations, demonstrating the benefits of the new primitives. The SJ implementation using the new primitives consistently outperforms an MPJ Express[2] [25] implementation.

Related and future work discussed in § **6**. Detailed benchmark results, omitted proofs and source code can be found at `http://www.doc.ic.ac.uk/~omp08`.

2 Session-Typed Programming in SJ

This section firstly reviews the key concepts of session-typed programming using Session Java (SJ) [20, 21]. In (1), we outline the basic methodology, and (2) the protocol structures supported by SJ. We then introduce the new session programming features developed by 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. (5) describes

[2] Extensive benchmarks comparing MPJ Express to other MPI implementations are presented in [31]. The benchmarks show performance competitive with C-based MPICH2.

aliasing control in SJ for zero-copy messaging in shared memory sessions. Finally, (6) describes the *topology verifier* for parallel algorithms, and (7) summarises the SJ toolchain.

(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 [21], 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 following code example implements an Alice process conforming to the `P` protocol:

```
sess.send(42); Data d = (Data) sess.receive(); // !<int>.?(Data)
```

The `sess` 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 implementing 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 Bob process implementing protocol `PDual` can be written as:

```
int i = s.receiveInt(); s.send(new Data()); // ?(int).!<Data>
```

(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 Figure 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 `![`τ`]*` type given by `IntAndBoolStream`, where τ is the `!{Label1:` τ_1`, Label2:` τ_2`, ...}` type implemented by the latter. Processes *P2* and *P3* passively follow the selected branch and the iteration decisions (received as internal control messages)


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

P1: s.outwhile(x < 10) {          // s follows IntAndBoolStream
      s.outbranch(Label1) {
        s.send(42);
    }}
P2: s2.outwhile(s1.inwhile()) {   // s1 follows IntAndBoolDual
      s1.inbranch() {             // s2 follows IntStream
        case Label1:
          int i = s1.receiveInt();
          s2.send(i);
        case Label2:
          boolean b = s1.receiveBoolean();
          s2.send(42);
    }}
P3: s.inwhile {                   // s follows IntStreamDual
      int i = s.receiveInt();
    }
```

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

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 [21] derived from standard session type theory [17, 39]; 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 Figure 1 additionally illustrate session *iteration chaining*, forming a linear pipeline as depicted at the top of the Figure. 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 [11] must forbid operations on any session other than the loop target, 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. Section 4 shall formalise and prove the correctness of this new feature.

(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

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

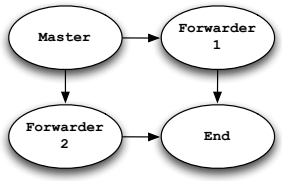


Fig. 2: Multi-channel iteration in a simple grid topology.

pipeline is a powerful benefit over traditional stream-based dataflow [33]. More complex topologies, however, such as rings and meshes, require iteration signals to be directly 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.

Figure 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) Aliasing control and session communication. The SJ language features the `noalias` modifier for compile-time aliasing control. In summary, any object reference stored in a `noalias` variable is guaranteed, through a combination of static typing restrictions and adapted operational semantics, to be the *sole* reference to that object. For example, typing disallows non-`noalias` (i.e. ordinary Java) variables as assignment expressions to a `noalias` variable; and a method invocation with a `noalias` variable as an argument expression causes the variable to become `null` (i.e. be consumed) at runtime. A detailed account of aliasing control in SJ is available in [19, § 3.11]

Aliasing control is a crucial element of session typing in SJ, enabling the sequence of primitives performed via a session socket variable (implicitly `noalias`) to be determined, and hence checked against the protocol, at compile-time. However, `noalias` can in turn be integrated with session communication to support zero-copy messaging optimisation for shared memory sessions, as in the case of parallel programming for multi-core machines. The basic idea is that any `noalias` object passed to the `send` operation can effectively be passed by reference without copying the object. Unlike [12, 33, 34], SJ `noalias` message passing supports more general Java objects; and a key point is that *transport-independent* sessions promote this optimisation *transparently* in shared memory environments while retaining consistent communication semantics in distributed environments.

(6) Session topology verifier for parallel programs. In previous work, the static safety guarantees offered by the SJ compiler were limited to scope of each independent *binary* (two-party) session. This means that, while any one session was e.g. 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 need to be correctly composed.

As a solution to this issue, we add a separate topology verifier to the SJ toolchain. The verifier statically ensures that session processes are configured according to the correct topology, and, in conjunction with the SJ compiler, precludes *global* deadlocks. Section 4 characterises a notion of correct topology and proves its correctness for mesh and ring topologies, used in the examples in § 3. The verifier takes as input a configuration file listing the physical nodes where each process class is to be deployed and the runtime links between these processes. At deployment time, each process uses the `ConfigLoader` utility class to load its parameters from the configuration file, ensuring that the run-time topology follows the static specification.

(7) Running SJ parallel programs. Figure 3 summarises the steps involved in the deployment of a type-safe SJ parallel program on a distributed computing cluster.

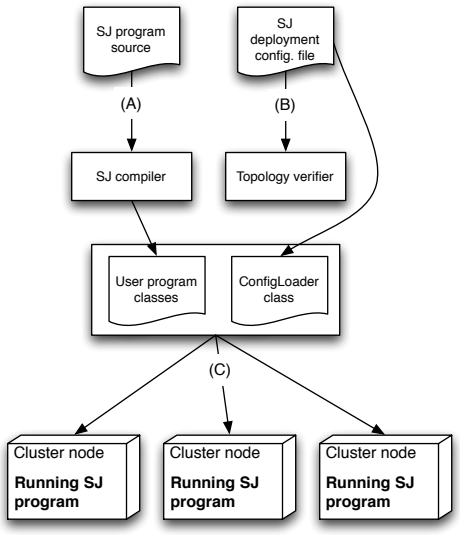


Fig. 3: The SJ parallel programming toolchain.

The program is written as a collection of SJ classes, one for each type of process in the program to be deployed. Processes differ by their position in the network topology, and their role in the coordination of the overall parallel algorithm. The user also writes a configuration file describing how to connect the process classes. When classes are ready, they are (A) compiled into standard bytecode by the SJ compiler, which checks the correct implementation of each binary session. After compiling the SJ source, the topology verifier is used to (B) check the topology declared in the configuration file. Finally, the verified, compiled classes are (C) deployed on the cluster, and make use of the `ConfigLoader` utility class from the SJ libraries to establish sessions with their assigned neighbours in the configuration file, ensuring safe execution of the parallel program.

3 Parallel Algorithms in SJ

This section demonstrates SJ session programming and the application of session types for structured, safe parallel programming. We present the implementation of two advanced parallel algorithms, an n-Body simulation and a Jacobi method for the Discrete Poisson Equation. These examples were chosen both as representative real-world paral-

lel programming applications, and because they exemplify two important communication topologies, the ring and the two-dimensional mesh. We compare implementations with and without the new multi-channel primitives, and explain the benefits they bring to parallel programming.

3.1 *n*-Body: Ring Topology

The parallel *n*-Body algorithm organises the constituent processes into a circular pipeline, an example of the ring communication topology. The ring topology is used by many other parallel algorithms, like matrix multiplication and LU matrix decomposition [5].

The *n*-Body problem involves finding the motion, according to classical mechanics, of a system of particles given their masses and initial positions and velocities. Parallelism in the simulation algorithm is achieved by dividing the particle set among a set of m worker processes. The idea is that each simulation step involves a series of inner steps, which perform a running computation while forwarding the data from each process around the ring one hop at a time; after $m-1$ inner steps, each process has seen all the data from every other, and the current simulation step is complete.

The following session type describes the communication protocol of our implementation. This is the session type for a *Worker*'s interaction with its left neighbour.

```
protocol WorkerToLeft
  sbegin.      // Accept session request from left neighbour
  !<int>.      // Forward init counter to determine number of processes
  ?[                  // Main loop (loop controlled by left neighbour)
   ?[                 // Pipeline stages within each simulation step
     !<Particle[]>    // Pass current particle state along the ring
   ]* ]*
```

The interaction with the right neighbour follows the dual protocol. The *WorkerLast* and *Master* nodes follow slightly different protocols, in order to close the ring structure and bootstrap the pipeline interaction.

In the SJ implementation, each node establishes two sessions with the left and right neighbours, and the iteration of every session in the pipeline is centrally controlled by the *Master* node. Without the multi-channel iteration primitives, there is no adequate way of closing the ring (sending data from the *WorkerLast* node to the *Master*); the only option is to open and close a temporary session with each iteration (Figure 4) [2], an inefficient and counter-intuitive solution, as depicted on the left in Figure 6 (the loosely dashed line indicates the temporary connection).

By contrast, Figure 5 gives the implementation of the ring topology using a multi-`outwhile` at the *Master* node, and a multi-`inwhile` at *WorkerLast*. Data is still passed left-to-right, but the final iteration control link (the bold arrow on the right in Figure 6) is reversed. This allows the *Master* to create the final link just once (at the start of the algorithm) like the other links, and gives the *Master* full control over the whole pipeline.

3.2 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 [2] 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

```
Master :
    right.outwhile(condition) {
      processData();
      left = chanLast.accept();
      newData = left.receive();
      right.send(data);
    }
Worker :
    right.outwhile(left.inwhile) {
      processData();
      newData = left.receive();
      right.send(data);
    }
WorkerLast :
    left.inwhile {
      processData();
      newData = left.receive();
      right = chanLast.request();
      right.send(data);
    }
```

Fig. 4: Implementation of the ring topology, single-channel primitives only.

```
Master :
  <left,right>.outwhile(moreNodes)
       {
    newData = left.receive();
    right.send(data);
  }
Worker :
  right.outwhile(left.inwhile) {
    newData = left.receive();
    right.send(data);
  }
WorkerLast :
  <left,right>.inwhile {
    newData = left.receive();
    right.send(data);
  }
```

Fig. 5: Improved implementation of the ring topology using multi-channel primitives.

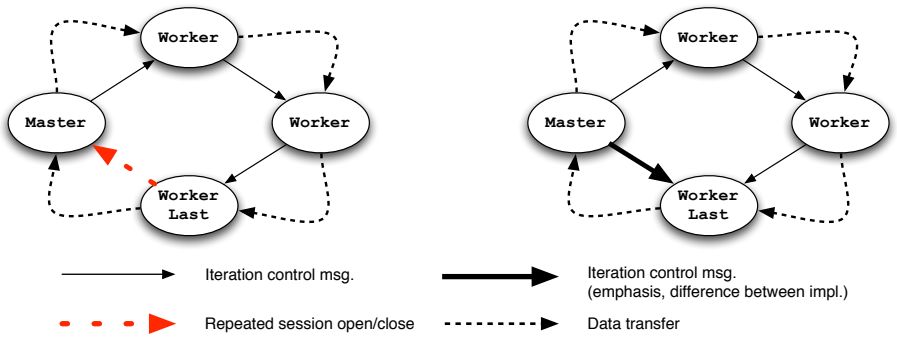


Fig. 6: Communication patterns in n-Body implementations.

using a two-dimensional decomposition, i.e. using a 2D mesh communication topology. The mesh topology is also used in a range of other parallel algorithms [5].

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 sub-

```
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. 7: The session type between the Master and Worker processes for the parallel Jacobi Poisson algorithm.

grid. To update the points along the boundaries of each subgrid, neighbouring processes need to exchange their boundary values at the beginning of each iteration.

The session type for communication from the *Master* to either of the *Workers* under it or at its right is given in Figure 7. The *Worker*'s protocol for interacting with the *Master* is the dual of *MasterToWorker*; and the same protocol is used for interaction with other *Workers* at their right and bottom (except for *Workers* at the edges of the mesh).

Square grid decomposition and topology. As described above, we use a two-dimensional decomposition of the Jacobi algorithm: a 2D mesh allows a greater degree of parallelism to be exploited, giving improved performance in comparison to the one-dimensional decomposition studied in previous work [2]. In the 2D mesh implementations, a 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.

Implementing a 2D mesh using single-channel SJ primitives only. As listed in Figure 8, it is possible to express the complex 2D mesh using single-channel primitives only. However, this implementation suffers from the same problem as the original ring implementation: without the multi-channel primitives, costly extra sessions have to be opened and closed in every iteration (Figure 10). This problem is exacerbated by the large number of connections in the 2D mesh (p^2, as opposed to p for n-Body, where p is the number of processes).

Efficient 2D mesh implementation using multi-channel primitives. Having noted the weakness of the above implementation, Figure 9 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, as in the n-Body case. [22] presents benchmark results for the parallel Jacobi implementation.

4 Multi-channel Session π-Calculus

This section starts by formalising the new nested iterations and multi-channel communication primitives, as an extension of the session π-calculus [10, 17]. The calculus is

```
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. 8: 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. 9: Efficient 2D mesh implementation using multi-`outwhile` and multi-`inwhile`.

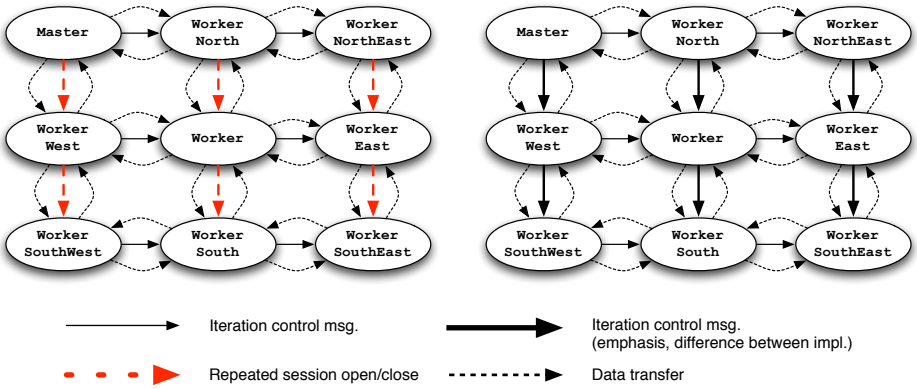


Fig. 10: Initial and improved communication patterns in the 2D mesh implementation.

concise but accurately captures parallel programming idioms such as fork/join and synchronisation, using sequencing, parallel composition, and the new iteration primitives. We then prove that our n-Body and Jacobi algorithms are type-safe, communication-safe and deadlock-free. Our proof method essentially follows the SJ toolchain of Figure 3, and consists of the following steps:

1. We first define programs (*i*.e. starting processes) including the new primitives, and then define operational semantics with running processes modeling 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 n-Body and Jacobi algorithms conform to a well-formed topology, we conclude that they satisfy the above three properties.

4.1 Syntax

The session π-calculus we treat extends [10, 17]. Figure 11 defines its syntax. Channels $(u, u', ...)$ can be either of two sorts: *shared channels* (a, b, x, y) or *session channels* $(k, k', ...)$. Shared channels are used to initiate sessions; session channels are only used within open sessions. In accepting and requesting processes, the identifier 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 when the session has been opened and the connection established. Constants $(c, c', ...)$ and expressions $(e, e', ...)$ 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, as described in [17]. The first difference with [17] is the addition of *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 $\langle k_1 \dots k_n \rangle.\texttt{inwhile}$, in order to allow inwhile as an outwhile loop condition. The control messages created by outwhile appear only at runtime.

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

We formalise the operational semantics of the calculus by the reduction relation $\longrightarrow$, defined in Figure 11 up to the standard structural equivalence $\equiv$ plus the rule $\mathbf{0}\,; P \equiv P$. The reduction rules are based on those of the session π-calculus [17], taking into account the behaviour of sequencing and of the new iteration primitives. Reduction uses *evaluation contexts* defined as:

$$\begin{aligned} E ::= {} & [] \mid E; P \mid E \mid P \mid (\nu u)E \mid \mathsf{def}\ D\ \mathsf{in}\ E \\ & \mid \texttt{if}\ E\ \texttt{then}\ P\ \texttt{else}\ Q \mid \langle k_1 \dots k_n \rangle.\texttt{outwhile}(E)\{P\} \mid E + e \mid \cdots \end{aligned}$$

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.

(Values)

$$
\begin{array}{rcll}
v & ::= & a,b,x,y & \text{shared names} \\
& | & \texttt{true},\texttt{false} & \text{boolean} \\
& | & n & \text{integer}
\end{array}
$$

(Processes)

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} & \text{inaction} \\
& | & T & \text{prefixed} \\
& | & P \,;\, Q & \text{sequential} \\
& | & P \mid Q & \text{parallel} \\
& | & (\nu u)P & \text{hiding} \\
& | & \texttt{Err} & \text{error}
\end{array}
$$

(Declaration)

$$
\begin{array}{rcl}
D & ::= & X_1(x_1k_1) = P_1 \texttt{ and} \\
& & X_2(x_2k_2) = P_2 \texttt{ and} \cdots \\
& & X_n(x_nk_n) = P_n
\end{array}
$$

(Expressions)

$$
\begin{array}{rcll}
e & ::= & v \mid e+e \mid \texttt{not}(e) \ldots & \text{value, sum, not} \\
& | & \langle k_1 \ldots k_n \rangle.\texttt{inwhile} & \text{inwhile} \\
& | & \texttt{Err} & \text{error}
\end{array}
$$

(Prefixed processes)

$$
\begin{array}{rcll}
T & ::= & \overline{a}(k).P & \text{request} \\
& | & a(k).P & \text{acceptance} \\
& | & \overline{k}\langle e \rangle & \text{sending} \\
& | & k(x).P & \text{reception} \\
& | & k \lhd l & \text{selection} \\
& | & k \rhd \{l_1 : P_1 [] \cdots [] l_n : P_n\} & \text{branching} \\
& | & k\langle k' \rangle & \text{session sending} \\
& | & k(k').P & \text{session reception} \\
& | & \texttt{if } e \texttt{ then } P \texttt{ else } Q & \text{conditional} \\
& | & X[ek] & \text{variables} \\
& | & \texttt{def } D \texttt{ in } P & \text{recursion} \\
& | & \langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{Q\} & \text{multi-inwhile} \\
& | & \langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{P\} & \text{multi-outwhile} \\
& | & k \dagger [b] & \text{messages}
\end{array}
$$

Fig. 11: Syntax.

$$
\begin{array}{rcll}
a(k).P_1 \mid \overline{a}(k).P_2 & \longrightarrow & (\nu k)(P_1 \mid P_2) & (k \text{ is fresh}) \quad [\textsc{Link}] \\
\overline{k}\langle c \rangle \mid k(x).P_2 & \longrightarrow & P_2\{c/x\} & [\textsc{Com}] \\
k \lhd l_i \mid k \rhd \{l_1 : P_1 [] \cdots [] l_n : P_n\} & \longrightarrow & P_i & (1 \le i \le n) \quad [\textsc{Lbl}] \\
k\langle k' \rangle \mid k(k').P_2 & \longrightarrow & P_2 & [\textsc{Pass}] \\
\multicolumn{4}{l}{\texttt{if true then } P \texttt{ else } Q \longrightarrow P \qquad \texttt{if false then } P \texttt{ else } Q \longrightarrow Q} & [\textsc{If1}], [\textsc{If2}] \\
\texttt{def } X(xk) = P \texttt{ in } X[ck] & \longrightarrow & \texttt{def } X(xk) = P \texttt{ in } P\{c/x\} & [\textsc{Def}] \\
\langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{P\} \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] & \longrightarrow & P; \langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{P\} & [\textsc{Iw1}] \\
\langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{P\} \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] & \longrightarrow & \mathbf{0} & [\textsc{Iw2}] \\
\langle k_1 \ldots k_n \rangle.\texttt{inwhile}\{P\} \mid \Pi_{i\in\{1..n\}} k_i \dagger [b_i] & \longrightarrow & \texttt{Err} & (\text{otherwise}) \quad [\textsc{Iw3}] \\
E[\langle k_1 \ldots k_n \rangle.\texttt{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] & \longrightarrow & E[\texttt{true}] & [\textsc{IwE1}] \\
E[\langle k_1 \ldots k_n \rangle.\texttt{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] & \longrightarrow & E[\texttt{false}] & [\textsc{IwE2}] \\
E[\langle k_1 \ldots k_n \rangle.\texttt{inwhile}] \mid \Pi_{i\in\{1..n\}} k_i \dagger [b_i] & \longrightarrow & E[\texttt{Err}] & (\text{otherwise}) \quad [\textsc{IwE3}] \\
E[e] \longrightarrow^* E'[\texttt{true}] \Rightarrow & & & \\
\quad E[\langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{P\}] & \longrightarrow & E'[P; \langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{P\}] & \\
& & \quad \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{true}] & [\textsc{Ow1}] \\
E[e] \longrightarrow^* E'[\texttt{false}] \Rightarrow & & & \\
\quad E[\langle k_1 \ldots k_n \rangle.\texttt{outwhile}(e)\{P\}] & \longrightarrow & E'[\,\mathbf{0}\,] \mid \Pi_{i\in\{1..n\}} k_i \dagger [\texttt{false}] & [\textsc{Ow2}] \\
P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q' \equiv Q & \Rightarrow & P \longrightarrow Q & [\textsc{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' & [\textsc{Eval}]
\end{array}
$$

In [Iw2] and [Ow2], we assume $E = E' \mid \Pi_{i\in\{1..n\}} k_i \dagger [b_i]$

Fig. 12: Reduction rules.

Rule [Com] sends data ($e \downarrow c$ means that the expression e evaluates to the value c). 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 [17].

Rule [Iw1] synchronises with n asynchronous messages if they all carry `true`. In this 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 reduces to the error state by rule [Iw3], as explained in § 2 (4). The rules for expressions are defined similarly. The rules for outwhile generates appropriate messages. Note that the assumption $E[e] \longrightarrow E'[b]$ 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 formulate a well-formed topology.

Types. The syntax of types, an extension of [17], follows:

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

Sorts include a pair type for a shared channel and base types. Partial session type τ represents intermediate sessions. ε represents inaction and $\tau;\tau$ is a sequential composition. The rest is from [17]. 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 τ_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 τ_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). ε 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.

Duality. To ensure communication compatibility, each session type has its dual type. A simple example is $![\mathsf{bool}].\mathsf{end}$ and $?[\mathsf{bool}].\mathsf{end}$. The two types are dual so that sending of a bool matches with receiving of a bool. If the receiver is changed to $?[\mathsf{bool}];?[\mathsf{bool}].\mathsf{end}$ then a communication mismatch happens. A typing defined next can ensure such incompatibilities between two communicating parties does not happen. It is defined as: $\overline{\varepsilon} = \varepsilon$, $\overline{\tau;\,\tau} = \overline{\tau};\,\overline{\tau}$, $\overline{![S]} = ?[S]$, $\overline{?[S]} = ![S]$, $\overline{\oplus\{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}} = \&\{l_i\colon \overline{\tau_i} \ldots, l_n\colon \overline{\tau_n}\}$, $\overline{\&\{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}} = \oplus\{l_i\colon \overline{\tau_i} \ldots, l_n\colon \overline{\tau_n}\}$, $\overline{![\tau]^*} = ?[\overline{\tau}]^*$, $\overline{?[\tau]^*} = ![\overline{\tau}]^*$, $\overline{\mathbf{x}} = \mathbf{x}$, $\overline{\mu\mathbf{x}.\tau} = \mu\mathbf{x}.\overline{\tau}$ $\overline{\alpha^\dagger} = \overline{\alpha}^\dagger$, $\overline{\dagger} = \dagger$, and $\overline{\tau.\mathsf{end}} = \overline{\tau}.\mathsf{end}$.

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 $\Delta ::= \emptyset \mid \Delta \cdot k\colon \beta$. Γ is the *standard environment* which associates shared channel names to shared session types and process variable to process type. Δ is the *session environment* which associates session channels to running session types, which represents the open communication protocols. We often omit Δ or Γ 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 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 undefined. (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 α 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. Other rules are similar with and left to Appendix A (Figure 14).

$$\frac{\Delta = k_1\colon ?[\tau_1]^*.\mathsf{end}, ..., k_n\colon ?[\tau_n]^*.\mathsf{end}}{\Gamma;\Delta \vdash \langle k_1 \ldots k_n \rangle.\mathtt{inwhile} \triangleright \mathsf{bool}} \quad \frac{\Gamma \vdash b \triangleright \mathsf{bool}}{\Gamma \vdash k \dagger [b] \triangleright k\colon \dagger} \qquad \text{[EInwhile],[Message]}$$

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

$$\frac{\Gamma \vdash Q \triangleright \Delta \cdot k_1\colon\ \tau_1.\mathsf{end} \cdot \cdots \cdot k_n\colon\ \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots\ k_n \rangle.\mathtt{inwhile}\{Q\} \triangleright \Delta \cdot k_1\colon\ ?[\tau_1]^*.\mathsf{end}, ..., k_n\colon\ ?[\tau_n]^*.\mathsf{end}} \qquad \text{[Inwhile]}$$

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

[EInwhile] is a rule for inwhile-expression. The iteration session type of k_i is recorded in Δ. 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.

Definition 4.1 (Well-formed topology.). *A group of N processes $P_{1,1} \mid \ldots \mid P_{R,N}$ conforms to a* well-formed topology *if (we write $Q[\vec{k}]$ if Q's free session channels are $\vec{k}$):*

$$
\begin{aligned}
P_{1,1} &= \langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\} \\
P_{r,i} &= \langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{ \\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\} \textit{ when } i<S \\
P_{R,i} &= \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\} \qquad \textit{when } S<i\leq N
\end{aligned}
$$

with $2 \leq i \leq N$ and $2 \leq r \leq R$, and there exists a bijective function f such that (1) (starter) $f(k_{1,j}) = k_{r'_{i'},j'}$ or (2) (inductive case) $f(k_{r_i,j}) = k'_{r'_{i'},j'}$ with $r < r'$, $1 \leq j \leq n_{r,i}$ and $1 \leq j' \leq n_{r',i'}$

$$
\begin{aligned}
\textit{where}\quad & \Gamma \vdash Q_1 \triangleright \{k_{1,1}\colon T_{1,1},\ldots,k_{1,n_{1,1}}\colon T_{1,n_{1,1}}\} \\
& \Gamma \vdash Q_i \triangleright \{k_{r_i,1}\colon T_{r_i,1},\ldots,k_{r_i,n_{r,i}}\colon T_{r_i,n_{r,i}},k'_{r'_{i'},1}\colon T_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\colon T_{r'_{i'},n_{r',i'}}\} \textit{ when } i<S \\
& \Gamma \vdash Q_i \triangleright \{k_{R_i,1}\colon T_{R_i,1},\ldots,k_{R_i,n_{R,i}}\colon T_{R_i,n_{R,i}}\} \textit{ when } S<i\leq N \\
& \Gamma \vdash Q_1 \mid Q_2 \mid \ldots \mid Q_n \triangleright \{\tilde{k}\colon \tilde{\perp}\}
\end{aligned}
$$

with $T_{1,j} = \overline{T_{r'_{i'},j'}}$ and $T_{r_i,j} = \overline{T_{r'_{i'},j'}}$.

Subprocesses $P_{r,i}$ are indexed by their *rank* r and their process index i. The processes form a *directed acyclic graph*: they can only send outwhile control messages to processes of a strictly higher rank (condition $r < r'$). This guarantees the absence of cycles, necessary to prove deadlock-freedom. The DAG is only allowed one source (process $P_{1,1}$), but can have one to many sinks. Sinks have process indexes greater than S. This also ensures control messages are always consistent as they flow from a single source, and is essential in proving that well-formed processes never reduce to `Err`.

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.

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 do not contain either a type error of values or constants in the standard sense (e.g. `if` 100 `then` P `else` Q).

To formalise communication safety, we need the following notions. Write $\texttt{inwhile}(Q)$ for either `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 $(\overline{k}\langle e\rangle, k(x).Q)$, $(k \triangleleft l, k \triangleright \{l_1 : Q_1 [] \cdots [] l_n : Q_n\})$, $(\overline{k}\langle k'\rangle, k(k').P)$, $(\langle k_1 \ldots k_n\rangle.\texttt{outwhile}(e)\{P\}, \langle k'_1 \ldots k'_m\rangle.\texttt{inwhile}(Q))$ 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 [39, 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 multiparty processes which form a single well-formed topology whose bodies in inwhile and outwhile are inductively deadlock-free can always move or become the null process.

Theorem 4.3 (Deadlock-freedom) *Assume G forms a well-formed topology and $\Gamma \vdash G \triangleright \Delta$. Suppose a parallel composition of subprocesses $\Pi_{i \in \{1,..,n\}} Q_i$ (bodies inside in and outwhile) in the definition of the well-formed topology do not contain in or outwhile and are deadlock-free. Then P is deadlock-free.*

Now we write an implementation of n-Body algorithm (omitting the tailing processes).

$$P_1 \equiv \langle k_{12}, k_{13}\rangle.\texttt{outwhile}(e)\{\overline{k_{12}}\langle \texttt{Particle}[]\rangle \mid k_{13}(y).\ \mathbf{0}\ \}$$
$$P_2 \equiv k_{23}.\texttt{outwhile}(k_{12}.\texttt{inwhile})\{\overline{k_{23}}\langle \texttt{Particle}[]\rangle \mid k_{12}(y).\ \mathbf{0}\ \}$$
$$P_3 \equiv \langle k_{13}, k_{23}\rangle.\texttt{inwhile}\{\overline{k_{13}}\langle \texttt{Particle}[]\rangle \mid k_{23}(y).\ \mathbf{0}\ \}$$

where the typing of the processes are (omitting end): $\Gamma \vdash P_1 \triangleright \{k_{12} : ![![U]]^*,\ k_{13} : ![?[U]]^*\}$, $\Gamma \vdash P_2 \triangleright \{k_{23} : ![![U]]^*,\ k_{12} : ?[?[U]]^*\}$, and $\Gamma \vdash P_3 \triangleright \{k_{13} : ?[![U]]^*,\ k_{23} : ?[?[U]]^*\}$. Similarly we can write an implementation of Jacobi algorithm. Then we have:

Proposition 4.4 (Correctness of n-Body and Jacobi) *The n-Body and Jacobi algorithms in Figures 5 and 9 are type-safe, communication-safe and deadlock-free.*

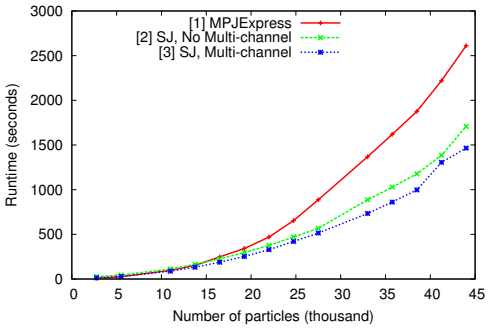
For the proof, we first prove the above implementation conforms the well-formed ring (mesh) topology and then shows the well-formed ring (mesh) topology is a well-formed topology (see Appendix A.1). [27, § 4.7] lists the detailed proofs.

5 Performance Evaluation

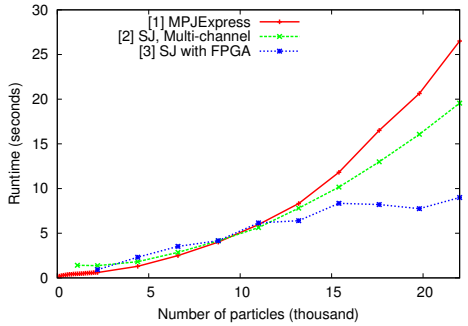
This section presents performance results for several implementations of the n-Body algorithm, as presented in § 3.1.

We evaluated our implementations on the Axel heterogeneous computing cluster [35]. This cluster is composed of nodes including a multi-core CPU, a general-purpose GPU, and an FPGA, and was built to investigate the combined use of these heterogeneous computing resources for compute-intensive applications. We used 11 nodes for our benchmark. All nodes include an Alpha-Data ADM-XRC-5T2 FPGA. Nine of them comprise an AMD PhenomX4 9650 2.30GHz CPU with 8GB RAM. The other two nodes have two Intel Xeon E5420 2.50GHz CPUs each, and respectively 16 and 12GB of RAM. Nodes are interconnected via Gigabit Ethernet. Each of the 11 nodes processed an equal share of particles.

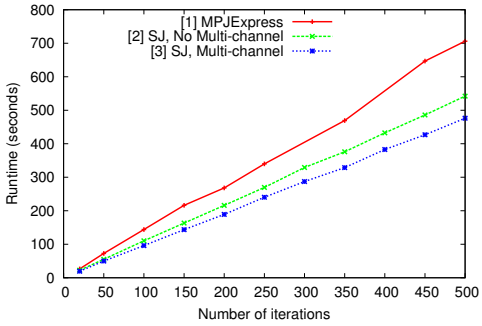
The main objective of these benchmarks is to investigate the benefits of the new multi-channel primitives. Consequently, we benchmarked implementations of the *n*-Body algorithm both with and without the new primitives (Figure 13a). We also evaluated a hybrid implementation using the FPGA for numerical computation, and the CPU to coordinate communication and control the FPGA. We compared our *n*-Body implementations to another using MPJ Express for reference.



(a) 350 iterations



(b) 20 iterations



(c) Varying iterations

Figure (a) shows execution times for our *n*-Body implementations, for increasing numbers of particles (bodies) in the *n*-Body simulation, and with 350 inner iterations of the algorithm, where the original SJ version had to keep reopening a temporary session. Both SJ implementations are consistently faster than MPJ Express, up to 44% for multi-channel SJ with 44000 particles. We observe a clear improvement when using the new multi-channel primitives in the SJ implementations.

Figure (b) uses a similar setup, but this time with 20 inner iterations. This is used to show the gains from FPGA acceleration. The FPGA-accelerated SJ implementation suffers from high overhead at lower particle counts, but becomes the fastest after 11500 particles and scales much better with larger data volumes.

We ran another series of measurements, this time keeping the number of particles constant, but varying the number of inner iterations. The results (c) demonstrate the growing impact of the temporary session as we add inner iterations. In percentage terms though, the gains from multi-channel primitives are quite stable, with an average of 12% speed gain (standard deviation: 1.1%). Similarly, the multi-channel SJ implementation is on average 31% faster than MPJ (standard deviation: 2.6%).

We believe we have demonstrated the benefits of SJ's new multi-channel primitives for parallel programming. SJ has become a viable alternative to MPI programming in Java, consistently outperforming MPJ Express in this benchmark.

6 Related and Future Work

Implementations of session types. SJ was introduced in [21] as the first general-purpose session-typed programming language. This work investigates SJ and session types for parallel programming, and introduces new multi-channel primitives, allowing programmers to efficiently express common and complex communication topologies. Another recent extension of SJ added session-typed primitives for event-based programming [20]. These directions differ in their target application domain, and have contributed complementary developments to session programming. The preliminary experiments with parallel algorithms in SJ leading to the present work were reported in a workshop paper [2]. This early work considered only simple iteration chaining, and not 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 type soundness proofs for the new primitives; and comprehensive benchmark results from high performance clusters and an advanced FPGA implementation.

Our current version of SJ implements binary session types, with full support for multi-channel communications, multiple interleaved sessions, delegation, named recursion, and n-ary branching. Other implementations have been presented for Haskell [26, 29, 30]. These are all library-based implementations, offering only limited features of binary session types. The Bica language [14] 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 progress across multiple sessions. Session typecase [20] is an alternative mechanism for supporting sessions as object fields without requiring an additional typing layer controlling the order of method calls such as that of [14].

Other implementations support multiparty session types (MPST). Early work [8] applies them to security concerns, which includes a type-checker and code generator for ML. A recent implementation extends SJ for MPST [32] and studies type-directed optimisations for the extended language. The implemented multiparty sessions are less expressive than that of the formal model originally presented in [18], as sessions cannot be interleaved in the same process. Other session-based optimisations applied to buffered communication are presented in [9] but have yet to be implemented.

Message-based parallel programming. MPI [24] is one of the most widely-used APIs for parallel programming using message passing. Implementations supply concrete language and transport bindings, such as C, C++, Fortran, and Java over one or more specific transports. This work focuses on language and typing support for communications programming, rather than introducing a supplementary API. In comparison to the standard MPI libraries [15, §4], SJ offers productivity gains coming from natural abstraction of communication actions by typed sessions and the associated static assurance of type and protocol safety. Recent work [36] 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. Some of the code verifications took over two days. 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. Appendix B). The MPI API remains low-level, easily leading to synchronisation errors, message

type errors and deadlocks [15]. From our experiences, we found programming these and other message-based parallel algorithms with SJ through typed sessions based on Java much easier than programming based on the basic 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. Additionally, SJ integrates objects and sessions to support e.g. Java objects as high-level messages types through its remote class loading facility [21]. SJ leverages session types to overcome several MPI problems: our type-based approach enables structured communications programming, and yields a clear definition of a class of communication-safe and deadlock-free programs as proved in Theorems 4.2 and 4.3: if multiple SJ parallel programs conform to a well-formed topology (Definition A.1), their type and communication have been checked to be safe without having to explore all execution states for all possible thread interleavings. Finally, benchmark results in §5 demonstrate how SJ programs can deliver the above benefits and still outperform a Java-based MPI implementation [25].

OpenMP and PGAS languages. OpenMP [28] is a combination of pragma-based program transformation and libraries for extracting latent parallelism (shared memory multithreading) from sequential code. X10 [7, 23, 37], Chapel [6] and Fortress [13] are recent PGAS (Partitioned Global Address Space) languages for HPC that share a notion of partitioned globally accessible memory locations, with different access semantics for local and remote partitions. The model applies both to non-uniform memory hierarchies within a node, and to distributed memory locations on clusters. These languages focus on reducing programming complexity for shared memory parallelism through a range of annotations and high-level constructs for coordinating and synchronising thread behaviours. Even though PGAS languages offer a convenient model and primitives for parallel programming, they lack the static safety offered by session programming. Their programming model is also fundamentally different from that of SJ: rather than seeking to hide communication under a distributed shared memory abstraction, SJ promotes the encapsulation of an explicitly structured series of message exchanges as a session for increased modularity.

Aliasing control. The noalias modifier in SJ is similar to the `unique` annotation of [1] for unshared references. SJ demonstrates how the integration of object alias control and message passing programming can have significant performance benefits whilst retaining semantic transparency. SJ differs from [1] by directly capturing the semantics of noalias assignment and argument passing operationally – setting a consumed reference to `null`, rather than enforcing particular patterns for variable usage. This design choice allows SJ, in contrast to [1], to dispense with manual synchronisation for field accesses and leads to a different inference of noalias-compatible classes.

Kilim [34] is an actor framework for Java. Messages sent by actors are statically guaranteed to have at most one owner at any time, allowing efficient zero-copy transfer. However, message classes in Kilim must be made specifically for message-passing, and cannot be regular Java API classes. Other earlier works on unique references are surveyed in [16].

StreamFlex [33] is a real-time streaming API for Java. It makes use of a type-based classification of heap objects ensuring single ownership of messages, and leading to a high throughput. The typing disciplines in these two works only support ownership tracking for specific message objects, and will not track ownership for general objects.

Both SJ with the `noalias` keyword, and an extension of Scala [16] support aliasing control for standard, unchanged classes. Annotations are only required at the point of use. The SJ `noalias` approach is more restricted than that of [16], as it does not allow local aliases. In SJ, we chose the simplest possible approach that would cover both session type checking needs, and efficient message passing. The capability-based approach of [16] concentrates on zero-copy message passing, and can thus allow the extra flexibility of local aliases. Further comparisons between SJ's `noalias` modifier and other works can be found in [19, § 3.11].

Future work. Although we believe our simple, novel method for guaranteeing deadlock-freedom is very effective in practice, we also plan to implement multiparty session types for increased expressiveness. Preliminary results from a manual SJ-to-C translation [27] have shown large performance gains. Future implementation efforts will include a natively compiled, C-like language focused on low overheads and efficiency for HPC and systems programming. We also plan to incorporate recent, unimplemented theoretical advances, including the fine-grained failure handling of [4], parameterised multiparty sessions types [38] for more flexible topologies and integration with logical reasoning [3] in order to prove the correctness of parallel algorithms.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.
2. A. Bejleri, R. Hu, and N. Yoshida. Session-Based Programming for Parallel Algorithms. In A. R. Beresford and S. Gay, editors, *PLACES*, EPTCS, 2009.
3. 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.
4. S. Capecchi, E. Giachino, and N. Yoshida. Global Escape in Multiparty Sessions. In *FSTTCS '10*, 2010. To appear.
5. H. Casanova, A. Legrand, and Y. Robert. *Parallel Algorithms*. Chapman & Hall, 2008.
6. Chapel homepage. `http://chapel.cs.washington.edu/`.
7. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05*, pages 519–538. ACM, 2005.
8. R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. A Secure Compiler for Session Abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.
9. P.-M. Deniélou and N. Yoshida. Buffered Communication Analysis in Distributed Multiparty Sessions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010.
10. M. Dezani-Ciancaglini, U. de' Liguoro, and N. Yoshida. On Progress for Structured Communications. In *TGC '07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.
11. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP '06*, volume 4067 of *LNCS*, pages 328–352, 2006.
12. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. *SIGOPS Operating Systems Review*, 40(4):177–190, 2006.
13. Fortress homepage. `http://projectfortress.sun.com/Projects/Community`.
14. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular Session Types for Distributed Object-Oriented Programming. In M. V. Hermenegildo and J. Palsberg, editors, *POPL '10*, volume 45, pages 299–312. ACM, 2010.
15. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

16. P. Haller and M. Odersky. Capabilities for Uniqueness and Borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer, 2010.
17. 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.
18. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008.
19. R. Hu. *Structured, Safe and High-level Communications Programming with Session Types*. PhD thesis, Imperial College London, 2010.
20. 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.
21. 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.
22. Y. Kryftis. *Session-based Programming for Message-Passing-based Parallel Algorithms*. Master's thesis, Imperial College London, 2009. `http://www.doc.ic.ac.uk/~yk208/`.
23. J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In R. Govindarajan, D. A. Padua, and M. W. Hall, editors, *PPoPP*, pages 25–36. ACM, 2010.
24. Message Passing Interface. `http://www.mcs.anl.gov/research/projects/mpi/`.
25. MPJ Express homepage. `http://mpj-express.org/`.
26. M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL '04*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
27. N. Ng. High Performance Parallel Design based on Session Programming. MEng thesis, Department of Computing, Imperial College London, 2010. `http://www.doc.ic.ac.uk/~cn06/individual-project/`.
28. OpenMP homepage. `http://openmp.org/`.
29. R. Pucella and J. A. Tov. Haskell Session Types with (Almost) No Class. In *Haskell '08*, pages 25–36. ACM, 2008.
30. M. Sackman and S. Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, Imperial College London, July 2008.
31. 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.
32. K. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient Session Type Guided Distributed Interaction. In D. Clarke and G. Agha, editors, *Coordination Models and Languages*, volume 6116 of *LNCS*, pages 152–167. Springer, 2010.
33. 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.
34. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP '08*, volume 5142 of *LNCS*, pages 104–128. Springer, 2008.
35. K. H. Tsoi and W. Luk. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In *FPGA '10*, pages 115–124. ACM, 2010.
36. 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.
37. X10 homepage. `http://x10.sf.net`.
38. 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.
39. 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.

A Appendix to Section 4

This section lists the omitted definitions from Section 4. Structural congruence rules are defined in Figure 13; full typing rules can be found in Figure 14. In this context, $\mathrm{fn}(Q)$ denotes a set of free shared and session channels, and $\mathrm{fpv}(D)$ stands for a set of free process variables. In the typing system, Δ is complete means that Δ includes only $\mathtt{end}$ or $\bot$.

$$P \equiv Q \text{ if } P \equiv_\alpha Q \quad P \mid \mathbf{0} \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu u)P \mid Q \equiv (\nu u)(P \mid Q) \quad \text{if } u \notin \mathrm{fn}(Q)$$
$$(\nu u)\,\mathbf{0} \equiv \mathbf{0} \qquad \mathtt{def}\ D\ \mathtt{in}\ \mathbf{0} \equiv \mathbf{0} \qquad \mathbf{0}\,;\, P \equiv P$$
$$(\nu u)\mathtt{def}\ D\ \mathtt{in}\ P \equiv \mathtt{def}\ D\ \mathtt{in}\ (\nu u)P \quad \text{if } u \notin \mathrm{fn}(D)$$
$$(\mathtt{def}\ D\ \mathtt{in}\ P) \mid Q \equiv \mathtt{def}\ D\ \mathtt{in}\ (P \mid Q) \quad \text{if } \mathrm{fpv}(D) \cap \mathrm{fpv}(Q) = \emptyset$$
$$\mathtt{def}\ D\ \mathtt{in}\ (\mathtt{def}\ D'\ \mathtt{in}\ P) \equiv \mathtt{def}\ D\ \mathtt{and}\ D'\ \mathtt{in}\ P \quad \text{if } \mathrm{fpv}(D) \cap \mathrm{fpv}(D') = \emptyset.$$

Fig. 13: Structural congruence.

A.1 Well-Formed Ring and Mesh Topologies

We define well-formed ring and mesh topologies. We can check that they conform to the general definition of well-formed topology (Definition 4.1). Figure 15 shows the rank of each process for each topology, indicating how both rings and meshes map to the general definition.

Definition A.1. A process conforms to a *well-formed ring topology* if:

$$\begin{aligned}
P_1 &= \langle k_{1,2}, k_{1,n} \rangle.\mathtt{outwhile}(e)\{Q_1[k_{1,2}, k_{1,n}]\} \\
P_{i \in \{2..n-1\}} &= k_{i,i+1}.\mathtt{outwhile}(\langle k_{i-1,i} \rangle.\mathtt{inwhile})\{Q_i[k_{i,i+1}, k_{i-1,i}]\}\ 2 \le i \le n-1 \\
P_n &= \langle k_{1,n}, k_{n-1,n} \rangle.\mathtt{inwhile}\{Q_n[k_{1,n}, k_{n-1,n}]\} \\
&\textit{and} \\
\Gamma &\vdash Q_1 \triangleright \{k_{1,2} \colon T_{1,2},\ k_{1,n} \colon T_{1,n}\} \\
\Gamma &\vdash Q_i \triangleright \{k_{i,i+1} \colon T_{i,i+1},\ k_{i-1,i} \colon T'_{i-1,i}\} \\
\Gamma &\vdash Q_n \triangleright \{k_{1,n} \colon T_{1,n}{}',\ k_{n-1,n} \colon T_{n-1,n}{}'\} \\
&\textit{and} \\
\Gamma &\vdash Q_1 \mid Q_2 \mid \ldots \mid Q_n \triangleright \{\tilde{k} \colon \tilde{\bot}\} \\
&\textit{with } \overline{T_{i,j}} = T'_{i,j}
\end{aligned}$$

Definition A.2. *A process group*

$$\begin{aligned}
&P_{NW} \mid P_{NE} \mid P_{SW} \mid P_{SE} \mid P_{N_1} \ldots \mid P_{N_m} \mid P_{S_1} \ldots \mid P_{S_m} \\
&\quad \mid P_{E_1} \ldots \mid P_{E_n} \mid P_{W_1} \ldots \mid P_{W_n} \mid P_{C_2 2} \ldots \mid P_{C_{n-1 m-1}}
\end{aligned}$$

$$\frac{}{\Gamma \vdash 1 \triangleright \mathsf{nat}} \qquad \frac{}{\Gamma \vdash \mathtt{true}, \mathtt{false} \triangleright \mathsf{bool}} \qquad \frac{\Gamma \vdash e_i \triangleright \mathsf{nat}}{\Gamma \vdash e_1 + e_2 \triangleright \mathsf{nat}} \qquad \text{[Nat],[Bool],[Sum]}$$

$$\frac{}{\Gamma \cdot a\colon S \vdash a \triangleright S} \qquad \frac{\Gamma;\ \Delta \vdash e \triangleright S}{\Gamma;\ \Delta, \Delta' \vdash e \triangleright S} \qquad \text{[Name],[Eval]}$$

$$\frac{\Delta = k_1\colon ?[\tau_1]^*.\mathsf{end}, ..., k_n\colon ?[\tau_n]^*.\mathsf{end}}{\Gamma; \Delta \vdash \langle k_1 \ldots k_n \rangle.\mathtt{inwhile} \triangleright \mathsf{bool}} \qquad \text{[EInwhile]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k\colon \varepsilon.\mathsf{end}}{\Gamma \vdash P \triangleright \bot} \qquad \frac{\Delta \text{ complete}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \text{[Bot],[Inact]}$$

$$\frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot k\colon \overline{\alpha}}{\Gamma \vdash \overline{a}(k).P \triangleright \Delta} \qquad \frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot k\colon \alpha}{\Gamma \vdash a(k).P \triangleright \Delta} \qquad \text{[Req],[Acc]}$$

$$\frac{\Gamma \vdash e \triangleright S}{\Gamma \vdash \overline{k}\langle e \rangle \triangleright \Delta \cdot k\colon\ !{[S]}.\mathsf{end}} \qquad \frac{\Gamma \cdot x\colon S \vdash P \triangleright \Delta \cdot k\colon \alpha}{\Gamma \vdash k(x).P \triangleright \Delta \cdot k\colon ?[S];\ \alpha} \qquad \text{[Send],[Rcv]}$$

$$\frac{\Gamma \vdash P_1 \triangleright \Delta \cdot k\colon \tau_1.\mathsf{end} \quad \cdots \quad \Gamma \vdash P_n \triangleright \Delta \cdot k\colon \tau_n.\mathsf{end}}{\Gamma \vdash\ k \triangleright \{l_1 : P_1 [] \cdots [] l_n : P_n\} \triangleright \Delta \cdot k\colon \&\{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}.\mathsf{end}} \qquad \text{[Br]}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k\colon \tau_j.\mathsf{end} \quad 1 \leq j \leq n}{\Gamma \vdash k \lhd l \triangleright \Delta \cdot k\colon\ \oplus\{l_1\colon \tau_1, \ldots, l_n\colon \tau_n\}.\mathsf{end}} \qquad \text{[Sel]}$$

$$\frac{}{\Gamma \vdash k\langle k' \rangle \triangleright \Delta \cdot k\colon\ !{[\alpha]}.\mathsf{end} \cdot k'\colon \alpha} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k\colon \beta \cdot k'\colon \alpha}{\Gamma \vdash k(k').P \triangleright \Delta \cdot k\colon ?[\alpha];\ \beta} \qquad \text{[Thr],[Cat]}$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta} \qquad \text{[If]}$$

$$\frac{\Gamma;\ \Delta \vdash e \triangleright \mathsf{bool} \qquad \Gamma \vdash P \triangleright \Delta \cdot k_1\colon \tau_1.\mathsf{end} \cdot \cdots \cdot k_n\colon \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots\ k_n \rangle.\mathtt{outwhile}(e)\{P\} \triangleright \Delta \cdot k_1\colon\ !{[\tau_1]}^*.\mathsf{end} \cdot \cdots \cdot k_n\colon\ !{[\tau_n]}^*.\mathsf{end}} \qquad \text{[Outwhile]}$$

$$\frac{\Gamma \vdash Q \triangleright \Delta \cdot k_1\colon \tau_1.\mathsf{end} \cdot \cdots \cdot k_n\colon \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \ldots\ k_n \rangle.\mathtt{inwhile}\{Q\} \triangleright \Delta \cdot k_1\colon\ ?[\tau_1]^*.\mathsf{end} \cdot \cdots \cdot k_n\colon\ ?[\tau_n]^*.\mathsf{end}} \qquad \text{[Inwhile]}$$

$$\frac{\Gamma \vdash b_i \triangleright \mathsf{bool}}{\Gamma \vdash \Pi_{i \in \{1..n\}} k_i \dagger [b_i] \triangleright k_1\colon \dagger, ..., k_n\colon \dagger} \qquad \frac{\Gamma \cdot a\colon S \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k\colon\ \bot}{\Gamma \vdash\ (\nu k)P \triangleright \Delta}$$

[Message],[NRes],[CRes]

$$\frac{\Gamma;\ \emptyset \vdash e \triangleright S}{\Gamma \cdot X : S\alpha \vdash X[ek] \triangleright \Delta \cdot k\colon \alpha} \qquad \frac{\Gamma \cdot X : S\alpha \cdot x\colon S \vdash P \triangleright k\colon \alpha \qquad \Gamma \cdot X : S\tau \vdash Q \triangleright \Delta}{\Gamma \vdash \mathtt{def}\ X(xk) = P\ \mathtt{in}\ Q \triangleright \Delta} \qquad \text{[Var],[Def]}$$

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

Fig. 14: Typing rules.

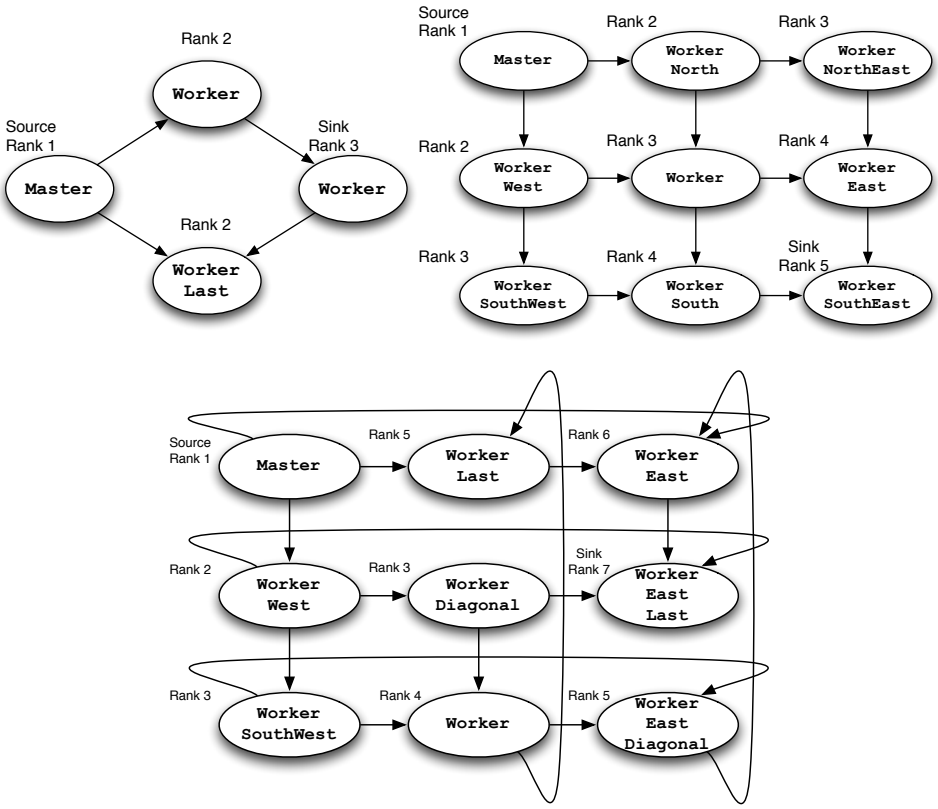


Fig. 15: Ring, mesh, and wraparound mesh topologies, with rank annotations.

conforms to a well-formed mesh topology *if:*

$$
\begin{aligned}
P_{NW} =& \langle t_1, l_1 \rangle . \texttt{outwhile}(e)\{Q_{NW}[t_1, l_1]\} \\
P_{N_j} =& \langle t_{j+1}, vc_{1j} \rangle . \texttt{outwhile}(t_j . \texttt{inwhile})\{Q_{N_j}[t_{j+1}, vc_{1j}, t_j]\} \\
P_{NE} =& r_1 . \texttt{outwhile}(t_m . \texttt{inwhile})\{Q_{NE}[r_1, t_m]\} \\
P_{W_i} =& \langle hc_{i1}, l_{i+1} \rangle . \texttt{outwhile}(l_i . \texttt{inwhile})\{Q_W[hc_{i1}, l_{i+1}, l_i]\} \\
P_{C_{ij}} =& \langle vc_{i+1\ j}, hc_{i\ j+1} \rangle . \texttt{outwhile}(\langle hc_{ij}, vc_{ij} \rangle . \texttt{inwhile})\{ \\
& \qquad Q_{C_{ij}}[vc_{i+1\ j}, hc_{i\ j+1}, hc_{ij}, vc_{ij}]\} \\
P_{E_i} =& r_{i+1} . \texttt{outwhile}(\langle hc_{im}, r_i \rangle . \texttt{inwhile})\{ \\
& \qquad Q_{E_i}[r_{i+1}, hc_{im}, r_i]\} \\
P_{SW} =& b_1 . \texttt{outwhile}(l_n . \texttt{inwhile})\{Q_{SW}[b_1, l_n]\} \\
P_{S_j} =& b_{j+1} . \texttt{outwhile}(\langle b_j, vc_{nj} \rangle . \texttt{inwhile})\{ \\
& \qquad Q_{S_j}[b_{j+1}, b_j, vc_{nj}]\} \\
P_{SE} =& \langle b_m, r_n \rangle . \texttt{inwhile}\{Q_{SE}[b_m, r_n]\}
\end{aligned}
$$

where $1 \leq i \leq n, 1 \leq j \leq m$, *and*

$$\begin{array}{ll}
& \Gamma \vdash Q_{NW} \triangleright \{t_1 : T_{t_1}, l_1 : T_{l_1}\} \\
& \Gamma \vdash Q_{N_j} \triangleright \{t_{j+1} : T_{t_{j+1}}, vc_{1j} : T_{vc_{1j}}, t_j : T'_{t_j}\} \\
& \Gamma \vdash Q_{NE} \triangleright \{r_1 : T_{r_1}, t_m : T'_{t_m}\} \\
& \Gamma \vdash Q_W \triangleright \{hc_{i1} : T_{hc_{i1}}, l_{i+1} : T_{l_{i+1}}, l_i : T'_{l_i}\} \\
& \Gamma \vdash Q_{C_{ij}} \triangleright \{vc_{i+1\ j} : T_{vc_{i+1\ j}}, hc_{i\ j+1} : T_{hc_{i\ j+1}}, hc_{ij} : T'_{hc_{ij}}, vc_{ij} : T'_{vc_{ij}}\} \\
& \Gamma \vdash Q_{E_i} \triangleright \{r_{i+1} : T_{r_{i+1}}, hc_{im} : T_{hc_{im}}, r_i : T'_{r_i}\} \\
& \Gamma \vdash Q_{SW} \triangleright \{b_1 : T_{b_1}, l_n : T'_{l_n}\} \\
& \Gamma \vdash Q_{S_j} \triangleright \{b_{j+1} : T_{b_{j+1}}, b_j : T'_{b_j}, vc_{nj} : T'_{vc_{nj}}\} \\
& \Gamma \vdash Q_{SE} \triangleright \{b_m : T'_{b_m}, r_n : T'_{r_n}\} \\
\textit{with} \quad & \overline{T_i} = T'_i
\end{array}$$

B Appendix to Section 3

We present here an additional parallel algorithm implementation in SJ, which further demonstrates the benefits of our multi-channel primitives.

B.1 Linear Equation Solver: Wraparound Mesh Topology

Linear equations are at the core of many engineering problems. Solving a system of linear equations consists in finding x such that $Ax = b$, where A is an $n \times n$ matrix and x and b are vectors of length n.

A whole range of methods for solving linear systems are available. One of the most amenable to parallelization is the Jacobi method. It is based on the observation that the matrix A can be decomposed into a diagonal component and a remainder: $A = D + R$. The equation $Ax = b$ is then equivalent to $x = D^{-1}(b - Rx)$, again equivalent to finding the solution to the n equations $\sum_{j=1}^{n} \alpha_{ij} x_j = b_i$ for $i = 1, ..., n$. Solving the i-th equation for x_i yields: $x_i = \frac{1}{\alpha_{ii}}(b_i - \sum_{j \neq i} \alpha_{ij} x_j)$, which suggests the iterative method: $x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} \alpha_{ij} x_j^{(k)})$, where $k \geq 0$ and $x^{(0)}$ is an initial guess at the solution vector. The algorithm iterates until the normalized difference between successive iterations is less than some predefined error.

Our parallel implementation of this algorithm uses p^2 processors in a $p \times p$ wraparound mesh topology to solve an $n \times n$ system matrix. The matrix is partitioned into submatrix blocks of size $\frac{n}{p} \times \frac{n}{p}$, assigned to each of the processors (see Figure 16).

Each iteration of the algorithm requires multiplications (in the term $\alpha_{ij} x_j$) and summation. Multiplications dominate execution time here, hence the parallelization concentrates on them. The horizontal part of the mesh acts as a collection of circular pipelines for multiplications. Their results are collected by the diagonal nodes, which perform the summation and the division by α_{ii}.

This gives the updated solution values for the iteration. These need to be communicated to other nodes for the next iteration. The vertical mesh connections are used for this purpose: the solution values are sent down by the diagonal node, and each worker node picks up the locally required solution values, and passes on the rest. The transmis-

sion wraps around at the bottom of the mesh, and stops at the node immediately above the diagonal, hence the lack of connectivity between the two in Figure 16.

Master (is also on the diagonal):
```
<under,right>.outwhile(
    hasNotConverged()) {
  prod = computeProducts();
  // horizontal ring, pass results
       to diagonal node
  ringData = prod;
  <left,right>.outwhile(count <
      nodesOnRow) {
    right.send(ringData);
    ringData = left.receive();
    computeSums(ringData);
    count++;
  }
  newX = computeDivision();
  under.send(newX);
}
```

Worker :
```
<under,right>.outwhile(<left,over
    >.inwhile) {
  prod = computeProducts();
  ringData = prod;
  right.outwhile(left.inwhile) {
    right.send(ringData);
    ringData = left.receive();
  }
  newX = over.receive();
  under.send(newX);
}
```

WorkerDiagonal :
```
<under,right>.outwhile(left.
    inwhile) {
  prod = computeProducts();
  ringData = prod;
  right.outwhile(left.inwhile) {
    right.send(ringData);
    ringData = left.receive();
    computeSums(ringData);
  }
  newX = computeDivision();
  under.send(newX);
}
```

WorkerEast :
```
under.outwhile(<right,left,over>.
    inwhile) {
  prod = computeProducts();
  ringData = prod;
  <left,right>.inwhile {
    right.send(ringData);
    ringData = left.receive();
  }
  newX = over.receive();
  under.send(newX);
}
```

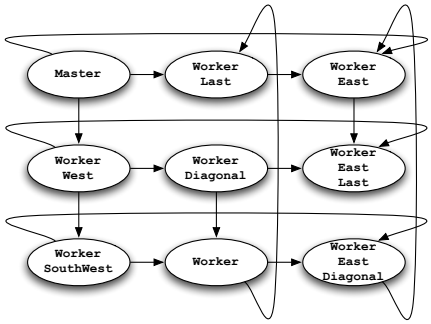


Fig. 16: Linear Equations Solver implementation using a wraparound mesh.

Note that contrary to the non-wraparound 2D-mesh of § 3.2, the sink of this well-formed topology (§ 4.1) is not the last node on the diagonal, but instead the node just above, called `WorkerEastLast`. This is because the diagonal nodes transmit updated values as explained above, and this transmission stops just before a complete wraparound.

Figure 15 shows node ranks for the wraparound mesh topology, along with the other topologies presented in the paper.

C Appendix - Proofs

Definition C.1. *Sequential composition of session type are defined as [10]:*

$$\tau;\ \alpha = \begin{cases} \tau.\alpha & \text{if } \tau \text{ is a partial session type and } \alpha \text{ is a completed session type} \\ \bot & \text{otherwise} \end{cases}$$

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

The first rule concatenates a partial session type τ with a completed session type α to form a new (completed) session type. The second rule can be decomposed to three parts:

1. $\Delta \setminus \mathrm{dom}(\Delta')$ extracts session types with sessions unique in Δ
2. $\Delta' \setminus \mathrm{dom}(\Delta)$ extracts session types with sessions unique in Δ'
3. $\{k\colon \Delta(k) \setminus \mathsf{end};\ \Delta'(k) \mid k \in \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta')\}$ modifies session types with a common session k in Δ and Δ' by removing end type from $\Delta(k)$ and concatenates the modified $\Delta(k)$ (which is now a partial session type) with $\Delta'(k)$ as described in the first rule.

Example C.1. Suppose $\Delta = \{k_1\colon \varepsilon.\mathsf{end},\ k_2\colon !{[\mathsf{nat}]}.\mathsf{end}\}$ and $\Delta' = \{k_2\colon ?{[\mathsf{bool}]}.\mathsf{end},\ k_3\colon !{[\mathsf{bool}]}.\mathsf{end}\}$. Since k_1 is unique in Δ and k_3 is unique in Δ', we have

$$\Delta \setminus \mathrm{dom}(\Delta') = \{k_1\colon \varepsilon.\mathsf{end}\} \text{ and } \Delta' \setminus \mathrm{dom}(\Delta) = \{k_3\colon !{[\mathsf{bool}]}.\mathsf{end}\}$$

A new session type is constructed by removing end in $\Delta(k_2)$, so the composed set of mappings is

$$\Delta;\ \Delta' = \{k_1\colon \varepsilon.\mathsf{end},\ k_2\colon ![\mathsf{nat}];\ ?{[\mathsf{bool}]}.\mathsf{end},\ k_3\colon !{[\mathsf{bool}]}.\mathsf{end}\}$$

Definition C.2. *Parallel composition of session and runtime type is defined as:*

$$\Delta \circ \Delta' = \Delta \setminus \mathrm{dom}(\Delta') \cup \Delta' \setminus \mathrm{dom}(\Delta) \cup \{k\colon \beta \circ \beta' \mid \Delta(k) = \beta \text{ and } \Delta'(k) = \beta'\}$$

$$\text{where } \beta \circ \beta' : \begin{cases} \alpha \circ \dagger = \alpha^{\dagger} \\ \alpha \circ \overline{\alpha} = \bot \\ \alpha \circ \overline{\alpha}^{\dagger} = \bot^{\dagger} \end{cases}$$

The parallel composition relation $\circ$ is commutative as the order of composition do not impact the end result.

We now present some auxilary results for subject reduction, the following proofs are modified from [39], and adapted to our updated typing system.

Lemma C.1 (Weakening Lemma). *Let $\Gamma \vdash P \triangleright \Delta$.*

1. *If $X \notin \mathrm{dom}(\Gamma)$, then $\Gamma \cdot X : S\alpha \vdash P \triangleright \Delta$.*
2. *If $a \notin \mathrm{dom}(\Gamma)$, then $\Gamma \cdot a : S \vdash P \triangleright \Delta$.*

3. *If* $k \notin \mathrm{dom}(\Delta)$ *and* $\alpha = \bot$ *or* $\alpha = \varepsilon.\mathsf{end}$, *then* $\Gamma \vdash P \triangleright \Delta \cdot k : \alpha$.

Proof. A simple induction on the derivation tree of each sequent. For 3, we note that in [INACT] and [VAR], Δ contains only $\varepsilon.\mathsf{end}$.

Lemma C.2 (Strengthening Lemma). *Let* $\Gamma \vdash P \triangleright \Delta$.

1. *If* $X \notin \mathrm{fpv}(P)$, *then* $\Gamma \setminus X \vdash P \triangleright \Delta$.
2. *If* $a \notin \mathrm{fn}(P)$, *then* $\Gamma \setminus a \vdash P \triangleright \Delta$.
3. *If* $k \notin \mathrm{fn}(P)$, *then* $\Gamma \vdash P \triangleright \Delta \setminus k$.

Proof. Standard.

Lemma C.3 (Channel Lemma).

1. *If* $\Gamma \vdash P \triangleright \Delta \cdot k : \alpha$ *and* $k \notin \mathrm{fn}(P)$, *then* $\alpha = \bot, \varepsilon.\mathsf{end}$.
2. *If* $\Gamma \vdash P \triangleright \Delta$ *and* $k \in \mathrm{fn}(P)$, *then* $k \in \mathrm{dom}(\Delta)$.

Proof. A simple induction on the derivation tree for each sequent.

We omit the standard renaming properties of variables and channels, but present the Substitution Lemma for names. Note that we do *not* require a substitution lemma for channels or process variables, for they are not communicated.

Lemma C.4 (Substitution Lemma). *If* $\Gamma \cdot x : S \vdash P \triangleright \Delta$ *and* $\Gamma \vdash c : S$, *then* $\Gamma \vdash P\{c/x\} \triangleright \Delta$

Proof. Standard.

We write $\Delta \prec \Delta'$ if we obtain Δ' from Δ by replacing $k_1 : \varepsilon.\mathsf{end}, \ldots, k_n : \varepsilon.\mathsf{end}$ $(n \geq 0)$ in Δ by $k_1 : \bot, \ldots, k_n : \bot$. If $\Delta \prec \Delta'$, we can obtain Δ' from Δ by applying the [BOT]-rule zero or more times.

Theorem C.1. *Subject congruence is defined by*

$$\Gamma \vdash P \triangleright \Delta \text{ and } P \equiv P' \text{ implies } \Gamma \vdash P' \triangleright \Delta$$

Proof. **Case** $P \mid \mathbf{0} \equiv P$**.** We show that if $\Gamma \vdash P \mid \mathbf{0} \triangleright \Delta$, then $\Gamma \vdash P \triangleright \Delta$. Suppose

$$\Gamma \vdash P \triangleright \Delta_1 \quad \text{and} \quad \Gamma \vdash \mathbf{0} \triangleright \Delta_2.$$

with $\Delta_1 \circ \Delta_2 = \Delta$. Note that Δ_2 only contains $\varepsilon.\mathsf{end}$ or $\bot$, hence we can set: $\Delta_1 = \Delta_1' \circ \{k : \varepsilon.\mathsf{end}\}$ and $\Delta_2 = \Delta_2' \cdot \{k : \varepsilon.\mathsf{end}\}$ with $\Delta_1' \circ \Delta_2' = \Delta_1' \cdot \Delta_2'$ and $\Delta = \Delta_1' \cdot \Delta_2' \cdot \{k : \bot\}$. Then by the [BOT]-rule, we have:

$$\Gamma \vdash P \triangleright \Delta_1' \cdot \{k : \bot\}$$

Notice that, given the form of Δ above, we know that $\mathrm{dom}(\Delta_2') \cap \mathrm{dom}(\Delta_1') \cdot \{k : \bot\}) = \emptyset$. Hence by applying Weakening, we have:

$$\Gamma \vdash P \triangleright \Delta_1' \cdot \Delta_2' \cdot \{k : \bot\}$$

as required.
For the other direction, we set $\Delta = \emptyset$ in [INACT].

Case $P \mid Q \equiv Q \mid P$**.** $\circ$ relation is commutative by the definition of $\circ$ (Definition C.2)
Case $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$**.** To show $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, where

$$\Gamma \vdash P \triangleright \Delta_1 \quad \Gamma \vdash Q \triangleright \Delta_2 \quad \Gamma \vdash R \triangleright \Delta_3$$

We assume $(\Delta_1 \circ \Delta_2) \circ \Delta_3$ is defined
Suppose $k\colon \beta_1 \in \Delta_1$ and $k\colon \beta_2 \in \Delta_2$, then we have

$$\begin{cases} \beta_1 = \alpha \;\; \beta_2 = \dagger \\ \beta_1 = \alpha \;\; \beta_2 = \overline{\alpha} \\ \beta_1 = \alpha \; \beta_2 = \overline{\alpha}^{\dagger} \\ \beta_1 = \dagger \;\; \beta_2 = \bot \end{cases}$$

Now suppose $k\colon \beta_3 \in \Delta_3$,
if $\beta_1 = \alpha \quad \beta_2 = \dagger$, then $\beta_3 = \overline{\alpha}$

$$\begin{aligned} (\beta_1 \circ \beta_2) \circ \beta_3 &= (\{k\colon \alpha\} \circ \{k\colon \dagger\}) \circ \{k\colon \overline{\alpha}\} = \{k\colon \bot^{\dagger}\} \\ \equiv \beta_1 \circ (\beta_2 \circ \beta_3) &= \{k\colon \alpha\} \circ (\{k\colon \dagger\} \circ \{k\colon \overline{\alpha}\}) = \{k\colon \bot^{\dagger}\} \end{aligned}$$

if $\beta_1 = \alpha \quad \beta_2 = \overline{\alpha}$, then $\beta_3 = \dagger$

$$\begin{aligned} (\beta_1 \circ \beta_2) \circ \beta_3 &= (\{k\colon \alpha\} \circ \{k\colon \overline{\alpha}\}) \circ \{k\colon \dagger\} = \{k\colon \bot^{\dagger}\} \\ \equiv \beta_1 \circ (\beta_2 \circ \beta_3) &= \{k\colon \alpha\} \circ (\{k\colon \overline{\alpha}\} \circ \{k\colon \dagger\}) = \{k\colon \bot^{\dagger}\} \end{aligned}$$

in all other cases, $k \notin \mathrm{dom}(\Delta_3)$ and therefore no parallel composition is possible.
Case $(\nu u)P \mid Q \equiv (\nu u)(P \mid Q)$ if $u \notin \mathrm{fn}(Q)$**.** The case when u is a name is standard. Suppose u is channel k and assume $\Gamma \vdash (\nu k)(P \mid Q) \triangleright \Delta$. We have

$$\frac{\Gamma \vdash P \triangleright \Delta_1' \qquad \Gamma \vdash Q \triangleright \Delta_2'}{\Gamma \vdash P \mid Q \triangleright \Delta' \cdot k\colon \bot}$$

with $\Delta' \cdot k\colon \bot = \Delta_1' \circ \Delta_2'$ and $\Delta' \prec \Delta$ by [Bot]. First notice that k can be in either Δ_i' or in both. The interesting case is when it occurs in both; from Lemma C.3(1) and the fact that $k \notin \mathrm{fn}(Q)$ we know that $\Delta_1' = \Delta_1 \cdot k\colon \varepsilon.\mathsf{end}$ and $\Delta_2' = \Delta_2 \cdot k\colon \varepsilon.\mathsf{end}$. Then, by applying the [Bot]-rule to k in P, we have $\Gamma \vdash P \triangleright \Delta_1 \cdot k\colon \bot$, and by applying [CRes] we obtain $\Gamma \vdash (\nu k)P \triangleright \Delta_1$. On the other hand, by Strengthening, we have $\Gamma \vdash Q \triangleright \Delta_2$. Then, the application of [Conc] yields $\Gamma \vdash (\nu k)P \mid Q \triangleright \Delta'$. Then by applying the [Bot]-rule, we obtain $\Gamma \vdash (\nu k)P \mid Q \triangleright \Delta$, as required. The other direction is easy.
Case $(\nu u)\ \mathbf{0} \equiv \mathbf{0}$**.** Standard by Weakening and Strengthening.
Case $\mathtt{def}\ D\ \mathtt{in}\ \mathbf{0} \equiv \mathbf{0}$**.** Similar to the first case using Weakening and Strengthening.
Case $(\nu u)\mathtt{def}\ D\ \mathtt{in}\ P \equiv \mathtt{def}\ D\ \mathtt{in}\ (\nu u)P$ if $u \notin \mathrm{fn}(D)$**.** Similar to the scope opening case using Weakening and Strengthening.
Case $(\mathtt{def}\ D\ \mathtt{in}\ P) \mid Q \equiv \mathtt{def}\ D\ \mathtt{in}\ (P \mid Q)$ if $\mathrm{fpv}(D) \cap \mathrm{fpv}(Q) = \emptyset$**.** Similar with the scope opening case using Weakening and Strengthening.
Case $\mathbf{0}\,; P \equiv P$**.** We show that if $\Gamma \vdash \mathbf{0}\,; P \triangleright \Delta$, then $\Gamma \vdash P \triangleright \Delta$. Suppose

$$\Gamma \vdash \mathbf{0} \triangleright \Delta_1 \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta_2.$$

with $\Delta_1; \Delta_2 = \Delta$. Δ_2 only contains $\varepsilon.\mathsf{end}$ or $\bot$, by definition of sequential composition (Definition C.1), $\Delta(k) = \Delta_1(k).\Delta_2(k) = \varepsilon.\Delta_2(k) = \Delta_2(k)$ as required.

Theorem C.2. *The following subject reduction rules hold for a* well-formed topology.

$$\Gamma \vdash P \triangleright \Delta \text{ and } P \longrightarrow P' \text{ implies } \Gamma \vdash P' \triangleright \Delta' \quad \text{such that} \quad \begin{array}{r} \Delta(k) = \alpha \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^\dagger \end{cases} \\ \Delta(k) = \alpha^\dagger \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^\dagger \end{cases} \end{array}$$

Under a well-formed intermediate topology

$$\Gamma \vdash P \triangleright \Delta \text{ and } P \longrightarrow^* P' \text{ implies } \Gamma \vdash P' \triangleright \Delta' \quad \text{such that} \quad \begin{array}{r} \Delta(k) = \alpha \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^\dagger \end{cases} \\ \Delta(k) = \alpha^\dagger \Rightarrow \begin{cases} \Delta'(k) = \alpha \\ \Delta'(k) = \alpha^\dagger \end{cases} \end{array}$$

Proof. We assume that

$$\Gamma \vdash e \triangleright S \quad \text{and} \quad e \downarrow c \quad \text{implies} \quad \Gamma \vdash c \triangleright S \tag{1}$$

and prove the result by induction on the last rule applied.

Case `inwhile`/`outwhile` for N processes $(\nu \tilde{k})(P_{1,1} \mid \ldots \mid P_{R,N})$**.** Assume well-formed topology (4.1) Case $E[e] \longrightarrow E[\texttt{true}]$

By [Ow1],

$$\begin{aligned}
&(\nu \tilde{k})\ (\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\} \\
&\qquad \mid \langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{ \\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\} \text{ when } i < S \\
&\qquad \mid \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\}) \text{ when } S < i \leq N \\
\longrightarrow^*\ &(\nu \tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}];\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e')\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\} \\
&\qquad \mid k_{1,1} \dagger [\texttt{true}] \mid \ldots \mid k_{1,n_{1,1}} \dagger [\texttt{true}] \\
&\qquad \mid \langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{ \\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\} \text{ when } i < S \\
&\qquad \mid \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\}) \text{ when } S < i \leq N
\end{aligned}$$

$$\begin{aligned}
&\Gamma \vdash (Q_1;P_{1,1} \mid k_{1,1} \dagger [\texttt{true}] \mid \ldots \mid k_{1,n_{1,1}} \dagger [\texttt{true}] \mid P_{r,i} \text{ when } i < S \mid P_{R,i} \text{ when } S < i \leq N) \\
&\quad \triangleright \{k_{1,1} : T_{1,1};![T_{1,1}]^* \circ ?[T_{1'_{1'},1}]^{*\dagger},\ldots,k_{1,n_{1,1}} : T_{1,n_{1,1}};![T_{1,n_{1,1}}]^* \circ ?[T_{1'_{1'},n_{1'1'}}]^{*\dagger}, \\
&\qquad k_{r_i,1} : ![T_{r_i,1}]^* \circ ?[T_{r'_{i'},1'}]^*,\ldots,k_{r_i,n_{r,i}} : ![T_{r_i,n_{r,i}}]^* \circ ?[T_{r'_{i'},n_{r'_{i'},1'}}]^*\}
\end{aligned}$$

By [IwE1],

$$
\begin{aligned}
&(\nu\tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}];\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\}\\
&\quad\mid k_{1,1}\dagger[\texttt{true}]\mid\ldots\mid k_{1,n_{1,1}}\dagger[\texttt{true}]\\
&\quad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S\\
&\quad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N\\
\longrightarrow^*\ &(\nu\tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}];\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\}\\
&\quad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\texttt{true})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S \text{ and } r=2\\
&\quad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S\\
&\quad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash(Q_1;P_{1,1}\mid P_{r,i}\text{ when } i<S\mid P_{R,i}\text{ when } S<i\leq N)\\
&\quad\triangleright\{k_{1,1}\colon T_{1,1};![T_{1,1}]^*\circ T_{1'_{1'},1};?[T_{1'_{1'},1}]^*,\ldots,k_{1,n_{1,1}}\colon T_{1,n_{1,1}};![T_{1,n_{1,1}}]^*\circ T_{1'_{1'},n_{1'_{1'},1'}};?[T_{1'_{1'},n_{1'_{1'},1'}}]^*\\
&\qquad k_{r_i,1}\colon ![T_{r_i,1}]^*\circ?[T_{r'_{i'},1'}]^*,\ldots,k_{r_i,n_{r,i}}\colon ![T_{r_i,n_{r,i}}]^*\circ?[T_{r'_{i'},n_{r',i'}}]^*\text{ where r = 2}\\
&\qquad k_{r_i,1}\colon ![T_{r_i,1}]^*\circ?[T_{r'_{i'},1'}]^*,\ldots,k_{r_i,n_{r,i}}\colon ![T_{r_i,n_{r,i}}]^*\circ?[T_{r'_{i'},n_{r',i'}}]^*\}
\end{aligned}
$$

By [Ow1],

$$
\begin{aligned}
&(\nu\tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}];\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\}\\
&\quad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\texttt{true})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S \text{ and } r=2\\
&\quad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S\\
&\quad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N\\
\longrightarrow^*\ &(\nu\tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}];\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\}\\
&\quad\mid Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}];\\
&\qquad\qquad\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S \text{ and } r=2\\
&\quad\mid k_{2_i,1}\dagger[\texttt{true}]\mid\ldots\mid k_{2_i,n_{2_{i,1}}}\dagger[\texttt{true}]\\
&\quad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S\\
&\quad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N
\end{aligned}
$$

$$\Gamma \vdash (Q_1;P_{1,1} \mid Q_i;P_{2,i} \mid k_{2_i,n_{2_i,1}} \dagger [\texttt{true}] \mid \ldots \mid k_{2_i,n_{2_i,1}} \dagger [\texttt{true}] \mid P_{r,i} \text{ when } i < S \mid P_{R,i} \text{ when } S < i \leq N)$$
$$\triangleright \{k_{1,1} : T_{1,1}; ![T_{1,1}]^* \circ T_{1'_{1'},1}; ?[T_{1'_{1'},1}]^*, \ldots, k_{1,n_{1,1}} : T_{1,n_{1,1}}; ![T_{1,n_{1,1}}]^* \circ T_{1'_{1'},n_{1'_{1'},1'}}; ?[T_{1'_{1'},n_{1'_{1'},1'}}]^*,$$
$$k_{r_i,1} : T_{r_i,1}; ![T_{r_i,1}]^* \circ ?[T_{r'_{i'},1'}]^{*\dagger}, \ldots, k_{r_i,n_{r,i}} : T_{r_i,n_{r,i}}; ![T_{r_i,n_{r,i}}]^* \circ ?[T_{r'_{i'},n_{r',i'}}]^{*\dagger} \text{ where r = 3},$$
$$k_{r_i,1} : ![T_{r_i,1}]^* \circ ?[T_{r'_{i'},1'}]^*, \ldots, k_{r_i,n_{r,i}} : ![T_{r_i,n_{r,i}}]^* \circ ?[T_{r_i,n_{r',i'}}]^* \}$$

By repeatedly apply [Ow1] and [IwE1] for i = 3 to j, where r_j = R - 1

$$\begin{aligned}
&(\nu \tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]; \langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\} \\
&\quad \mid Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]; \\
&\qquad\qquad \langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{ \\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\} \text{ when } i < S \text{ and } r = 2 \\
&\quad \mid k_{2_i,1} \dagger [\texttt{true}] \mid \ldots \mid k_{2_i,n_{2_i,1}} \dagger [\texttt{true}] \\
&\quad \mid \langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{ \\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\} \text{ when } i < S \\
&\quad \mid \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\}) \text{ when } S < i \leq N \\
\longrightarrow^*\longrightarrow^*\ &(\nu \tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]; \langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\} \\
&\quad \mid Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]; \\
&\qquad\qquad \langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{ \\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\} \text{ when } i < S \\
&\quad \mid k_{R-1_i,1} \dagger [\texttt{true}] \mid \ldots \mid k_{R-1_i,n_{R-1_i,1}} \dagger [\texttt{true}] \\
&\quad \mid \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\}) \text{ when } S < i \leq N
\end{aligned}$$

$$\Gamma \vdash (Q_1;P_{1,1} \mid Q_i;P_{r,i} \text{ when } i < S \mid k_{R-1_i,1} \dagger [\texttt{true}] \mid \ldots \mid k_{R-1_i,n_{R-1_i,1}} \dagger [\texttt{true}] \mid P_{R,i} \text{ when } S < i \leq N)$$
$$\triangleright \{k_{1,1} : T_{1,1}; ![T_{1,1}]^* \circ T_{1'_{1'},1}; ?[T_{1'_{1'},1}]^*, \ldots, k_{1,n_{1,1}} : T_{1,n_{1,1}}; ![T_{1,n_{1,1}}]^* \circ T_{1'_{1'},n_{1'_{1'},1'}}; ?[T_{1'_{1'},n_{1'_{1'},1'}}]^*,$$
$$k_{r_i,1} : T_{r_i,1}; ![T_{r_i,1}]^* \circ T_{r'_{i'},1'}; ?[T_{r'_{i'},1'}]^*, \ldots, k_{r_i,n_{r,i}} : T_{r_i,n_{r,i}}; ![T_{r_i,n_{r,i}}]^* \circ T_{r'_{i'},n_{r',i'}}; ?[T_{r'_{i'},n_{r',i'}}]^*,$$
$$k_{R-1_i,1} : ![T_{R-1_i,1}]^* \circ ?[T_{R-1'_{i'},1'}]^{*\dagger}, \ldots, k_{R-1_i,n_{R-1,i}} : ![T_{R-1_i,n_{R-1_i,1}}]^* \circ ?[T_{R-1'_{i'},n_{R-1',i'}}]^{*\dagger} \}$$

Finally apply [Iw1],

$$
\begin{aligned}
&(\nu\tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}];\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\}\\
&\qquad\mid Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}];\\
&\qquad\qquad\qquad\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\ \text{when } i<S\\
&\qquad\mid k_{R-1_i,1}\dagger[\texttt{true}]\mid\ldots\mid k_{R-1_i,n_{R-1_i,1}}\dagger[\texttt{true}]\\
&\qquad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\ \text{when } S<i\leq N\\
\longrightarrow^*\ &(\nu\tilde{k})\ (Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}];\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\}\\
&\qquad\mid Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}];\\
&\qquad\qquad\qquad\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\ \text{when } i<S\\
&\qquad\mid Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}];\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\ \text{when } S<i\leq N
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash(Q_1;P_{1,1}\mid Q_i;P_{r,i}\ \text{when}\ i<S\mid Q_i;P_{R,i}\ \text{when}\ S<i\leq N)\\
&\quad\triangleright\{k_{1,1}:T_{1,1};![T_{1,1}]^*\circ T_{1'_{1'},1};?[T_{1'_{1'},1}]^*,\ldots,k_{1,n_{1,1}}:T_{1,n_{1,1}};![T_{1,n_{1,1}}]^*\circ T_{1'_{1'},n_{1'_{1'},1'}};?[T_{1'_{1'},n_{1'_{1'},1'}}]^*,\\
&\qquad k_{r_i,1}:T_{r_i,1};![T_{r_i,1}]^*\circ T_{r'_{i'},1'};?[T_{r'_{i'},1'}]^*,\ldots,k_{r_i,n_{r,i}}:T_{r_i,n_{r,i}};![T_{r_i,n_{r,i}}]^*\circ T_{r'_{i'},n_{r',i'}};?[T_{r'_{i'},n_{r',i'}}]^*\}\\
&\Gamma\vdash(Q_1;P_{1,1}\mid Q_i;P_{r,i}\ \text{when}\ i<S\mid Q_i;P_{R,i}\ \text{when}\ S<i\leq N)\\
&\quad\triangleright\{k_{1,1}:\bot,\ldots,k_{1,n_{1,1}}:\bot,k_{r_i,1}:\bot,\ldots,k_{r_i,n_{r,i}}:\bot\}
\end{aligned}
$$

Case $E[e]\longrightarrow E[\texttt{false}]$
By [Ow2],

$$
\begin{aligned}
&(\nu\tilde{k})\ (\langle k_{1,1},\ldots,k_{1,n_{1,1}}\rangle.\texttt{outwhile}(e)\{Q_1[k_{1,1},\ldots,k_{1,n_{1,1}}]\}\\
&\qquad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\ \text{when } i<S\\
&\qquad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\ \text{when } S<i\leq N\\
\longrightarrow^*\ &(\nu\tilde{k})\ (\ \mathbf{0}\ \mid k_{1,1}\dagger[\texttt{false}]\mid\ldots\mid k_{1,n_{1,1}}\dagger[\texttt{false}]\\
&\qquad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\ \text{when } i<S\\
&\qquad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\ \text{when } S<i\leq N
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash(\ \mathbf{0}\ \mid k_{1,1}\dagger[\texttt{false}]\mid\ldots\mid k_{1,n_{1,1}}\dagger[\texttt{false}]\mid P_{r,i}\ \text{when}\ i<S\mid P_{R,i}\ \text{when}\ S<i\leq N)\\
&\quad\triangleright\{k_{1,1}:\tau.\mathsf{end}\circ?[T_{1'_{1'},1}]^{*\dagger},\ldots,k_{1,n_{1,1}}:\tau.\mathsf{end}\circ?[T_{1'_{1'},n_{1'_{1'},1'}}]^{*\dagger},\\
&\qquad k_{r_i,1}:![T_{r_i,1}]^*\circ?[T_{r'_{i'},1'}]^*,\ldots,k_{r_i,n_{r,i}}:![T_{r_i,n_{r,i}}]^*\circ?[T_{r'_{i'},n_{r',i'}}]^*\}
\end{aligned}
$$

By [IwE2],

$$
\begin{aligned}
&(\nu\tilde{k})\,(\,\mathbf{0}\mid k_{1,1}\dagger[\texttt{false}]\mid\ldots\mid k_{1,n_{1,1}}\dagger[\texttt{false}]\\
&\qquad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S\\
&\qquad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N\\
\longrightarrow^*\;&(\nu\tilde{k})\,(\,\mathbf{0}\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\texttt{false})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S \text{ and } r=2\\
&\qquad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S\\
&\qquad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash(\,\mathbf{0}\mid P_{r,i}\text{ when } i<S\mid P_{R,i}\text{ when } S<i\leq N)\\
&\quad\triangleright\{k_{1,1}\colon\tau.\mathsf{end}\circ\tau.\mathsf{end},\ldots,k_{1,n_{1,1}}\colon\tau.\mathsf{end}\circ\tau.\mathsf{end}\\
&\qquad k_{r_i,1}\colon![T_{r_i,1}]^*\circ?[T_{r'_{i'},1'}]^*,\ldots,k_{r_i,n_{r,i}}\colon![T_{r_i,n_{r,i}}]^*\circ?[T_{r'_{i'},n_{r',i'}}]^*\}
\end{aligned}
$$

By [Ow2],

$$
\begin{aligned}
&(\nu\tilde{k})\,(\,\mathbf{0}\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\texttt{false})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S \text{ and } r=2\\
&\qquad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S\\
&\qquad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N\\
\longrightarrow^*\;&(\nu\tilde{k})\,(\,\mathbf{0}\mid k_{2_i,1}\dagger[\texttt{false}]\mid\ldots\mid k_{2_i,n_{2_i,1}}\dagger[\texttt{false}]\\
&\qquad\mid\langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\text{ when } i<S \text{ and } r\neq 2\\
&\qquad\mid\langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\text{ when } S<i\leq N
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash(\,\mathbf{0}\mid k_{2_i,1}\dagger[\texttt{false}]\mid\ldots\mid k_{2_i,n_{2_i,1}}\dagger[\texttt{false}]\mid P_{r,i}\text{ when } i<S\mid P_{R,i}\text{ when } S<i\leq N)\\
&\quad\triangleright\{k_{1,1}\colon\tau.\mathsf{end},\ldots,k_{1,n_{1,1}}\colon\tau.\mathsf{end}\\
&\qquad k_{r_i,1}\colon\tau.\mathsf{end}\circ?[T_{r'_{i'},1'}]^{*\dagger},\ldots,k_{r_i,n_{r,i}}\colon\tau.\mathsf{end}\circ?[T_{r'_{i'},n_{r',i'}}]^{*\dagger}\text{ where r = 2},\\
&\qquad k_{r_i,1}\colon![T_{r_i,1}]^*\circ?[T_{r'_{i'},1'}]^*,\ldots,k_{r_i,n_{r,i}}\colon![T_{r_i,n_{r,i}}]^*\circ?[T_{r'_{i'},n_{r',i'}}]^*\}
\end{aligned}
$$

By repeatedly apply [Ow2] and [IwE2] for i = 3 to j, where r_j = R - 1

$$
\begin{aligned}
&(\nu\tilde{k})\ (\ \mathbf{0}\ \mid k_{2_i,1}\,\dagger\,[\texttt{false}]\mid \ldots \mid k_{2_i,n_{2_i,1}}\,\dagger\,[\texttt{false}]\\
&\qquad \mid \langle k_{r_i,1},\ldots,k_{r_i,n_{r,i}}\rangle.\texttt{outwhile}(\langle k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}\rangle.\texttt{inwhile})\{\\
&\qquad\qquad Q_i[k_{r_i,1},\ldots,k_{r_i,n_{r,i}},k'_{r'_{i'},1},\ldots,k'_{r'_{i'},n_{r',i'}}]\}\ \text{when } i<S\\
&\qquad \mid \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\ \text{when } S<i\leq N\\
\longrightarrow^*\longrightarrow^*\ &(\nu\tilde{k})\ (\ \mathbf{0}\ \mid\ \mathbf{0}\ \text{ when } i<S \mid k_{R-1_i,1}\,\dagger\,[\texttt{false}]\mid \ldots \mid k_{R-1_i,n_{R-1_i,1}}\,\dagger\,[\texttt{false}]\\
&\qquad \mid \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\ \text{when } S<i\leq N
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash (\ \mathbf{0}\ \mid\ \mathbf{0}\ \text{ when } i<S \mid k_{R-1_i,1}\,\dagger\,[\texttt{false}]\mid \ldots \mid k_{R-1_i,n_{R-1_i,1}}\,\dagger\,[\texttt{false}]\mid P_{R,i}\ \text{when } S<i\leq N)\\
&\quad \triangleright\{k_{1,1}:\tau.\mathsf{end},\ldots,k_{1,n_{1,1}}:\tau.\mathsf{end},k_{r_i,1}:\tau.\mathsf{end},\ldots,k_{r_i,n_{r,i}}:\tau.\mathsf{end},\\
&\qquad k_{R-1_i,1}:\tau.\mathsf{end}\circ ?[T_{R-1_i,1}]^{*\dagger},\ldots,k_{R-1_i,n_{R-1,i}}:\tau.\mathsf{end}\circ ?[T_{R-1'_{i'},n_{R-1',i'}}]^{*\dagger}\}
\end{aligned}
$$

Lastly apply [Iw2],

$$
\begin{aligned}
&(\nu\tilde{k})\ (\ \mathbf{0}\ \mid\ \mathbf{0}\ \text{ when } i<S\\
&\qquad \mid k_{R-1_i,1}\,\dagger\,[\texttt{false}]\mid \ldots \mid k_{R-1_i,n_{R-1_i,1}}\,\dagger\,[\texttt{false}]\\
&\qquad \mid \langle k_{R_i,1},\ldots,k_{R_i,n_{R,i}}\rangle.\texttt{inwhile}\{Q_i[k_{R_i,1},\ldots,k_{R_i,n_{R,i}}]\})\ \text{when } S<i\leq N\\
\longrightarrow^*\ &(\nu\tilde{k})\ (\ \mathbf{0}\ \mid\ \mathbf{0}\ \text{ when } i<S \mid\ \mathbf{0}\ \text{ when } S<i\leq N)
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\vdash (\ \mathbf{0}\ \mid\ \mathbf{0}\ \text{ when } i<S \mid\ \mathbf{0}\ \text{ when } S<i\leq N)\\
&\quad \triangleright\{k_{1,1}:\tau.\mathsf{end},\ldots,k_{1,n_{1,1}}:\tau.\mathsf{end},k_{r_i,1}:\tau.\mathsf{end},\ldots,k_{r_i,n_{r,i}}:\tau.\mathsf{end}\}
\end{aligned}
$$

Finally, apply [Bot].
For other cases, the proof is similar to [27, P. 56-60]