MEng Individual Project Report

HIGH PERFORMANCE PARALLEL DESIGN BASED ON SESSION PROGRAMMING

Nicholas Ng (cn06@doc.ic.ac.uk)
Department of Computing
Imperial College London

Supervisor
Nobuko Yoshida (yoshida@doc.ic.ac.uk)

Second Marker
Wayne Luk (wl@doc.ic.ac.uk)

Contents

Abstract

Session programming is a programming model based on the theory of session types, a typing system for π-calculus. Session types is developed to model structured interaction between processes and correctly typed process will have the property of communication safety. Session Java (SJ) is a full implementation of session types in Java. In this project, We aim to introduce the session programming model to Axel, a heterogeneous cluster with both FPGAs and GPUs as hardware accelerators to design communication safe parallel algorithms.

We give an implementation of a parallel algorithm, n-body simulation, on the Axel cluster, using SJ and FPGAs. We also give a translation of our SJ n-body simulation into C to get a higher performance. We find good performance improvements in both implementations, without compromising safety property of our program.

Finally, we present a formalisation of two new *multichannel* SJ primitives for parallel programming. We use the formalisation to prove the correctness of our n-body implementation and generalise the proof to a ring topology used by parallel algorithms in SJ.

Acknowledgements

I would like to thank the following people, without whom this project might not be as successful. My supervisor, Dr. Nobuko Yoshida for her enthusiasm and guidance throughout the project, my second marker, Prof. Wayne Luk for his constructive advices and feedback on the project, Andi Bejleri for the crash course on session types, Brittle Tsoi for his help on FPGA and the Axel cluster, Olivier Pernet for his insights and advices to the direction of the project, Raymond Hu for advices and support on SJ, Wilhelm Kleiminger for moep and proofreading my final report, and finally, my family for their support and love throughout my four years at university.

Chapter 1

Introduction

In 1965, Gordon Moore predicted that the number of transistors on a chip doubles about every two years [?]. 45 years later, Moore's Law remained valid and is generalised to describe the performance growth of microprocessors. Until recent years, microprocessor manufacturers have enjoyed performance increase simply by cramming more transistors on a single microprocessor. As the cost of performance rose to an unfeasible level due to power consumption, to keep up with Moore's Law, research and development on computer architecture turned towards parallelising techniques on existing hardware. Multicore processor architecture rose in popularity, today it is easy to find dual-, quad-, even hexa-cores on a single processor dice, and we are expecting to see processors with as much as 80-cores in the next few years [?].

On a much bigger scale, another type of parallel architecture is *computer clusters*. It is a form of distributed computing, where multiple standalone cluster nodes are connected and computation jobs are shared between the nodes. Each node can work on their partition of jobs in parallel. From outside the cluster, the jobs submitted to the cluster are completed as if a single computer is used. As a cluster can be built using commodity hardware, it is a cost-effective way of building supercomputers.

Parallel models of computing has shown it self as a promising direction to higher performance computer architecture.

Another trend that saw a lot of interest lately is the use of *hybrid architectures* to achieve high performance. Instead of using a centralised computation model based on the CPU, parts of computations are delegated to other specially designed hardware which can perform the computation more efficiently.

Field Programmable Gate Array (FPGA) is a type of reconfigurable integrated circuit. FPGAs can be configured to represent software instructions directly in hardware during runtime. On CPUs, all instructions need to go through the fetch-decode-execute cycle before they can be executed. Implementations in FPGAs do not require the fetch-decode phase since the instructions are already represented in the hardware circuit. As a result, FPGAs are much more efficient than CPUs in computation-heavy tasks. Also because the computations are done in hardware circuit, pipelined instructions can be executed in parallel as a physical property of electric circuits.

A modern GPU can double as a many-core general purpose processor. A GPU has hundreds of processing cores which are very capable at floating point computation, as they are usually used for graphics calculations. *Common Unified Device Architecture* (CUDA) and ATI's *Stream* are software frameworks that allow the use of the GPU cores for non-graphical computations. This is known as *General Purpose computation on Graphics Processing Units* (GPGPU). Compared to traditional CPUs where support for *Single Instruction Multiple Data* (SIMD) is limited to the *Streaming SIMD Extensions* (SSE) instruction set which can work on at most four single precision floating point number in parallel, the GPU can work on hundreds of data in parallel in the GPU cores.

The performance edge in parallel architectures do not come without its own problems. Parallel programming is a much less understood model than traditional serial programming model. Some of the problems were solved by the implicit programming model, where programmers do not need to understand parallel programming and parallelisation is done implicitly in hardware or by compiler optimisations. The advantage of this technique is a guarantee of a certain degree of parallelism and the correctness of the parallelised section since most of the optimisations are rather conservative. On the other hand, this approach might not always give the optimal result if a there exists specific ways to parallelise the code as the programmer do not have control over the implicit optimisations. The alternative, explicit parallelism comes typically in form of message-passing. Often, small trivial mistakes in the program will result in parallel synchronisation issues, race conditions, or deadlocks. Combined with the interleaving of executions, parallel programming in this model cannot guarantee communication safety and is difficult to identify problems as the execution sequence can be undeterministic. For years, computer science theorists seek to understand parallel programming model and search for solutions by formalising and modelling concurrent processes. Amongst the more active researches, the Actor model and process algebras, such as Calculus of Communicating Systems CCS [**?**] and its successor π-calculus [**?**, **?**], are fields that found most success. With a model of processes interactions, it is much easier to understand the properties of concurrent systems to prevent the issues common in parallel programming.

Session types [**?**, **?**, **?**] were developed as a typing system for π-calculus. Interactions between parties are conducted over private channels called sessions. A session type specifies the sequence and typing of interactions in the session. Session types captures the fact that communicating parties must have a compatible typing between them. For example, if a sender intends to send an integer, the receiver must be expecting to receive an integer as well; otherwise there would be problem. By analysing the session typing of communicating processes and making sure only compatible processes can start a session, we are able to show that *communication safety property* holds for session-typed processes - deadlocks are not possible. If a parallel system is modelled in π-calculus is shown to be type-safe by session types, we are confident to say that the system is communication safe.

Session types as a typing system alone cannot be used directly as a design tool. *Session Java* (SJ) [**?**] is an implementation of session types as an extension of Java. It is designed to be a non-intrusive addition to Java and integrates well with the object oriented setting. SJ brought the full theoretical session programming framework to a programming language in common use. It is a powerful tool for programmers that can ensure session compatibility within the programming language, without first modelling interactions in the pure theoretical framework, to create communication safe code.

1.1 This project

This project aims to explore ways of applying session programming to heterogeneous clusters, which uses acceleration hardware such as FPGAs or GPUs.

We wish to demonstrate uses of session-type based Java to design parallel algorithms which are communication safe, efficient and easily readable on our target platform. We also aim to generalise and extend our approach to other similar platforms so designs on these platforms can also make use of session programming.

1.2 Contributions

In this report we document our findings and results on applying session programming to design parallel algorithms on heterogeneous clusters. We made the following contributions:

- Introduced an architecture for high performance parallel application design with *session Java* (SJ) using heterogeneous hardware (§3.2).

- Implemented a parallel n-body simulation in SJ accelerated by FPGA on a heterogeneous cluster (§3.3), with full sets of benchmark results to compare the performance with and without acceleration hardware (§5.3). Our implementation using SJ and FPGA yields up to 2 times speedup in the best performance.

- Implemented a C library that can be used for session programming in the C programming language (§3.4.2), and a C implementation of n-body simulation translated from SJ to demonstrate the use of the library (§3.4). The translated code has on average 5 times speedup over SJ implementation.

- Presented a formalisation and first correctness proof of a pair of new multichannel SJ primitives - **`inwhile`** and **`outwhile`** in SJ, designed to implement parallel algorithms. (§4.1) The two primitives can represent parallel topologies more naturally and were shown to be more efficient than its single-channel counterpart.

- Proved our implementation of n-body simulation deadlock free, using the formalisation of the new multichannel SJ primitives (§4.7).

Report organisation

The report contains six chapters and each chapter is organised as follows:

- Chapter 2: Background will cover the background theories behind session types and session Java. We will also introduce our target platform, a heterogeneous cluster called Axel which contains FPGAs and GPUs as processing elements.
- Chapter 3: Design and Implementation will give the design and implementation details of the main result of the project, an implementation of n-body simulation using SJ and FPGA. We will also include a version of the implementation translated from SJ to C, which is more suitable for deployment on high performance clusters.
- Chapter 4: Correctness proof of n-body implementation will detail an extension to the session type introduced in [**?**] to include multichannel `inwhile` and `outwhile` SJ primitives used for parallel programming in SJ. We will then show a correctness proof of our n-body implementation based on the updated session type.
- Chapter 5: Testing and Evaluation will discuss and evaluate alternative designs and compare benchmark results of different implementations of n-body simulation.
- Chapter 6: Conclusion will conclude our findings of the project and outline potential future works.

Chapter 2

Background

In this chapter we will discuss fundamental background theories of which session programming is based on. This includes π-calculus (§2.1) - the process calculi which is the modelling basis for session programming, session types (§2.2) - the typing system of π-calculus for sessions-based communication and an introduction to session Java (§2.4) which is the main programming tool we are going to use in the project.

Next, we will introduce the target platform for the project (§2.5) - Axel, a heterogeneous cluster with *Field Programmable Gate Arrays* (FPGAs) and *Graphics Processing Units* (GPUs) as acceleration hardware.

Finally, we will briefly look at the current parallel programming model of Axel and session Java to implement our choice of parallel algorithm - n-body simulation (§2.6).

2.1 Pi-calculus

π-calculus [**?**] is a process calculus proposed by Milner, Parrow and Walker as a successor to *Calculus of Communicating Systems* (CCS) as a model to study concurrent mobile systems. It uses message passing and is distinguished from CCS by its use of names in messages rather that values in CSS. The difference between value passing and name passing is that only name passing allows sending and receiving of *channel names* as messages, so channels can be 'reconfigured' at run time. This makes π-calculus more expressive and suited for mobile processes [**?**]. There are many variants of π-calculus for different applications, due to the notational differences in different domains. π-calculus and its variants lie a foundation for modelling communication systems, from simple asynchronous π-calculus and spi calculus for session types and cryptography [**?**] respectively, to more advanced calculi such as 3π for developmental and systems biology [**?**].

2.1.1 Asynchronous π-calculus

Asynchronous π-calculus is the simplest variant of π-calculus and is the variant which session type (§2.2) is based on.

$P, Q ::=$	processes
$\mathbf{0}$	nil process
$P \mid Q$	parallel composition of P and Q
$(\nu a)P$	generation of a with scope P
$!P$	replication of P
$\bar{u}\langle v\rangle$	output of v on channel u
$u(x).P$	input of *distinct* variables x on u, with continuation P

Fig. 2.1: *Syntax of asynchronous π-calculus*

Syntax

The main difference between full π-calculus and asynchronous π-calculus is asynchronicity. This means after an output action, there is no continuation and the process terminates when the message is delivered. Communication with asynchronous π-calculus is therefore deadlock free. Asynchronous communication can be used to simulate synchronous communication, and is common in distributed systems. Fig. 2.1 shows the syntax of asynchronous π-calculus.

Reduction rules

$$\bar{a}\langle v\rangle \mid a(x).P \;\rightarrow\; P\{{}^{v}/_{x}\} \qquad [\textsc{Com}]$$

$$\frac{P \;\rightarrow\; P'}{P \mid Q \;\rightarrow\; P' \mid Q} \qquad [\textsc{Par}]$$

$$\frac{P \;\rightarrow\; P'}{(\nu a)P \;\rightarrow\; (\nu a)P'} \qquad [\textsc{Res}]$$

$$\frac{P \equiv Q \;\rightarrow\; Q' \equiv P'}{P \;\rightarrow\; P'} \qquad [\textsc{Struct}]$$

Fig. 2.2: *Base reduction rules of asynchronous π-calculus*

Reduction is the method for processes in π-calculus to interact. If reduction is not possible, no action can be performed.

It should be noted that in π-calculus interactions between processes are initiated by parallel composition; with processes alone, π-calculus and the reduction rules are meaningless.

P	$::= \texttt{request}\ a(k)\ \texttt{in}\ P$	session request
	$\mid \texttt{accept}\ a(k)\ \texttt{in}\ P$	session acceptance
	$\mid k![\tilde{e}];P$	data sending
	$\mid k?(\tilde{x})\ \texttt{in}\ P$	data reception
	$\mid k \lhd l;P$	label selection
	$\mid k \rhd \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\}$	label branching
	$\mid \texttt{throw}\ k[k'];P$	channel sending
	$\mid \texttt{catch}\ k(k')\ \texttt{in}\ P$	channel reception
	$\mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional branch
	$\mid P \mid Q$	parallel composition
	$\mid \texttt{0}$	inaction
	$\mid (\nu u)\ P$	name/channel hiding
	$\mid \texttt{def}\ D\ \texttt{in}\ P$	recursion
	$\mid X[\tilde{e}\tilde{k}]$	process variables
e	$::= c$	constant
	$\mid e+e' \mid e-e' \mid e\times e \mid \texttt{not}(e) \mid \ldots$	operators
D	$::= X_1(\tilde{x}_1\tilde{k}_1) = P_1\ \texttt{and} \cdots \texttt{and}\ X_n(\tilde{x}_n\tilde{k}_n) = P_n$	declaration for recursion

Fig. 2.3: *Session calculus syntax from [?]*

2.2 Session types

A session is a predefined sequence of exchanging messages otherwise known as a protocol. Session types were developed as a typing system of the π-calculus for use by communication-based concurrent programming languages with basic communication constructs. The theory of session types is defined in terms of session calculus based on asynchronous π-calculus, originally introduced by Honda et al. [?] and their work was subsequently revised by Yoshida and Vasconcelos [?] which became the basis of session Java. Session calculus is a building block of session types, where session type defines compatible sessions in terms of session calculus.

2.2.1 Syntax of session calculus

Basic constructs Processes exchange messages by pairs of send and receive actions. Data sending and receiving are the basic constructs in session type, along with inaction process, parallel composition of processes and name restriction (ν). These basic constructs can be directly translated to asynchronous π-calculus.

$$\begin{aligned}
\texttt{accept}\ a(k)\ \texttt{in}\ P &= \texttt{request}\ a(k)\ \texttt{in}\ P \\
k![\tilde{e}];P &= k?(\tilde{x})\ \texttt{in}\ P \\
k \lhd l;P &= k \rhd \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\} \\
\texttt{throw}\ k[k'];P &= \texttt{catch}\ k(k')\ \texttt{in}\ P \\
\mathbf{0} &= \mathbf{0}
\end{aligned}$$

Fig. 2.4: *Dual actions*

Label branching Label branching is a feature in session calculus for *structured external choice*. Without label branching, it is not possible for sessions to exhibit different behaviour on different conditions or they will be incompatible. Thus the use of session calculus and session types will be very limited and only useful in serial and simple communications, if choices are not possible.

Label branching is done by sending and receiving a *label*, and based on the content of the label, the session type following the selection action can be different as long as the session type of the counterpart participant remains compatible with the receiver after sending the said label.

Session delegation Because of the name-passing property of π-calculus, it is possible to pass more information and use the sessions more flexibly. This allows sessions to be passed to other processes as a parameter via a channel, subprocesses can use the received session as a session rather than a value. By offloading parts of responsibilities of the parent process to the subprocesses, we can distribute processing to lower level or smaller processes, without sacrificing the advantages of using session-types because all delegated processes also follow a subset of session-type from the top level. Most importantly, the top-level process does not need to be informed about the delegation which allows a higher level view when designing distributed systems with session types.

Duality We mentioned the importance of interaction in the previous section (§2.1.1). A correct session type for two interacting process requires the sessions in the same channel to be 'associated with complementary behaviours' [**?**, Definition 5.2], this important requirement provides the theoretical basis for communication safe processes. In the syntax given in Fig. 2.3, complementary actions are shown in Fig. 2.4; these pairs of actions, when composed together, will reduce without getting stuck. A sound type system will not cause stuck errors if the implementation is correct.

As an example of session type, Example 1 is definition of a simple sum server system that adds and returns the sum of two numbers in *SumServer* supplied by *SumClient*. This will be be used in subsequent sections to demonstrate the similarities and difference between session type and its derivatives.

$$\begin{array}{lll}
G ::= & & \text{Global types} \\
 & \mid\ \mathtt{p} \to \mathtt{p}' : \langle U \rangle . G & \text{Message} \\
 & \mid\ \mathtt{p} \to \mathtt{p}' : \{l_k : G_k\}_{k \in K} & \text{Branching} \\
 & \mid\ \mu \mathbf{x}.G & \text{Recursion} \\
 & \mid\ \mathbf{x} & \text{Type variable} \\
 & \mid\ G\ \mathtt{i} & \text{Application} \\
 & \mid\ \mathsf{end} & \text{Null}
\end{array}$$

Fig. 2.5: *Global types and type reduction from [**?**]*

Example 1.

$$\begin{aligned}
\mathit{SumServer} &= \mathtt{accept}\ a(k)\ \mathtt{in}\ k?(\tilde{x})\ \mathtt{in}\ k?(\tilde{y})\ \mathtt{in}\ k![x+y];\mathbf{0} \\
\mathit{SumClient} &= (\nu k)\ \mathtt{request}\ a(k)\ \mathtt{in}\ k![42];k![77];k?(\mathit{result})\ \mathtt{in}\ \mathbf{0} \\
\mathit{SummingSystem} &= (\nu a)\ \mathit{SumClient} \mid \mathit{SumServer}
\end{aligned}$$

2.3 Multiparty session types

Session types introduced in §2.2 describes communication between two parties. When a communication involves more than two participants, the communication can be modelled by multiple binary sessions between any two of the participants. All communications between any two participants can be guaranteed compatible and error free by the safety property of binary session type. However, binary sessions cannot prevent interleaving of sessions in a communication with multiple binary sessions. Interleaving sessions might allow incorrect communication logic or cause problems such as deadlocks because the execution sequence is not determined at design time.

This will be a problem when using binary session types in a system where different participants of communication are implemented by different parties, and each party is given a protocol specification which they code according to. The final product will be correct in the local view but could be incorrect in the global view because the parties do not have the information of the global session type. Therefore to design a correct multiparty protocol, no assumption of execution order should be made and the communication of different participants should be specified explicitly.

The basic constructs of a multiparty session type are almost identical to binary session type, but add a global type (Fig. 2.5) on top of the endpoint (or local) session type. The global type specifies the global progress of the communication, and *projects* to endpoint session type for each of the participants.

Global session types provide a theoretical basis to prove that a communication is correct in the global view, where the participants and order of communication are defined explicitly in the design. The projection of global type to local type will ensure such properties are preserved after the operation.

2.4 Session programming with SJ

Session types are a basis for session-based programming, but do not describe a standalone programming language. As a result, session types and the object-oriented programming language Java are combined to create *Session Java* (SJ) [**?**, **?**] [1]. SJ is an extension of Java and thus the syntax of SJ is identical to Java, with extra primitives for session programming. This is best illustrated with a simple example:

```
public class Server {
   // Session declaration
   final noalias protocol p_server {
      sbegin.?(int).?(int).!<int>
   }
   public run(int port) {
      final noalias SJServerSocket svr;
      final noalias SJSocket sock;
      try (svr) {
         svr = SJServerSocketImpl.create(p_server, port);
         try (sock) {
            sock = svr.accept();
            int x = sock.receiveInt(); int y = sock.receiveInt();
            sock.send(x + y);
         } catch ( ... ) {}
      } catch ( ... ) {}
   }
}

public class Client {
   // Session declaration
   final noalias protocol p_client {
      cbegin.!<int>.!<int>.?(int)
   }
   public run(String host, int port) {
      final noalias SJService svc
            = SJService.create(p_client,host,port);
      final noalias SJSocket sock;
      try (sock) {
         sock = svc.request();
         sock.send(42); sock.send(77);
         int result = sock.receiveInt();
      } catch ( ... ) {}
   }
}
```

Listing 2.1: *SJ sum server/client*

[1] Currently, Session Java is only an implementation of *binary* session type and extension for *multiparty* session type is planned.

We now look at the *SumServer/SumClient* example again to show a basic communication system with SJ. A session calculus version was given in Example 1 in the previous section.

1. A client sends two numbers to the server
2. The server replies with the sum of the two numbers received

The communication primitives of SJ are similar to conventional socket programming (`request`, `accept`, `send`, `receive`), except for the `protocol` code block. The `protocol` defines the *session typing* of the program, introduced in the previous section (§2.2). With the session type of the program defined, it is possible to [**?**]:

1. Ensure the implementation conforms to the specified `protocol` by static checks at compile time.
2. Check that the two communicating programs are compatible, by a duality check of the `protocol` at the start of communication.
3. Simplify checking protocol correctness by abstracting away implementation details and checking only the session type.

Table 2.1 shows the relationship between protocol and Java code, which allows the communication safety checks mentioned above.

Protocol	SJ code		Line
`sbegin`	`accept()`	Starts a server session	9
`cbegin`	`request()`	Starts a client session	23
`!<`*datatype*`>`	`send()`	Sends an object with *datatype*	24
`?(`*datatype*`)`	`receive()`[2]	Receives an object with *datatype*	10

Table 2.1: *Session type and the corresponding Java code*

Below we list three scenarios in the *SumServer* example that can demonstrate the benefits of the safety checks:

The protocol is correct but the implementation does not conform to the protocol If the implementation of *SumServer* receives three integers instead of two as stated in the protocol, the SJ compiler will throw an exception.

The protocol and implementation are both correct but the protocols are not dual of each other If *SumServer* replies with the result at the end of the execution, and *SumClient* is not receiving the final result, ie. Server: `sbegin.?(int).?(int).!<int>` and Client: `cbegin.!<int>.!<int>`. When the connection between the two processes is established, an incompatible session exception is thrown.

[2]`receiveInt()` is a shortcut to receive an `int`

A logic error exists in the protocol design If *SumServer* sends a result before receiving any values, and *SumClient* is also compatible with the server, ie. Server: `sbegin.!<int>.?(int).?(int)` and Client: `cbegin.?(int).!<int>.!<int>`. The same applies to analysing problems with distributed deadlocks, where the processing of the values is less important than the main source of problem - communication primitives. With SJ the developer can reason about the problem in the protocol level, eg. "Because the result is sent before the numbers are received and processed, therefore changing the arguments to *SumServer* do not influence the result" without necessarily understanding the operation done on the two arguments. Also because of the conformance check, we can be assured that the code implements the protocol *without* looking at the code to find the problem.

In session types and π-calculus, a new channel is 'generated' or 'restricted' by using the operator ν *channelname*. In session programming, this corresponds to the action of creating a new `Socket`. The socket will contain transmission between the participants of communication once it is created. The `send` and `receive` object methods of the socket, and in session calculus you can only input and output on a given channel.

2.4.1 Branching

When programming conditional statements, often different choices will branch to different behaviours of the program. To model that, label branching in session type is used:

Protocol	SJ code	
`!{` $LABEL_0$:$session_0$, $LABEL_1$:$session_1$, ... `}`	`sock.outbranch(LABEL`$_0$`) { code`$_0$` }` `sock.outbranch(LABEL`$_1$`) { code`$_1$` }` ...	Send `LABEL`$_0$ Send `LABEL`$_1$
`?{` $LABEL_0$:$session_0$, $LABEL_1$:$session_1$, ... `}`	`sock.inbranch {` `case LABEL`$_0$`: { code`$_0$` }` `case LABEL`$_1$`: { code`$_1$` }` ... `}`	Receive `LABEL`$_0$ Receive `LABEL`$_1$

Table 2.2: *Branching in session programming*

Table 2.2 shows the receiving and sending of labels. The code blocks `?{}` and `!{}` represent receive label and send label respectively. Sending of labels is usually used in conjunction with conditional statements, for example in Listing 2.2.

```
final noalias protocol p_hwc {
    cbegin.!{LOWER:?(int), UPPER:?(String)}
}
...

if (userInput.equalsIgnoreCase("lower")) {
    sock.outbranch(LOWER) {
        System.out.println("LOWER branch; Server replies with #"
        + sock.recieveInt());
    }
} else {
    sock.outbranch(UPPER) {
        System.out.println("UPPER branch; Server replies with "
        + (String) sock.receive);
    }
}
```

Listing 2.2: *Example usage of label sending*

2.4.2 Iteration

Iteration in session programming translates to replication in π-calculus. In π-calculus processes can be repeated and this forms the loop-body of an iteration. Iteration is not part of the session calculi defined in [**?**] but will be formalised in this report. By using an explicit looping construct that is similar to normal Java programming (**`outwhile`**/**`inwhile`** vs. while), the reasoning of iteration is thus simpler to the programmer. Table 2.3 shows the syntax of **`outwhile`** and **`inwhile`**. The only difference between the two is **`outwhile`** controls the looping condition, and **`inwhile`** reacts passively. The iteration construct therefore also work as a synchronisation mechanism between the sessions. To implement iteration with the same semantic in MPI, the single line code might be expanded to [3]:

```
// outwhile(condition)


while (condition) {
    MPI.COMM_WORLD.Barrier();
    /* outwhile code */
    MPI.COMM_WORLD.Send(
                condition, ...);
}
```

Listing 2.3: **`outwhile`** *in MPJ Express*

```
// inwhile()
MPI.COMM_WORLD.Recv(
            condition, ... );
while (condition) {
    MPI.COMM_WORLD.Barrier();
    /* inwhile code */
    MPI.COMM_WORLD.Recv(
                condition, ...);
}
```

Listing 2.4: **`inwhile`** *in MPJ Express*

Protocol	SJ code

`![` *session in iteration* `]*`	`s1.outwhile(`*`condition`*`){ ... }`
	`s1.outwhile(s2.inwhile; `*`condition`*`){ ... }`

`?[` *session in iteration* `]*`	`sock.inwhile(){ ... }`

Table 2.3: *Iteration in session programming*

[3]Example uses syntax of MPJ Express, a Java MPI implementation

Note that the alternate form of **`outwhile`** that uses **`inwhile`** as condition is a new SJ primitive for implementing parallel algorithm. The formalisation and proofs will be given in §4.1.

2.4.3 Delegation

Sessions can be delegated to other components, to expression delegation of session in the protocol, we simply replace the type of the message by a session, as shown in Table 2.4. *delegated session* in the table represents the session type of the *initialised SJSocket* and of `rcvdSession`. If we look closely the primitives of session delegation is identical to ordinary `send` and receive in SJ, except the content is a session rather than a usual data type (but lends itself to the Java Object model where every type is a subtype of `Object`). Session delegation is an important tool to distribute tasks.

Protocol	SJ code	
`!<`*session*`>`	`sock.send(`*`delegateSJSocket`*`)`	Send a session
`?(`*session*`)`	`SJSocket session = sock.receive()`	Receive a session

Table 2.4: *Session delegation in session programming*

2.4.4 Non session-based alternatives

The implementation of SJ is most similar to that of the MPI standard (§2.4.4) in terms of communication model (message passing) and design. There are also other distributed message passing system such as Java *Remote Method Invocation* RMI, but the design and uses are in a different domain compared to SJ.

Message-Passing Interface (MPI)

MPI is a message-passing library interface specification [?] and is commonly used in the high performance computing field for message-passing based parallelism.

Using MPJ Express, an implementation of the MPI standard in Java, it was shown in [?] that there are many similarities between the two but the main differences are:

MPI has more features SJ does not have multicast-type message send primitive, but theory for multiparty session type §2.3 have been developed for future implementation [?] in SJ.

MPI is a low level protocol which makes it prone to communication mismatch or deadlocks due to explicit message passing [?]. A communication mismatch in MPI such as a `MPI_Send` without a corresponding `MPI_Recv`, will not cause problem until some point in the execution. Scenario 2 of safety check examples above will will cause MPI but not with SJ. In a distributed database system, it would require a rollback on all previous calculations. SJ's safety properties (§2.4) will prevent the incompatible sessions from starting.

SJ has high level session abstraction so the code is more structured and more readable than MPI, this gives the programmer an advantage to focus on more important communication/protocol details.

SJ is not an external library SJ was designed to be a full object-oriented programming language. Implementations of MPI are external libraries since it is only a communications standard. As a domain-specific language, syntax for tasks common to communications programming can be built into the syntax and will be more natural to use, despite the small difference between Java and SJ. Examples include the try-channel syntax to catch exceptions from within a specific channel (line 8 in Listing 2.1), and the different forms of **`outwhile`**/**`inwhile`** as a session-type specific looping technique.

Taking the example of outwhile and inwhile in Listing 2.3 and 2.4 again, the iteration feature in MPI is less readable than in SJ because the special iteration syntax is not found in MPI.

2.4.5 Related work

Implementation of session types had been developed for other languages such as Haskell [**?**, **?**]. Other work on session-type with C-like languages [**?**, **?**] does not take the direction of implementing the full session type system. SJ is the first practical session-type based object-oriented programming language.

2.5 Axel

Axel [**?**] is a heterogeneous computer cluster built at Imperial College. The cluster consists of 18 computing nodes, and each of the nodes contains a x86 CPU, a number of *Graphics Processing Unit*s (GPU) and most of the nodes contain a *Field Programmable Gate Array* (FPGA) device. FPGAs and GPUs are used on Axel as hardware accelerating components.

Axel is the target platform for this project. We wish to deploy SJ on the cluster and use session programming to improve parallel design. Below is an overview of hardware and software currently on the cluster.

2.5.1 Hardware arrangement

NNUS clusters and UNNS clusters There are two ways of grouping hardware accelerators (or *Processing Elements*, PE) in a heterogeneous cluster, namely *Nonuniform Node Uniform Systems* (NNUS) and *Uniform Node Nonuniform Systems* (UNNS).

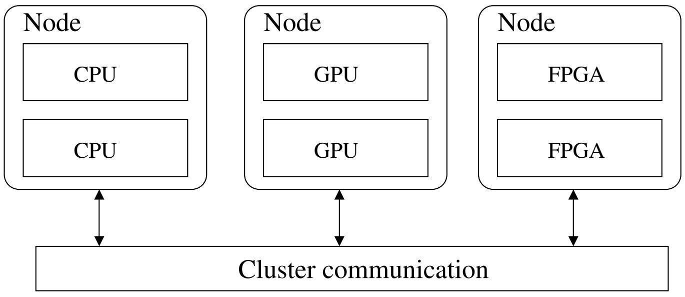


Fig. 2.6: *Nodes in a generic UNNS cluster*

In a UUNS cluster, each node of the cluster hosts a single type of PEs. In the example of Fig. 2.6, three nodes of the cluster hosts CPUs, GPUs and FPGAs respectively. For nodes that hosts non-CPU PEs, special hardware are needed to control the nodes because they cannot run ordinary operating systems. Examples of UUNS clusters are SRC-7 MapStation and RASC server from SGI [**?**].

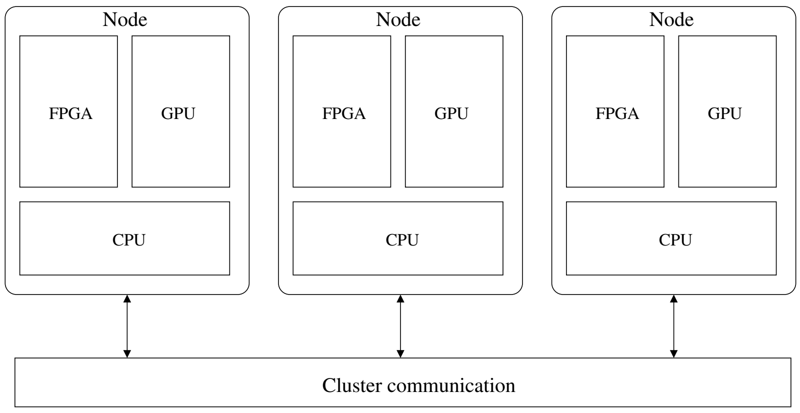


Fig. 2.7: *Nodes in a generic NNUS cluster*

On the other hand, in a NNUS cluster, each node of the cluster contains different PEs (thus *Nonuniform Node*). The PEs of the NNUS cluster example in Fig. 2.7 shown to be on the same node are CPU, FPGA and GPU. All nodes in the cluster have the same arrangement. This makes it easy to put together commodity hardware and build a cluster (eg. Beowulf clusters)

Axel is a **NNUS** heterogeneous cluster, meaning that each node in the cluster will contain different types of PEs. Each node on Axel can be used as an independent x86 PC equipped with hardware accelerators (FPGA board and GPUs). The details are shown in Fig. 2.8.

2.5.2 Software

All the nodes in the cluster run a standard Ubuntu Linux (`amd64` architecture). The following software and frameworks are installed to program different hardware components of Axel:

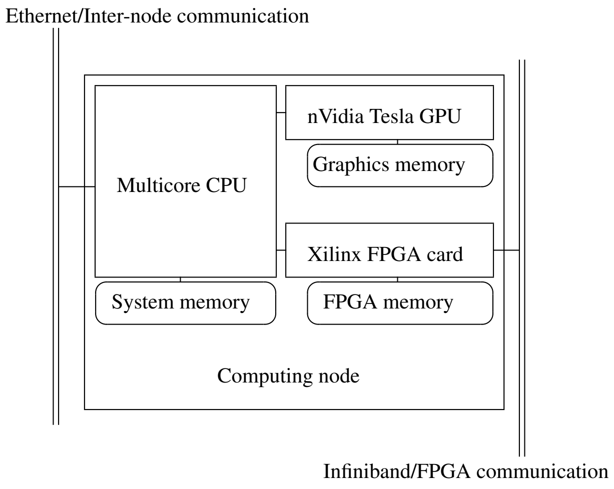


Fig. 2.8: *Axel's NNUS arrangement*

CPU The CPUs are standard multicore x86 CPUs and GCC is used with OpenMPI[4] to produce executables to run on the CPU. The main use of the CPU in a complete Axel application is to coordinate communication between computing nodes using MPI, but it can also be used for general CPU based computation.

nVidia GPU All the GPU used are nVidia Tesla cards, designed for high performance computing rather than general graphics rendering. nVidia provides the *Common Unified Device Architecture* (CUDA) framework for GPU programing. CUDA is is the standard *General-Purpose computing on Graphics Processing Unit* (GPGPU) framework for nVidia products and provides a C-like environment for the Tesla GPU platform [**?**].

FPGA Xilinx ISE 10.1 is used for development of hardware logic for FPGA hardware compilation to the FPGA devices, and all the runtime access to the FPGA devices are done in a very low-level memory mapped I/O and DMA, via a vendor supplied library, exposing an API in a C programming environment.

The compilation and execution of an Axel application is not done in a single executable. The application consists of a CPU-part that initialises the data and distributes to the GPU-part and the FPGA-part of the application. The workload split is described in an XML file for maximum flexibility, which is first read by the CPU-part to segment the data. It is possible to setup the application such that the CPU will do part of the calculation but it is usually used exclusively for I/O coordination and inter-component/inter-node communication. A **map-reduce** framework is used in the cluster for parallel programming where segments of calculations are 'mapped' to the computing elements (eg. FPGA, GPU cores) and the results are 'reduced' and collected by the I/O handling code running on the CPU which, in turn, 'reduces' the results to the master node which started off 'mapping' the input on each computing node.

[4]An implementation of MPI in C, see §2.4.4 for details of MPI

2.5.3 Performance

CPUs are designed for general purpose computation, and because of the underlying von Neuman architecture, the CPU works like an instruction interpreter and follows a fetch-decode-execute cycle to execute stored instructions. It is thus very flexible, but sacrifices the performance in a computation heavy program, where most of the execution time was spent on fetch and decode instead of the more important execute step.

Until recently, GPUs are used solely as graphics rendering hardware. With the increasing demand of modern gaming software and high throughput graphics calculations on display cards, the graphics manufacturers moved from using fixed numbers of dedicated vertex and pixel shaders to unified shaders. This allows a dynamic allocation of graphics hardware for different purposes and higher utilisation of the graphics hardware. GPU becoming less specific to graphics processing gave rise to GPGPU, where GPUs can be used for non-graphics calculations in a limited way. The advantage of GPGPU is the number of graphics processing **cores** available. These cores can do only a small number of tasks at one time but with multiple cores processing is done in parallel. Typical gaming GPUs come with a few hundred processing cores.

FPGAs essentially allow hardware execution of computer programs. As there is no need to interpret or decode the program code and can be executed directly in hardware, compared to the interpreted code model of CPU, there is less wastage of resources. Most important of all, FPGA can be easily reconfigured to carry other tasks unlike immutable dedicated hardware accelerators.

With the reasons stated above, GPUs and FPGAs are much more suited to the use cases of parallel high performance computing. Results of benchmarks [**?**] showed that utilising all the components gives a much better performance than using the components individually, where the magnitude of acceleration is in the descending order of *FPGA*, *GPU* and *CPU*.

On the GPU-only version and the multithreaded CPU version, the execution time of an n-body simulation of 81902 particles in a single time step is 1.1 times and 3.5 times slower than 10-core FPGA version respectively. In the heterogeneous version which uses both GPU and FPGA in a processing node, the workload is load balanced by assigning 2/3 of workload to FPGA and 1/3 to GPU, this gives the overall speedup of 2.1 times over the FPGA-only version.

2.6 Parallel Algorithms

2.6.1 N-body simulation

N-Body simulations are systems to simulate particle movement and interaction due to gravitational forces action on each other.

Each particle in the n-body simulation has a position, vector velocity and mass. In each time step, the positions and velocity of the vectors are recalculated using the velocity and acceleration. The base algorithm is shown in the algorithm below (from [**?**]).

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      if (j != i) {
         rx = p[j].x - p[i].x;
         ry = p[j].y - p[i].y;
         rz = p[j].z - p[i].z;

         dd = rx*rx + ry*ry + rz*rz + EPS;
         d = 1 / sqrtf(dd * dd * dd);

         s = p[j].m * d;

         a[i].x += rx * s;
         a[i].y += ry * s;
         a[i].z += rz * s;
      }
   }
}
```

Listing 2.5: *An algorithm for n-body simulation*

N-body algorithm is highly parallelisable because every particle in the algorithm does not interfere with the content of other particles during the calculation in a time step. The calculation for each article can be done individually in parallel by different computing components.

Current Implementation on Axel On Axel, the implementation is a straightforward translation of the algorithm, by looping over each particles. The *i*-loop is split evenly to different nodes and the *j*-loop is partitioned for FPGA and GPU to compute. The results are then distributed using the `MPI_AllToAll` function to other nodes.

Implementation in SJ In SJ, due to the lack of a multicast-type send, the algorithm cannot be implemented directly. Instead, the n-body algorithm is implemented in 3 parts using a ring topology [**?**]. The `Master` process forwards initial data to a number of `Worker` processes, which is chained together and the last `Worker` process connects to the `Master` process to complete the ring. Since the session type of the first and last worker that communicates directly to the `Master` process, the last `Worker` is slightly different from other `Worker`s, we need to further distinguish them from other worker components. The `Worker` processes will carry out most of the computation.

In each iteration the data is forwarded to each `Worker` through the chain, and adds the results of previous iteration to the data set to be forwarded to the next `Worker` and continues until all nodes have received set of particles from other nodes once. When the data is seen by all `Worker`s, the positions of each particles are updated using the overall velocities and acceleration acting on each particles.

2.7 Summary

In this chapter we have introduced a theoretical framework for modelling structured communications in concurrent systems. Session calculus, a process calculi based on the asynchronous π-calculus and its typing system - session types, forms the basis of a session programming. Session typing ensures that only compatible processes can establish a session and guarantees communication safety.

We then described a full implementation of session-based programming language *Session Java* (SJ), combining Java with sessions.

Next we detailed the target platform for our project, Axel, a cluster with FPGAs and GPUs as acceleration hardware. We also included a comparison of performance between the different components in existing implementations to see a general picture.

Finally, we finished the backgrounds by looking at the differences in parallel programming model of the existing n-body algorithm implemented in Axel and SJ.

Chapter 3

Design and Implementation

In this chapter we will first look at an overall design of parallel applications with SJ on Axel (§3.2), then an implementation of the n-body simulation with SJ on the Axel cluster using CPU and FPGA (§3.3).

Next, we will present a translation of our n-body implementation in SJ to C (§3.4). We will detail the main contribution of the translation, a SJ communication primitives library for C (§3.4.2). The C translation brings session typed SJ programs to C, which is a much more suitable target language and programming environment for high performance computing.

3.1 Design goals

The aim of the project is to develop an approach for designing parallel high performance applications on heterogeneous clusters with session programming. The main criteria we considered were:

Efficiency Existing implementations of parallel algorithms on heterogeneous clusters are very efficient. We aim to keep the performance of our designs as close to current implementations as possible, while getting the advantages of session programming.

(Communication) Safety Session types were designed such that a communication between incompatible sessions will not begin and programs will behave according to its predefined protocol. It is difficult to verify correctness of complex parallel applications design, but session types and their safety property can guarantee the programs are free from incompatible interaction patterns.

Readability Session programming is a high level description of communication. Existing parallel design processes on clusters typically involves using low level libraries and working close to the metal. The instructions to develop for these libraries are verbose and require very explicit instructions (eg. MPI, §2.4.4) for simple tasks, which obfuscates the more important details of process communication. In contrast, the high level SJ abstracts most of the implementation in the runtime system such as the transport medium (TCP vs. UDP vs. shared memory) and puts the focus on communication. Design errors can be identified easily with help of session typing §2.2 and automatic type-checking with SJ §2.4.

However, readability in session programming is a property that comes from structured code and the high level of abstraction and cannot be easily quantified by numerical metrics.

Before we commence our discussion on design and implementation of SJ applications on Axel, we should point out that Axel is an example of NNUS cluster (§2.5.1 contains details of two kinds of cluster arrangement). While our design is targeted to Axel specifically, in theory the design principals can be generalised and applied to NNUS clusters with similar architecture.

3.2 Session Java on Axel

In this section we will discuss the design and the overall architecture for SJ applications to run on Axel.

3.2.1 Overall design

Session Java is an extension of Java. Features of object oriented programming are available in SJ and allow us to create structured and easily reusable code.

The main processing elements on the Axel cluster are CPUs, GPUs and FPGAs. In the initial phase of development, we first implement our choice of algorithm in pure SJ. This allow us to identify any problems with the communication design before involvement of the new hardware, and familiarise with the facilities available on Axel.

Class organisation

By organising classes into suitable packages and class hierarchy, we reduced the efforts needed when implementing for FPGAs or any other acceleration hardware. This is possible because acceleration hardware do not participate in the flow-control of the algorithms. Typically complicated and computation heavy sections of an algorithm are isolated to a single function. The function will then be implemented on acceleration hardware, and no changes to the other parts of the program are required.

FPGA: do one thing and do it well It is uncommon to delegate multiple tasks during a single execution on one piece of acceleration hardware. Suppose two different tasks are implemented on the same piece of hardware, and the two tasks are executed in parallel. Tasks and programs are mapped to the physical hardware on FPGAs. When #1 of the two tasks are being executed, only a portion of the hardware is used - the rest of the hardware will be idle because they are designed to run task #2. Therefore we are not fully utilising the hardware every time the hardware is used.

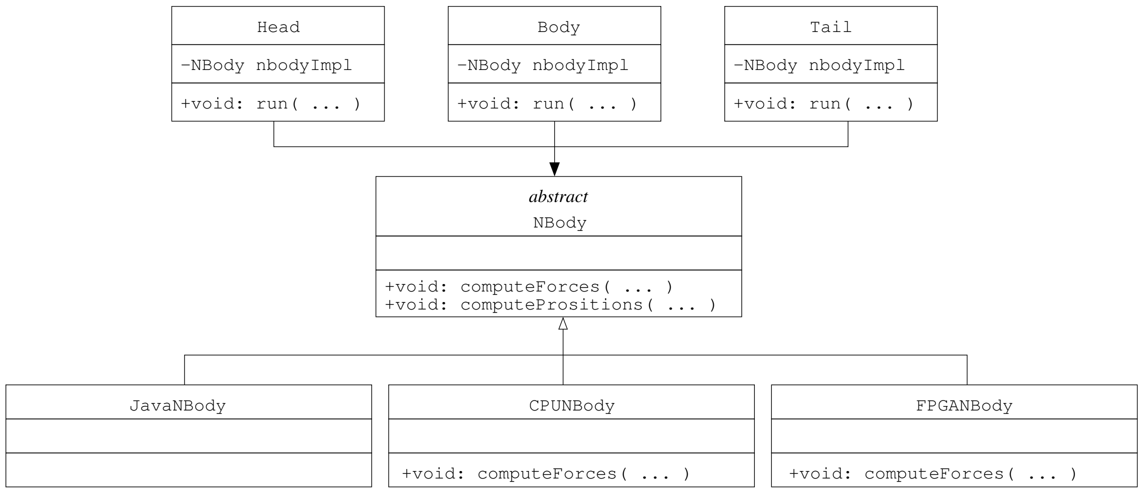


Fig. 3.1: *Simplified UML diagram of our system*

Despite FPGAs are known for the ability to reconfigure at runtime, it is a lengthy process and offers no practical advantage if we need to switch between tasks and reconfigure constantly.

In the class diagram shown above, `NBody` class can be replaced by the main component of other algorithms. This will allow different algorithm to use the same class structure for their implementation, such as an implementation of inner product we will detail in next chapter in §5.2.

`NBody` The abstract class contains all functions used by the algorithm. This class should be replaced by a similar class that contains all core functionalities when implementing other algorithms.

`JavaNBody`, `CPUNBody`, `FPGANBody` These classes are solid implementations of the `NBody` class. The main functions will take the input values and map them on different hardware or software implementations, then return the results as a Java array. The process is transparent to the caller.

`Head`, `Body`, `Tail` classes contain code to set up the topology of the application and SJ communication between nodes of the cluster.
`Head` is the component that runs on the first node and act as the initiator.
`Body` is the worker component and can be chained together with other `Body` nodes.
`Tail` is the last worker component that connects `Body` with `Head` on the other side of the ring. More details on the sessions of the components will be given in the next subsection. §3.2.2

All of the classes take a constructor of class `NBody` to select the implementation to use. This allows combinations such as `FPGANBody`/`Head`, `CPUNBody`/`Body` or `JavaNBody`/`Tail` to be constructed flexibly and easily.

This design uses the well-known design patterns of *strategy* and *template method* [**?**]. The two patterns help separating communication from the algorithm body, and make it easier to reuse

```java
public abstract class NBody {
   public abstract void computeForces( ... );
   ...
}

public class JavaNBody extends NBody {
   public void computeForces( ... ) { ... }
   ...
}

public class Head {
   NBody nbody; // set by constructor or injected
   public void algorithmBody( ... ) {
      ...
      nbody.computeForces( ... );
      ...
   }
}
```

Listing 3.1: *Strategy/template method pattern*

the same algorithm outline and implement it in different hardware. Listing 3.1 outlines how the classes are used.

3.2.2 Application topology and session typing

The implementation of a n-body simulation follows a similar ring topology as described in §2.6.1, the only difference is the addition of initial hardware set-up and tear-down phases at the end of execution.

Session interaction

The structure we are going to describe is not limited to n-body simulations and can be adapted to any algorithm that uses a ring topology. Fig. 3.2 shows the interaction between the node and Table 3.1 gives the session declaration in each of the nodes.

We have covered the meanings of components of SJ `protocols` in §2.4, and we will revisit the declaration shown here in the next chapter (§4.7.1).

Partitioning of data

The input data is uniformly partitioned into n parts, where n is the number of cluster nodes in the simulation at execution time. Each node is responsible for outputting the particle positions of its allocated set of particles, and will keep track of their velocities and acceleration components at each steps of calculation.

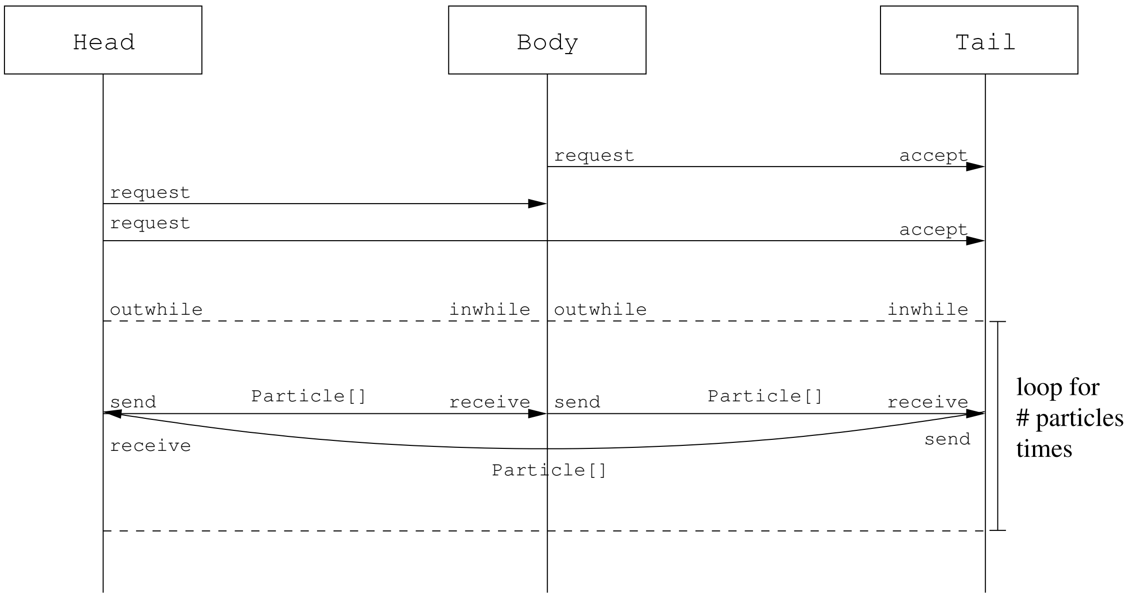


Fig. 3.2: *Interaction between `Head`, `Tail` and a single `Body` node*

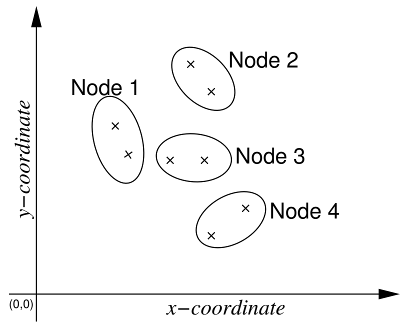
Node	Session between		SJ session declaration (`protocol`)
`Head`	`Head`	`Tail`	`cbegin.![?(Particle[])]*`
`Head`	`Head`	`Body`	`cbegin.?(int)![!<Particle[]>]*`
`Body`	`Body` $_{i-1}$	`Body` $_i$	`sbegin.!<int>.?[?(Particle[])]*`
`Body`	`Body` $_i$	`Body` $_{i+1}$	`cbegin.?(int).![![Particle[]]]*`
`Tail`	`Body`	`Tail`	`sbegin.!<int>.?[?(Particle[])]*`
`Tail`	`Head`	`Tail`	`sbegin.?[!<Particle[]>]*`

Table 3.1: *SJ session declaration for ring topology*

The numbers in Fig. 3.3 represent the node number which the particles are from. In the *initial* round, velocities and acceleration components of each particle are calculated against each other in the same node.

Next, each node forwards the initial set (or the received set after the initial round) of particle positions to the adjacent node. Since velocities and acceleration components can be accumulated, when a set of particles is received, each node can immediately update the velocities and acceleration of their **own** set of particles without keeping a copy of the received particles.

After 3 rounds, all nodes will have seen all the particles and can perform calculations to update the positions of the particles they were allocated. With n nodes participating in the simulation, the calculate-and-forward step is repeated for $n-1$ steps instead of 3 in our example above in order for all nodes to see all particles at once.



Node/partition #	Initial	Round 1	Round 2	Round 3
Node 1 (`Head`)	{1}	{1,4}	{1,4,3}	{1,4,3,2}
Node 2 (`Body`)	{2}	{2,1}	{2,1,4}	{2,1,4,3}
Node 3 (`Body`)	{3}	{3,2}	{3,2,1}	{3,2,1,4}
Node 4 (`Tail`)	{4}	{4,3}	{4,3,2}	{4,3,2,1}

Fig. 3.3: *Partitioning of data and calculation for 4 nodes*

3.3 SJ with FPGA

After looking at the general architecture of SJ parallel applications on Axel, in this section we will detail an implementation of the n-body simulation using FPGA.

3.3.1 The need for cross-language features

As we have described in the introduction of the Axel cluster §2.5.2, the development environment for all of the heterogeneous components of Axel is C. However SJ is based on Java, and cannot use executables or shared libraries in C (or other *native* code) directly.

Java is an interpreted programming language that runs in the *Java Virtual Machine* (JVM). By design, the underlying architecture of the hardware is abstracted by the JVM completely, making Java a very portable language but it is not possible to access the memory or execute native instructions directly.

We also have the following considerations in programming language choice:

- Java is not as fast as native compiled languages such as C/C++ in most scenarios [**?**].
- SJ is a communication based language, the most important feature are its *communication capabilities* and *safety properties*.
- FPGAs and GPUs can be accessed by libraries supplied by vendor, but only C APIs and C-based development environments are available.

Combining the best points of SJ and acceleration hardware, we should delegate all communication and I/O coordination to SJ and all computation-related tasks to acceleration hardware for its performance.

In this design, SJ/Java will have to inter-operate with native libraries to access and control the acceleration hardware.

Alternative 1 It is certainly possible to take an extreme and approach this problem by re-implementing SJ and create a new session-types based language in C/C++. Despite the performance advantage, this will be a lot more involved than extending Java to SJ. C/C++ does not come with the rich set of readily usable network and datastructure libraries found in Java and upon which SJ depends heavily on. For the purpose of high performance computing, this would be the best option. In the next section we will present a C-translation of SJ built around the concept of a session-type based C/C++ programming language.

Alternative 2 The other end of the extreme is to translate all hardware drivers to Java so SJ can initiate computations from within the JVM. In the case of GPU, we will be translating the complete CUDA framework to Java. Some current unofficial implementations exists [**?**, **?**]; and for FPGA, none of the vendors provide Java APIs for control and access. Moreover, ways of accessing system memory are very limited in Java. The design of Java is to *prevent* this mode of operation to decouple applications running in the JVM from the underlying hardware. This method is not easily generalisable.

Alternative 3 With the reasons above combined, the remaining option is to mix Java and C using a suitable bridging library. This way SJ can be used in its natural form, and hardware drivers and shared library code written in C can be used as it is designed. The bridging library should handle type conversions and data access between the two sides.

3.3.2 Java Native Interface

With *Java Native Interface* (JNI), applications written in languages other than Java can be accessed by Java using an interface understandable by the JVM. The user will write native code and export selected native subroutines via JNI. This is the standard native programming environment supported by the Java specification, and provides very fine grain control for data shared between the two sides. An example is given in the appendix §A.1.

3.3.3 Java Native Access

Java Native Access (JNA) [**?**] is an API built on top of JNI and takes on Java-C bridging in a very different approach. It does not require any boilerplate code in the native language, and therefore can use existing native libraries *without modifications* at the cost of performance.

JNA uses `libffi` to analyse the structure of the shared library at runtime. FFI stands for *Foreign Function Interface*, the purpose of FFI is to convert calling conventions and coordinate between programming languages [**?**]. JNA also came with a basic type-mapping infrastructure to allow data exchanged between the two sides. In addition to primitive type mappings, the library also maps `Structure` class to C-struct, and Java arrays (non-contiguous in memory) to C arrays (contiguous). An example is given in the appendix §A.2.

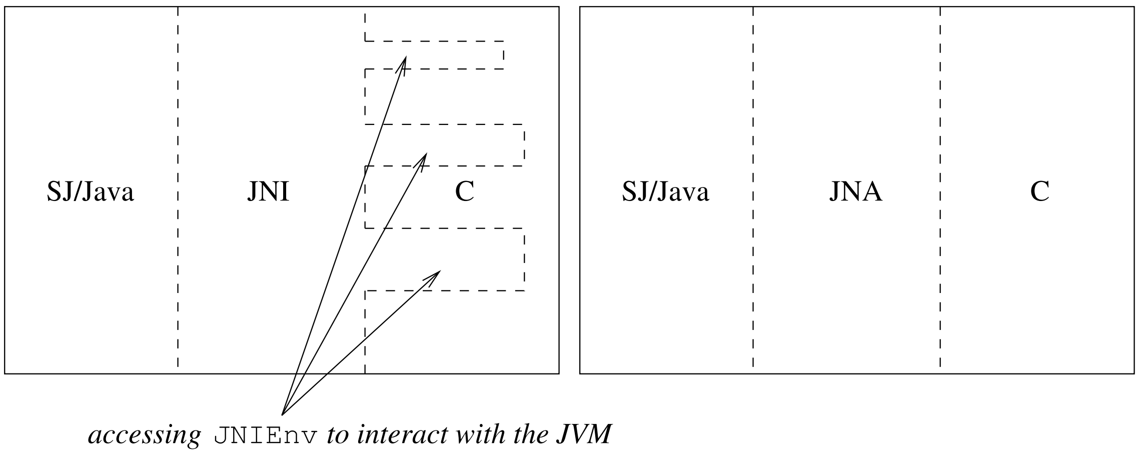


Fig. 3.4: *Platform integration using JNI and JNA. JNA has a much cleaner interface than JNI*

The above short introduction we showed that JNA is more flexible than JNI, where JNI forces a very tight integration between JVM and native platform.

For our implementation, we have chosen JNA over JNI because the bridging code has very little to do with session programming. Using JNA allows much more rapid prototyping to check correctness of communication. In JNA, we could simply load a different hardware driver if the underlying hardware is changed, as long as the interface of the compiled code remains the same.

In the next section §3.4 we will see a version of the implementation translated from SJ to C, which takes advantage of the shared library being an isolated component from Java.

3.3.4 Problems encountered

During the implementation, we encountered some problems and limitations

Java send buffer Java TCP sockets, which is what SJ uses when running on the cluster, is non-blocking. The sockets are made non-blocking by queueing the values to send in a send buffer. However, when the buffer is full, the semantics of Java TCP sockets became blocking. While there is a way to change the send buffer size, the send buffer size has a hard upper limit of 131071 bytes. Setting the send buffer size above this value will not change the actual buffer size. This caused some problems when trying to benchmark the performance of any SJ implementations using a very high number of particles.

This problem was addressed recently (and indirectly), by a new TCP socket implementation that uses custom TCP send and receive queues for events-based session programming [**?**].

JNA data conversion In C and most native languages, arrays are represented by contiguous blocks of main memory. In Java, array elements can be distributed all over the Java mem-

```
for (i = 0; i < particlesPerNode; i++) {
    for (j = 0; j < particlesPerNode; j++) {

        ri = receivedParticles[j].x - particles[i].x;
        rj = receivedParticles[j].y - particles[i].y;
        m  = receivedParticles[j].m

        if (ri != 0) {
            ai += (ri < 0 ? -1 : 1) * G * m / (ri * ri);
        }

        if (rj != 0) {
            aj += (rj < 0 ? -1 : 1) * G * m / (rj * rj);
        }
    }

    particleVelocities[i].ai += ai;
    particleVelocities[i].aj += aj;
}
```

Listing 3.2: *algorithm for SJ implementation's n-body simulation*

ory space for efficient management of free space. JNA allows force creation of contiguous blocks of memory in Java, only then the piece of memory can be passed to a native function.

Our implementation of n-body simulation receives an array of `Particles` in each iteration and passes the array to a native function to process. Since the array is received *as a Java object*, the memory is not contiguous. In order to pass the array to the native functions, the array needs to be copied to another contiguous-memory array in every iteration. This overhead could not be eliminated.

3.4 C-translation of SJ n-body implementation

This section will present a translation of the described in the previous section (§3.3). The motivation behind this translation is **performance**. When using a bridging library between two languages with completely disjoint language runtime, there is *always* overhead associated with the conversion between data formats. Also, we have discussed in previous sections (§3.3) that acceleration hardware libraries are provided in a C programming environment, it would be much more convenient if we could use session programming in a C environment. However, we also noted that the standard C do not have a comprehensive networking and datastructure library as in Java, and difficulties of building a completely new programming language based on C similar to how SJ is built on top of Java.

Our proposed solution is a C library that provide networking primitives available in SJ to C. The primitives made available to the programmer should have the same or very similar semantics as their counterpart in SJ. For example, SJ's `send` primitive is non-blocking. The library should similarly implement a non-blocking `send` in the library. This library would be a step forward towards a session-types based C programming language we discussed in previous section (§3.3.1).

It should be reminded that the library is not designed to be used directly. The purpose of the library is to provide a building block of SJ programs translated to C, usage of the library does not automatically imply communication correctness if not translated from SJ.

3.4.1 Why would this work?

C/C++ shares a very similar language syntax. For basic language constructs and flow-control, conversion between the two languages are trivial. When the source SJ program uses a SJ primitive, the target C program will use one of the primitives provided in our library. This forms a backbone of our translated C program, as translation will be line by line, the sequence of invoking the SJ primitives will be identical in the translated version.

3.4.2 A SJ primitives library for C

Table 3.2 compares the equivalent primitives in the two languages:

C	SJ
`server_socket(port)`	`SJServerSocketImpl.create(p, port)`
`client_socket(host_addr, port)`	`SJService.create(p, host, port)`
`accept_connection(node)`	`node.accept()`
Included in `client_socket()`	`node.request()`
`send_int(node, value)`[1]	`node.sendInt(value)`
`inwhile(nodes[], nodes_count)`	`<node1,node2,..>.inwhile{}`
`outwhile(cond, node[], nodes_count)`	`<node1,node2,..>.outwhile(cond) {}`

Table 3.2: *C and SJ session primitives*

In the section where we introduced JNA (§3.3.3), we have briefly discussed that *shared library*[2] to access FPGA can be reused without modifications. In most use cases, the shared library will provide a single function that takes input data and forward them to the FPGA. For n-body simulation, the main function in the shared library is `compute_forces( ... )` shown in Fig. 3.5.

The translation do not need to worry about any new interface to access the acceleration hardware.

The syntax for **`inwhile`** and **`outwhile`** above shows slightly different syntax between the two languages. C-version of both constructs are typically used in conjunction with a while loop. The

[1] `send` is implemented for all primitive types

[2] shared library refers to the native code component in the SJ/FPGA implementation, compiled as a shared object (`.so`)

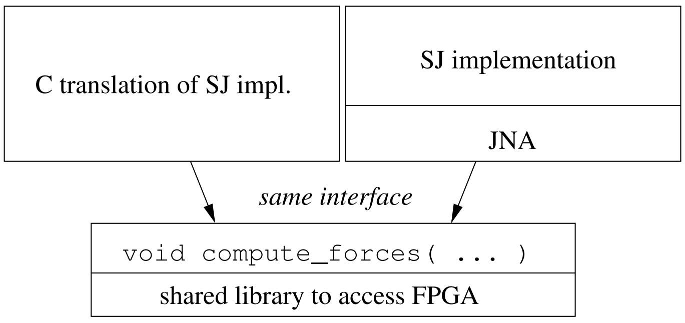

C translation of SJ impl.
SJ implementation
JNA
same interface
void compute_forces(...)
shared library to access FPGA

Fig. 3.5: *Shared library used from both SJ/JNA and C*

reason for the difference is because as a library external to the programming language, we are unable to modify the syntax of the language without going into parser of the compiler. We can, however, use C preprocessor macros to use a more familiar syntax. Listing 3.3 shows the usages of **`inwhile`** and **`outwhile`** in C.

```c
#define OUTWHILE( COND, MOEP, NR_OF_MOEP ) \
        while( outwhile( (COND), (MOEP), (NR_OF_MOEP) ) )
#define INWHILE( MOEP, NR_OF_MOEP ) \
        while( inwhile( (MOEP), (NR_OF_MOEP) ) )
...

outwhile_sfds[0] = next_fd;
outwhile_sfds[1] = tail_fd;
loop_index = 0;
while ( inwhile( loop_index < iter_count, outwhile_sfds, 2) ) { ... }
// OUTWHILE( loop_index < iter_count, outwhile_sfds, 2) { ... }

inwhile_sfds[0] = prev_fd;
inwhile_sfds[1] = head_fd;
while ( inwhile(inwhile_sfds, 2) ) { ... }
// INWHILE( inwhile_sfds, 2 ) { ... }
```

Listing 3.3: *C **`inwhile`** and **`outwhile`***

In addition to the control flow and communication primitives, the library also has support for error handling in form of UNIX signal handlers. The C language do not have try-catch or similar system for exceptions. In SJ, runtime errors in sessions such as network error will **`throw`** an exception and all communicating processes will receive a `SJFIN` exception and terminate immediately. The library keeps track of all active connections in each process. If any of the connections encountered a problem, the process will force close all the active connections from its side. The other ends of the connections will then receive a `SIGPIPE` signal and promptly close all active connections. The signal is then propagate to all other connected processes until all processes are terminated.

```c
 #include <signal.h>
 #include "sighandlers.h"
...
int main(int argc, char *argv[])
{
  signal(SIGPIPE, &sigpipe_handler);
  signal(SIGSEGV, &sigsegv_handler);
  ...
}
```

Listing 3.4: *Error handling in translated C*

To use the signal handlers provided, users only need to set the signal handlers to those provided by the library (header `sighandlers.h`) as shown in Listing 3.4.

Another consideration when designing the library is to make the translated C version have the same structure as the SJ version. For example, socket options when creating TCP sockets are completely hidden from the user. The options exposed to the user are the same as in Java version. Listing A.7 and A.8 shows the outline of the two versions without variable declarations and other unimportant details. The SJ implementation can almost map to the C-translation line by line.

3.4.3 Shortcomings of the library

The library is not a complete implementation of the full SJ. Session delegation and higher-order session manipulation is not possible with our library. The main reason is we lack a representation of *sessions*, which we omitted when we design our library for *translating* a session-based language to a non session-based language, rather than to add sessions to C.

3.5 Summary

In this chapter, we have discussed the details of our n-body implementation in SJ on the NNUS heterogeneous cluster Axel.

We looked at both the application architecture such as class layout and the session interaction pattern between the nodes in a ring-topology. We also looked at rationale to use a cross-language library *Java Native Access* (JNA), with a short introduction to the usage.

Finally we showed a manual translation of the SJ program to C, and a library that provides SJ primitives to C. The library contains a collection of light weight SJ primitives that mimics the SJ implementation, and can be a building block for an automatic SJ-to-C translator.

Chapter 4

Correctness Proof of N-body Implementation

In this chapter we will look at the formalisation of multichannel `inwhile` and `outwhile` primitives in SJ. The new primitives are designed for programming parallel algorithms in SJ, and had not been formalised in session calculus previously.

With the new formalisation, we will prove that our implementation of n-body simulation in SJ is *communication safe* from a global view and generalise the proof to algorithms with similar design and topology as our implementation.

We will first present an updated session calculus (§4.1) to include the new SJ primitives, where will will look at the syntax (§4.2), followed by the operational semantics (§4.3) and the type system (§4.4). Finally, we will look at *subject reduction* (§4.5) which will be used to prove communication safety property (§4.6)

4.1 Session calculus with multichannel in/outwhile

We will now present an extension of the session calculus to include the multichannel `inwhile` and `outwhile` SJ primitives used in parallel algorithms design with SJ. The original `inwhile` and `outwhile` primitives described in [**?**, **?**] only operates in a single session channel. Synchronisation of `outwhile` loop condition between multiple session channels are not possible without re-opening the last session in each iteration of `outwhile` loop.

The multichannel constructs are the key components of parallel design with SJ, and parallel topologies can be expressed more naturally. Fig. 4.1 shows how the constructs improve ring topology design, first introduced in [**?**], and is used in our n-body implementation.

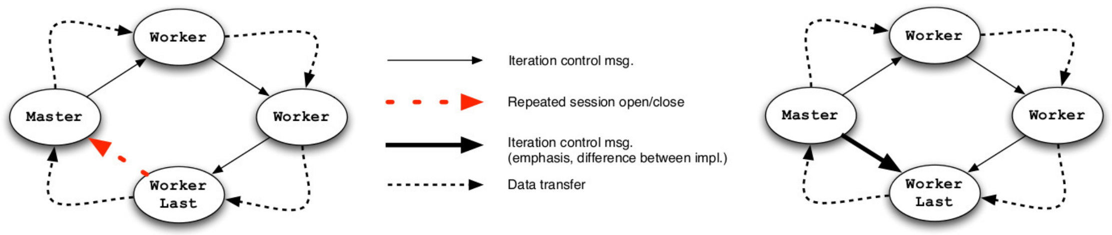


Fig. 4.1: *Comparison of ring topology in unichannel and multichannel* **`inwhile`** *and* **`outwhile`**

4.2 Syntax

The syntax of the updated session calculus with multichannel **`inwhile`** and **`outwhile`** is shown in Fig. 4.2.

The process definition is modified to include an `Err` process which represents a *while condition mismatch* in an `inwhile`/`outwhile` composition. *while condition mismatch* is further explained in the operational semantics §4.3.

Single channel `inwhile` and `outwhile` is sometimes written in the calculus as k.`inwhile`$\{\ Q\ \}$ and k.`outwhile`$(e)\{\ P\ \}$. This syntax is a shorthand for $\langle k\rangle$.`inwhile`$\{\ Q\ \}$ and $\langle k\rangle$.`outwhile`$(e)\{\ P\ \}$.

4.3 Operational semantics

The operational semantics are based on the reduction relation $\rightarrow$, and the reduction rules are given in Fig. 4.5. The session calculus is π-calculus extended with session primitives [**?**], so definition of structural congruence $\equiv$ is similar to π-calculus. Fig. 4.3 lists the structural congruence rules in our updated session calculus. An additional structural congruence rule in this calculus is $\mathbf{0}; P \equiv P$, which allows continuation in sequential composition (Definition 1 below).

To keep session reasoning simple, we introduce evaluation contexts. Evaluation contexts isolate subprocesses and allow subprocesses to reduce independent of influences external to the context. Our evaluation contexts are defined as:

$$E[] := [] \mid E[];P \mid E[] \mid P \mid (\nu u)\, E[] \mid \mathsf{def}\ D\ \mathsf{in}\ E[]$$

If the *head subprocess* P in $E[P]$ can be reduced using the reduction rules, then there is a dual *head subprocess* Q in $E[Q]$ that can be reduced [**?**]. This simplifies the reduction rules and allows us to avoid including explicit reduction rules such as sequential composition $(P;Q)$. The resulting reduction rules are shown in Fig. 4.5.

We have defined reduction rules for `inwhile` and `outwhile` such that they can reduce on their own. In particular, a single `outwhile` can generate an infinite number of $k \dagger [b]$ without constraints. Suppose the loop condition is `true` in the first run of the `outwhile`. Since `outwhile` can reduce without constraints, `outwhile` can reduce again, and the loop condition in this iteration is `false`.

P	$::= \mathtt{0}$		inaction
	$\mid T$		prefixed process
	$\mid P\ ;\ Q$		sequential composition
	$\mid P \mid Q$		parallel composition
	$\mid (\nu u)\ P$		name/channel hiding
	$\mid \mathtt{Err}$		error
T	$::= \mathtt{request}\ a(k)\ \mathtt{in}\ P$		session request
	$\mid \mathtt{accept}\ a(k)\ \mathtt{in}\ P$		session acceptance
	$\mid k![\tilde{e}]$		data sending
	$\mid k?(\tilde{x})\ \mathtt{in}\ P$		data reception
	$\mid k \lhd l$		label selection
	$\mid k \rhd \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\}$		label branching
	$\mid \mathtt{throw}\ k[k']$		channel sending
	$\mid \mathtt{catch}\ k(k')\ \mathtt{in}\ P$		channel reception
	$\mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$		conditional branch
	$\mid X[\tilde{e}\tilde{k}]$		process variables
	$\mid \mathtt{def}\ D\ \mathtt{in}\ P$		recursion
	$\mid \langle k_1 \dots\ k_n \rangle.\mathtt{inwhile}\{\ Q\ \}$	$n \geq 1$	multichannel inwhile
	$\mid \langle k_1 \dots\ k_n \rangle.\mathtt{outwhile}(e)\{\ P\ \}$	$n \geq 1$	multichannel outwhile
	$\mid k \dagger [b]$	$(b \in \mathtt{true}, \mathtt{false})$	(runtime syntax)
e	$::= c$		constant
	$\mid \langle k_1 \dots k_n \rangle.\mathtt{inwhile}$	$n \geq 1$	inwhile expression
	$\mid e + e' \mid e - e' \mid e \times e \mid \mathtt{not}(e) \mid \dots$		operators
D	$::= X_1(\tilde{x}_1\tilde{k}_1) = P_1\ \mathtt{and} \cdots \mathtt{and}\ X_n(\tilde{x}_n\tilde{k}_n) = P_n$		declaration for recursion

Fig. 4.2: *Session calculus syntax with multichannel* `inwhile` *and* `outwhile`

This gives us $k.\mathtt{outwhile}(e)\{\ P\ \} \mid k \dagger [\mathtt{true}] \mid k \dagger [\mathtt{false}] \mid k.\mathtt{inwhile}\{\ Q\ \}$ which causes us problems because we do not know which $k \dagger [b]$ to compose with `inwhile`.

This differs from our implementation where the while loop synchronises nodes and delivers loop conditions in order. To correctly reflect the actual behaviour of multichannel `inwhile` and `outwhile` constructs in the calculus, we have an extra constraint that **`inwhile` rules have precedence over `outwhile`** . This way, loop conditions holders, $k \dagger [b]$ will prevent the sessions from continuing without first consuming $k \dagger [b]$ with a matching `inwhile`.

$$
\begin{gathered}
P \equiv Q \text{ if } P \equiv_\alpha Q \\
P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
(\nu u)\ P \mid Q \equiv (\nu u)\ (P \mid Q) \quad \text{if } u \notin fu(()Q) \\
(\nu u)\ \mathbf{0} \equiv \mathbf{0} \\
\texttt{def}\ D\ \texttt{in}\ \mathbf{0} \equiv \mathbf{0} \\
(\nu u)\ \texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D\ \texttt{in}\ (\nu u)\ P \quad \text{if } u \notin fu(()D) \\
(\texttt{def}\ D\ \texttt{in}\ P) \mid Q \equiv \texttt{def}\ D\ \texttt{in}\ (P \mid Q) \quad \text{if } dpv(()D) \cap fpv(()Q) = \emptyset \\
\texttt{def}\ D\ \texttt{in}\ (\texttt{def}\ D'\ \texttt{in}\ P) \equiv\ \texttt{def}\ D\ \texttt{and}\ D'\ \texttt{in}\ P \quad \text{if } dpv(()D) \cap dpv(()D') = \emptyset. \\
\mathbf{0};\ P\ \equiv P
\end{gathered}
$$

Fig. 4.3: *Structural Congruence*

$$
E[] := [] \mid E[];P \mid E[] \mid P \mid (\nu u)\ E[] \mid \mathsf{def}\ D\ \mathsf{in}\ E[]
$$

Fig. 4.4: *Evaluation context*

4.4 Type system

The type system in this section is designed to guarantee communication safety and progress property with the new syntax and operational semantics. The full type syntax is given in Fig. 4.6.

Sorts contain the standard types and the pair of dual sessions $\langle \alpha, \overline{\alpha} \rangle$.

Partial session types are session types that does not include the end type. Partial session types are distinguished from completed session types so that they can be sequentially composed.

Completed session types are types that end with end or are equal to $\perp$.

In above syntax, $![\alpha]$ and $?[\alpha]$ are session delegation and session receive respectively. This makes use of the name-passing property from π-calculus that allows sending and receiving of channels (or *sessions* in the session calculus). The same typing syntax is used for ordinary type sending and receiving ($![\tilde{S}]$, $?[\tilde{S}]$). Iteration types ($?[\tau]^*$ and $![\tau]^*$) are introduced for `inwhile` and `outwhile` respectively. With iteration types, the partial type definition τ can be repeated for a number of times until the `outwhile` condition is no longer `true`.

In the syntax given, $\&\{l_1 \colon \tau_1, \dots, l_n \colon \tau_n\}.\mathsf{end} \equiv \&\{l_1 :: \tau_1.\mathsf{end}, \dots, l_n \colon \tau_n.\mathsf{end}\}$. This equivalence ensures all partial types $\tau_1 \dots \tau_n$ of label selection choices 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 $\tau;\varepsilon \equiv \tau$. The two equivalences allows us to continue reducing when one of the two processes $P;Q$ reduces to empty.

$$E_1[\texttt{accept}\ a(k)\ \texttt{in}\ P_1] \mid E_2[\texttt{request}\ a(k)\ \texttt{in}\ P_2] \ \rightarrow\ (E_1[P_1] \mid E_2[P_2]) \qquad (k \text{ is fresh}) \quad [\textsc{Link}]$$

$$E_1[k![\tilde{e}]] \mid E_2[k?(\tilde{x})\ \texttt{in}\ P_2] \ \rightarrow\ E_1[\mathbf{0}] \mid E_2[P_2[\tilde{c}/\tilde{x}]] \qquad (\tilde{e} \downarrow \tilde{c}) \quad [\textsc{Com}]$$

$$E_1[k \lhd l_i; P] \mid E_2[k \rhd \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\}] \ \rightarrow\ E_1[P] \mid E_2[P_i] \qquad (1 \le i \le n) \quad [\textsc{Label}]$$

$$E_1[\texttt{throw}\ k[k']] \mid E_2[\texttt{catch}\ k(k')\ \texttt{in}\ P_2] \ \rightarrow\ E_1[\mathbf{0}] \mid\ E_2[P_2] \quad [\textsc{Pass}]$$

$$E[\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q] \ \rightarrow\ E[P] \qquad (e \downarrow \texttt{true}) \quad [\textsc{If}1]$$

$$E[\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q] \ \rightarrow\ E[Q] \qquad (e \downarrow \texttt{false}) \quad [\textsc{If}2]$$

$$\texttt{def}\ D\ \texttt{in}\ (E[X[\tilde{e}\tilde{k}]]) \ \rightarrow\ \texttt{def}\ D\ \texttt{in}\ (E[P[\tilde{c}/\tilde{x}]]) \qquad (\tilde{e} \downarrow \tilde{c}, X(\tilde{x}\tilde{k}) = P \in D) \quad [\textsc{Def}]$$

$$E[\langle k_1 \ldots\ k_n\rangle.\texttt{inwhile}\{\ P\ \}] \mid k_1 \dagger [b_1] \mid \ldots \mid k_n \dagger [b_n] \ \rightarrow\ E[P; \langle k_1 \ldots\ k_n\rangle.\texttt{inwhile}\{\ P\ \}]$$
$$(\forall i \in 1..n, b_i = \texttt{true}) \quad [\textsc{Inwhi}1]$$

$$E[\langle k_1 \ldots\ k_n\rangle.\texttt{inwhile}\{\ P\ \}] \mid k_1 \dagger [b_1] \mid \ldots \mid k_n \dagger [b_n] \ \rightarrow\ E[\mathbf{0}]$$
$$(\forall i \in 1..n, b_i = \texttt{false}) \quad [\textsc{Inwhi}2]$$

$$E[\langle k_1 \ldots\ k_n\rangle.\texttt{inwhile}\{\ P\ \}] \mid k_1 \dagger [b_1] \mid \ldots \mid k_n \dagger [b_n] \ \rightarrow\ E[Err]$$
$$(\exists\ i,j\ b_i = \texttt{true} \wedge\ b_j = \texttt{false} \wedge 1 \le i,j \le n) \quad [\textsc{Inwhi}3]$$

$$E[\langle k_1 \ldots\ k_n\rangle.\texttt{outwhile}(e)\{\ P\ \}] \ \rightarrow\ E[P; \langle k_1 \ldots\ k_n\rangle.\texttt{outwhile}(e')\{\ P\ \}] \mid k_1 \dagger [b_1] \mid \ldots \mid k_n \dagger [b_n]$$
$$(\forall i \in 1..n, b_i = \texttt{true}) \qquad (E[e] \ \rightarrow\ E[\texttt{true}]) \quad [\textsc{Outwhi}1]$$

$$E[\langle k_1 \ldots\ k_n\rangle.\texttt{outwhile}(e)\{\ P\ \}] \ \rightarrow\ E[\mathbf{0}] \mid k_1 \dagger [b_1] \mid \ldots \mid k_n \dagger [b_n]$$
$$(\forall i \in 1..n, b_i = \texttt{false}) \qquad (E[e] \ \rightarrow\ E[\texttt{false}]) \quad [\textsc{Outwhi}2]$$

$$P \equiv P' \text{ and } P' \ \rightarrow\ Q' \text{ and } Q' \equiv Q \ \Rightarrow\ P \ \rightarrow\ Q \quad [\textsc{Str}]$$

$$P \ \rightarrow\ P' \ \Rightarrow\ E[P] \ \rightarrow\ E[P'] \quad [\textsc{Eval}]$$

Fig. 4.5: *Reduction rules*

4.4.1 Duality

To ensure communication compatibility, all session types have a dual-type in a well-typed program.

A simple example is $![\mathsf{bool}].\mathsf{end}$ and $?[\mathsf{bool}].\mathsf{end}$. The two session types are dual so that sending of a bool matches with receiving of a bool. If the typing of the receiver is changed to $?[\mathsf{bool}];?[\mathsf{bool}].\mathsf{end}$ then there is a communication mismatch after the first receive. Session types can ensure such incompatibilities between two communicating parties does not happen. Fig. 4.7 is complete list of dual-types in our type system.

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

Fig. 4.6: *Type syntax*

$$
\begin{array}{ccc}
\overline{\varepsilon} = \varepsilon & \overline{\tau;\ \tau} = \overline{\tau};\ \overline{\tau} & \overline{\alpha^\dagger} = \overline{\alpha}^\dagger \\
\overline{![\tilde{S}]} = ?[\tilde{S}] & \overline{\oplus\{l_1 : \tau_1, \dots, l_n : \tau_n\}} = \&\{l_i : \overline{\tau_i} \dots, l_n : \overline{\tau_n}\} & \overline{![\tau]} = ?[\tau] \\
\overline{?[\tilde{S}]} = ![\tilde{S}] & \overline{\&\{l_1 : \tau_1, \dots, l_n : \tau_n\}} = \oplus\{l_i : \overline{\tau_i} \dots, l_n : \overline{\tau_n}\} & \overline{?[\tau]} = ![\tau] \\
\overline{![\tau]^*} = ?[\overline{\tau}]^* & \overline{\mathbf{x}} = \mathbf{x} & \overline{\tau.\mathsf{end}} = \overline{\tau}.\mathsf{end} \\
\overline{?[\tau]^*} = ![\overline{\tau}]^* & \overline{\mu\mathbf{x}.\tau} = \mu\mathbf{x}.\overline{\tau} & \overline{\bot} = \bot
\end{array}
$$

Fig. 4.7: *Dual types*

4.4.2 Typing environment

The typing environment is defined in Fig. 4.8.

$$
\begin{array}{l}
\Gamma \ ::= \ \emptyset \mid \Gamma \cdot x : S \mid \Gamma \cdot X : \tilde{S}\tilde{\alpha} \\
\Delta \ ::= \ \emptyset \mid \Delta \cdot k : \alpha \mid \Delta \cdot k : \dagger
\end{array}
$$

Fig. 4.8: *Typing environments*

Γ is the *standard environment* that maps variables to sort types.

Δ is the *runtime environment* that contains session to session type mappings and the typing for $k \dagger [b]$, which holds `inwhile` and `outwhile` loop conditions.

4.4.3 Typing rules

Most of the typing rules remained the same as in [**?**] The major changes between the two version are

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

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : \varepsilon.\mathsf{end}}{\Gamma \vdash P \triangleright \bot} \qquad \frac{}{\Gamma \cdot a : S \vdash a \triangleright S} \qquad \frac{\Gamma;\, \Delta \vdash e \triangleright S}{\Gamma;\, \Delta, \Delta' \vdash e \triangleright S} \qquad [\textsc{Bot}],[\textsc{NameI}],[\textsc{Eval}]$$

$$\frac{\Delta = \{k_1 : \varepsilon.\mathsf{end}, \dots, k_n : \varepsilon.\mathsf{end},\ k'_1 : \bot, \dots, k'_m : \bot\}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad [\textsc{Inact}]$$

$$\frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle, \Gamma \vdash P \triangleright \Delta \cdot k : \overline{\alpha}}{\Gamma \vdash \mathtt{request}\ a(k)\ \mathtt{in}\ P \triangleright \Delta} \qquad \frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle, \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Gamma \vdash \mathtt{accept}\ a(k)\ \mathtt{in}\ P \triangleright \Delta} \qquad [\textsc{Req}],[\textsc{Acc}]$$

$$\frac{\Gamma;\, \emptyset \vdash \tilde{e} \triangleright \tilde{S}}{\Gamma \vdash k![e] \triangleright \Delta \cdot k : ![\tilde{S}].\mathsf{end}} \qquad \frac{\Gamma \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \Delta \cdot k : \alpha}{\Gamma \vdash k?(\tilde{x})\ \mathtt{in}\ P \triangleright \Delta \cdot k : ?[\tilde{S}];\, \alpha} \qquad [\textsc{Send}],[\textsc{Rcv}]$$

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

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : \tau_j.\mathsf{end}}{\Gamma \vdash k \lhd l \triangleright \Delta \cdot k : \oplus\{l_1 : \tau_1, \dots, l_n : \tau_n\}.\mathsf{end}} \quad (1 \le j \le n) \qquad [\textsc{Sel}]$$

$$\frac{}{\Gamma \vdash \mathtt{throw}\ k[k'] \triangleright \Delta \cdot k : ![\alpha].\mathsf{end} \cdot k' : \alpha} \qquad [\textsc{Thr}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : \beta \cdot k' : \alpha}{\Gamma \vdash \mathtt{catch}\ k(k')\ \mathtt{in}\ P \triangleright \Delta \cdot k : ?[\alpha];\, \beta} \qquad [\textsc{Cat}]$$

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

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

$$\frac{\Gamma;\, \Delta \vdash Q \triangleright \Delta \cdot k_1 : \tau_1.\mathsf{end} \cdot \cdots \cdot k_n : \tau_n.\mathsf{end}}{\Gamma \vdash \langle k_1 \dots\ k_n \rangle.\mathtt{inwhile}\{\ Q\ \} \triangleright \Delta \cdot k_1 : ?[\tau_1]^*.\mathsf{end} \cdot \cdots \cdot k_n : ?[\tau_n]^*.\mathsf{end}} \qquad [\textsc{InWhi}]$$

$$\frac{\Gamma \cdot a : S \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)\ P \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k : \bot}{\Gamma \vdash\ (\nu k)\ P \triangleright \Delta} \qquad [\textsc{NRes}],[\textsc{CRes}]$$

$$\frac{\Gamma;\, \emptyset \vdash \tilde{e} \triangleright \tilde{S}}{\Gamma \cdot X : \tilde{S}\tilde{\alpha} \vdash X[\tilde{e}\tilde{k}] \triangleright \Delta \cdot \tilde{k} : \tilde{\alpha}} \qquad [\textsc{Var}]$$

$$\frac{\Gamma \cdot X : \tilde{S}\tilde{\alpha} \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \tilde{k} : \tilde{\alpha} \qquad \Gamma \cdot X : \tilde{S}\tilde{\tau} \vdash Q \triangleright \Delta}{\Gamma \vdash \mathtt{def}\ X(\tilde{x}\tilde{k}) = P\ \mathtt{in}\ Q \triangleright \Delta} \qquad [\textsc{Def}]$$

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

Fig. 4.9: *Typing rules*

1. Abandoning the use of Θ for mapping from variables to *basis* (ie. $X : \tilde{S}\tilde{\alpha}$), and use Γ instead for the mapping (see Typing environment definition above §4.8).

2. Introducing the new typing rules [OUTWHI] and [INWHI], which corresponds to our new constructs.

3. Rules do not have a continuation after **;** (sequential composition). This is because we introduced evaluation contexts earlier (§4.3), and **;** will not appear in the *head subprocess* (eg. $E[P;Q]$ will be written $E[P];E[Q]$). We also have a new typing rule for sequential composition [SEQ] for this reason.

The rules [NAT], [BOOL], [SUM], [NAMEI], [EVAL] are basic language constructs (numbers, booleans, inductive definition of numbers, variables and evaluation of expressions).

[INACT] represents inaction, and has a 'end' typing.

[REQ], [ACC]; [SEND], [RCV] are pairs that represent establishment of session and value/name exchange respectively.

[BR], [SEL] are label branching and selection. Each of the branches have a subtype τ_i, and when the branches finishes, the whole typing $\&\{l_1 : \tau_1, \dots, l_n : \tau_n\}$ and $\oplus\{l_1 : \tau_1, \dots, l_n : \tau_n\}$ ends.

[THR], [CAT] are called session delegation, which comes from π-calculus where channels can be passed as names and use as channels. In the P following a `catch`, the process has a typing received from the `throw`ing side.

[IF], [NRES], [CRES], [VAR], [DEF] are conditionals, name restriction, channel restriction, variable process and recursive process definition respectively. Note that in [IF], Δ to prove e is set to $\emptyset$ to prevent trouble with its channel when inwhile is used as an expression.

[OUTWHI], [INWHI] are the main focus of this work. It represents multichannel `inwhile` and `outwhile`. (It would be easier to understand with $n = 1$, which makes it a simple `inwhile`/`outwhile` loop)

Finally, [SEQ], [CONC] are sequential composition and parallel composition respectively. They will be introduced in detail next as Definition 1 and 2.

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

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

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

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. $\Delta \setminus dom(\Delta')$ extracts session types with sessions unique in Δ

2. $\Delta' \setminus dom(\Delta)$ extracts session types with sessions unique in Δ'

3. $\{k\colon \Delta(k) \setminus \mathsf{end};\ \Delta'(k) \mid k \in dom(\Delta) \cap dom(\Delta')\}$ 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 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*

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

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

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

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

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

The rule can be decomposed into three parts:

1. $\Delta \setminus dom(\Delta')$ which extracts session and runtime types with sessions unique in Δ
2. $\Delta' \setminus dom(\Delta)$ which extracts session and runtime types with sessions unique in Δ'
3. $\{k\colon \beta \circ \beta' | \Delta(k) = \beta \text{ and } \Delta'(k) = \beta'\}$ has three cases
   - If one of the β is a $\dagger$, combine the session type with the $\dagger$ to form an intermediate runtime type $\alpha^{\dagger}$.
   - If βs are duals, combine the types to $\perp$. This covers cases for parallel compositions that does not involve runtime types.
   - If one of the β is an intermediate runtime type $\alpha^{\dagger}$, and the other β is the dual of α, combine the types to $\perp$ but mark the result as an intermediate runtime type $\perp^{\dagger}$ since the $\dagger$ has not been consumed.

Example 2. *Suppose* $\Delta = \{k_1\colon ![\mathsf{bool}].\mathsf{end},\ k_2\colon ![\mathsf{nat}].\mathsf{end},\ k_3\colon ?[\mathsf{nat}].\mathsf{end}\}$ *and*
$\Delta' = \{k_1\colon ?[\mathsf{bool}].\mathsf{end},\ k_2\colon ![\mathsf{bool}].\mathsf{end},\ k_4\colon ![\mathsf{bool}].\mathsf{end}\}$. *Since* k_3 *is unique in* Δ *and* k_4 *is unique in* Δ'*, the two sessions are included in* $\Delta \circ \Delta'$ *without modification. With* $\Delta(k_1) = \overline{\Delta'(k_1)}$ *and* $\Delta(k_2) \neq \overline{\Delta'(k_2)}$*,* k_1 *maps to bottom and* k_2 *is omitted. Therefore*

$$\Delta \circ \Delta' = \{k_1\colon \perp,\ k_3\colon ?[\mathsf{nat}].\mathsf{end},\ k_4\colon ![\mathsf{bool}].\mathsf{end}\}$$

4.5 Subject reduction

Next we are going to present *subject reduction* theorem. Subject reduction will enable us to reduce global composition of processes under a *well-formed ring topology* (defined in Definition 1), such as our implementation of n-body simulation. The main proof can be found in page 57.

Before we go into details of subject reduction theorem, we will begin with auxiliary results for later proofs to build on. The proofs presented here are based on [**?**] with modifications and additions to fit our updated type system with multichannel `inwhile` and `outwhile`.

The Weakening Lemma represents *adding* of mappings to the typing environment. Formally:

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

1. *If* $X \notin dom(\Gamma)$, *then* $\Gamma \cdot X : \tilde{S}\tilde{\alpha} \vdash P \triangleright \Delta$.
2. *If* $a \notin dom(\Gamma)$, *then* $\Gamma \cdot a : S \vdash P \triangleright \Delta$.
3. *If* $k \notin dom(\Delta)$ *and* $\alpha = \perp$ *or* $\alpha = \varepsilon.\mathsf{end}$, *then* $\Gamma \vdash P \triangleright \Delta \cdot k : \alpha$.

Proof. For the first two sequent, simple induction of the derivation tree can show that X and a do not interfere with the typing. For 3, we note that in [INACT] and [VAR], Δ contains only $\varepsilon.\mathsf{end}$ and $\perp$. □

The Strengthening Lemma represents *removal* of mappings from the typing environment, given that they do not change the typing of a process. Formally:

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

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

Proof. Start from $\Delta = \emptyset$, the by induction over all session constructs, showing all three sequent hold. □

The Channel Lemma states that if a channel is free in a process then it will have a typing in Δ, otherwise the typing can only be one of the end types that cannot react with other channels. Formally:

Lemma 3 (Channel Lemma)**.** 1. *If* $\Gamma \vdash P \triangleright \Delta \cdot k : \alpha$ *and* $k \notin fc(P)$, *then* $\alpha = \perp, \varepsilon.\mathsf{end}$.

2. *If* $\Gamma \vdash P \triangleright \Delta$ *and* $k \in fc(P)$, *then* $k \in dom(\Delta)$.

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 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. By induction on the derivation tree. □

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.

4.5.1 Well-formed topology

We now introduce the notion of well-formed *ring* topology. These are the conditions which a correctly designed parallel algorithm based on a ring topology must satisfy.

Definition 1. *A process is under a* well-formed ring topology *if:*

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

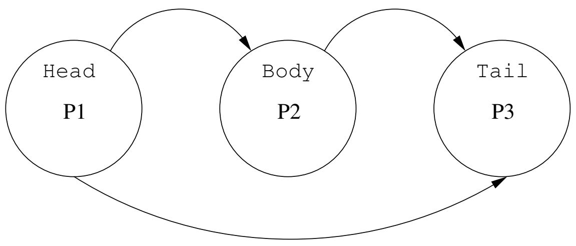


Fig. 4.10: *Ring topology for 3 processes, arrow shows direction of* `outwhile`

We also define a well-formed *intermediate* ring topology, which are the conditions that should hold when the reduction involves the runtime type † as intermediate steps.

Definition 2. *A process is under a well-formed intermediate ring topology if:*

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

4.5.2 Subject congruence theorem

Theorem 1. *Subject congruence is defined by*

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

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: $\Delta_1 = \Delta'_1 \circ \{\tilde{k} \colon \varepsilon.\tilde{\mathsf{end}}\}$ and $\Delta_2 = \Delta'_2 \cdot \{\tilde{k} \colon \varepsilon.\tilde{\mathsf{end}}\}$ with $\Delta'_1 \circ \Delta'_2 = \Delta'_1 \cdot \Delta'_2$ and $\Delta = \Delta'_1 \cdot \Delta'_2 \cdot \{\tilde{k} \colon \tilde{\bot}\}$. Then by the [Bot]-rule, we have:

$$\Gamma \vdash P \triangleright \Delta'_1 \cdot \{\tilde{k} \colon \tilde{\bot}\}$$

Notice that, given the form of Δ above, we know that $dom(\Delta'_2) \cap dom(\Delta'_1) \cdot \{\tilde{k} \colon \ \bot\}) = \emptyset$. Hence by applying Weakening, we have:

$$\Gamma \vdash P \triangleright \Delta'_1 \cdot \Delta'_2 \cdot \{\tilde{k} \colon \tilde{\bot}\}$$

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 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

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

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

$$\begin{cases} \beta_1 = \alpha & \beta_2 = \dagger \\ \beta_1 = \alpha & \beta_2 = \overline{\alpha} \\ \beta_1 = \alpha & \beta_2 = \overline{\alpha}^{\dagger} \\ \beta_1 = \dagger & \beta_2 = \perp \end{cases}$$

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

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

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

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

in all other cases, $k \notin dom(\Delta_3)$ and therefore no parallel composition is possible.

Case $(\nu u)\ P \mid Q \equiv (\nu u)\ (P \mid Q)$ if $u \notin fu(Q)$**.** 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

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

with $\Delta' \cdot k\colon\perp = \Delta_1' \circ \Delta_2'$ 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 3(1) and the fact that $k \notin fc(Q)$ 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 $\Gamma \vdash P \triangleright \Delta_1 \cdot k\colon\perp$, 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 `def` D `in` $\mathbf{0} \equiv \mathbf{0}$**.** Similar to the first case using Weakening and Strengthening.

Case (νu) `def` D `in` $P \equiv$ `def` D `in` $(\nu u)\ P$ if $u \notin fu(D)$**.** Similar to the scope opening case using Weakening and Strengthening.

Case (`def` D `in` P) $\mid Q \equiv$ `def` D `in` $(P \mid Q)$ if $dpv(D) \cap fpv(Q) = \emptyset$**.** 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 $\perp$, by definition of sequential composition (Definition 1), $\Delta(k) = \Delta_1(k).\Delta_2(k) = \varepsilon.\Delta_2(k) = \Delta_2(k)$ as required. □

4.5.3 Subject reduction theorem

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

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

Under a well-formed intermediate ring topology

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

Proof. We assume that

$$\Gamma \vdash e \triangleright S \quad \text{and} \quad e \downarrow c \quad \text{implies} \quad \Gamma \vdash c \triangleright S \tag{4.1}$$

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

Case [LINK] $(\texttt{accept}\ a(k)\ \texttt{in}\ P_1) \mid (\texttt{request}\ a(k)\ \texttt{in}\ P_2) \rightarrow (\nu k)\ (P_1 \mid P_2)$. Suppose $\Gamma \vdash (\texttt{accept}\ a(k)\ \texttt{in}\ P_1) \mid (\texttt{request}\ a(k)\ \texttt{in}\ P_2) \triangleright \Delta$. Then the assumption is derived from:

$$\frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Gamma \vdash P_1 \triangleright \Delta'_1 \cdot k\colon \alpha}{\Gamma \vdash \texttt{accept}\ a(k)\ \texttt{in}\ P_1 \triangleright \Delta'_1} \quad \text{and} \quad \frac{\Gamma \vdash a \triangleright \langle \alpha, \overline{\alpha} \rangle \quad \Gamma \vdash P_2 \triangleright \Delta'_2 \cdot k\colon \overline{\alpha}}{\Gamma \vdash \texttt{request}\ a(k)\ \texttt{in}\ P_2 \triangleright \Delta'_2}$$

and [BOT] with $\Delta'_i \prec \Delta_i$, [CONC] with $\Delta_1 \circ \Delta_2 = \Delta'$, and [BOT] with $\Delta' \prec \Delta$. Then applying [BOT] to P_1 and P_2, we have:

$$\frac{\Gamma \vdash P_1 \triangleright \Delta'_1 \cdot k\colon \alpha}{\Gamma \vdash P_1 \triangleright \Delta_1 \cdot k\colon \alpha} \quad \text{and} \quad \frac{\Gamma \vdash P_2 \triangleright \Delta'_2 \cdot k\colon \overline{\alpha}}{\Gamma \vdash P_2 \triangleright \Delta_2 \cdot k\colon \overline{\alpha}}$$

Then we apply [CONC] to P_1 and P_2 to obtain:

$$\frac{\Gamma \vdash P_1 \triangleright \Delta_1 \cdot k\colon \alpha \quad \Gamma \vdash P_2 \triangleright \Delta_2 \cdot k\colon \overline{\alpha}}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta' \cdot k\colon \bot}$$

Now applying [CRES] and [BOT], we are done.

Case [COM] $(k![\tilde{e}]; P_1) \mid (k?(\tilde{x})\ \texttt{in}\ P_2) \rightarrow P_1 \mid P_2[\tilde{c}/\tilde{x}]$ with $\tilde{e} \downarrow \tilde{c}$. The assumption is derived from:

$$\frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \quad \Gamma \vdash P_1 \triangleright \Delta'_1 \cdot k\colon \alpha}{\Gamma \vdash k![\tilde{e}]; P_1 \triangleright \Delta'_1 \cdot k\colon ![\tilde{S}]} \quad \text{and} \quad \frac{\Gamma \cdot \tilde{x}\colon \tilde{S} \vdash P_2 \triangleright \Delta'_2 \cdot k\colon \overline{\alpha}}{\Gamma \vdash k?(\tilde{x})\ \texttt{in}\ P_2 \triangleright \Delta'_2 \cdot k\colon ?[\tilde{S}]; \overline{\alpha}}$$

and [BOT] with $\Delta'_i \prec \Delta_i$, [CONC] with $\Delta_1 \circ \Delta_2 \cdot k\colon \bot = \Delta'$, and [BOT] with $\Delta' \prec \Delta$. Then by (4.1), we know $\Gamma \vdash \tilde{c} \triangleright \tilde{S}$. By applying Substitution Lemma, we have:

$$\Gamma \vdash P_2[\tilde{c}/\tilde{x}] \triangleright \Delta'_2 \cdot k\colon \overline{\alpha}$$

Now the application of [BOT] and [CONC] to P_1 and $P_2[\tilde{c}/\tilde{x}]$, then by [BOT], we complete this case.

Case [STR]**.** By Subject-Congruence.

Case `inwhile`/`outwhile` for 3 processes $(\nu k_{12},k_{23},k_{13})\ (P_1 \mid P_2 \mid P_3)$**.** Assume well-formed ring topology (Definition 1)

Case $E[e] \to E[\mathtt{true}]$

By [OUTWHI1],

$$\begin{aligned}
&(\nu k_{12},k_{23},k_{13})\ (\langle k_{13},k_{12}\rangle.\mathtt{outwhile}(e)\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
&\qquad k_{23}.\mathtt{outwhile}(k_{12}.\mathtt{inwhile})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
&\qquad \langle k_{13},k_{23}\rangle.\mathtt{inwhile}\{\ Q_n[k_{13},k_{23}]\ \}) \\
\to\ &(\nu k_{12},k_{23},k_{13})\ (k_{13} \dagger [\mathtt{true}] \mid k_{12} \dagger [\mathtt{true}] \mid \\
&\qquad Q_1[k_{13},k_{12}];\ \langle k_{13},k_{12}\rangle.\mathtt{outwhile}(e')\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
&\qquad k_{23}.\mathtt{outwhile}(k_{12}.\mathtt{inwhile})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
&\qquad \langle k_{13},k_{23}\rangle.\mathtt{inwhile}\{\ Q_3[k_{13},k_{23}]\ \})
\end{aligned}$$

$$\begin{aligned}
\Gamma \vdash (k_{13} \dagger [\mathtt{true}] \mid k_{12} \dagger [\mathtt{true}] \mid Q_1;P_1 \mid P_2 \mid P_3) \triangleright \{&k_{12}\colon T_{12};![T_{12}]^* \circ ?[T'_{12}]^{*\dagger}, \\
&k_{13}\colon T_{13};![T_{13}]^* \circ ?[T'_{13}]^{*\dagger}, \\
&k_{23}\colon ![T_{23}]^* \circ ?[T'_{23}]^*\}
\end{aligned}$$

By [INWHI1],

$$\begin{aligned}
&(\nu k_{12},k_{23},k_{13})\ (k_{13} \dagger [\mathtt{true}] \mid k_{12} \dagger [\mathtt{true}] \mid \\
&\qquad Q_1[k_{13},k_{12}];\ \langle k_{13},k_{12}\rangle.\mathtt{outwhile}(e')\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
&\qquad k_{23}.\mathtt{outwhile}(k_{12}.\mathtt{inwhile})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
&\qquad \langle k_{13},k_{23}\rangle.\mathtt{inwhile}\{\ Q_3[k_{13},k_{23}]\ \}) \\
\to\ &(\nu k_{12},k_{23},k_{13})\ (k_{13} \dagger [\mathtt{true}] \mid \\
&\qquad Q_1[k_{13},k_{12}];\ \langle k_{13},k_{12}\rangle.\mathtt{outwhile}(e')\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
&\qquad k_{23}.\mathtt{outwhile}(\mathtt{true})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
&\qquad \langle k_{13},k_{23}\rangle.\mathtt{inwhile}\{\ Q_3[k_{13},k_{23}]\ \})
\end{aligned}$$

$$\Gamma \vdash (k_{13} \dagger [\mathtt{true}] \mid Q_1;P_1 \mid P_2 \mid P_3) \triangleright \{k_{12}\colon T_{12};![T_{12}]^* \circ \varepsilon.\mathsf{end}^\dagger, k_{13}\colon T_{13};![T_{13}]^* \circ ?[T'_{13}]^{*\dagger}, k_{23}\colon ![T_{23}]^* \circ ?[T'_{23}]^*\}$$

By [OUTWHI1],

$$\begin{array}{rl}
(\nu k_{12},k_{23},k_{13}) & (k_{13} \dagger [\texttt{true}] \mid \\
& Q_1[k_{13},k_{12}];\ \langle k_{13},k_{12}\rangle.\texttt{outwhile}(e')\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
& k_{23}.\texttt{outwhile}(k_{12}.\texttt{inwhile})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
& \langle k_{13},k_{23}\rangle.\texttt{inwhile}\{\ Q_3[k_{13},k_{23}]\ \}) \\
\to (\nu k_{12},k_{23},k_{13}) & (k_{13} \dagger [\texttt{true}] \mid k_{23} \dagger [\texttt{true}] \mid \\
& Q_1[k_{13},k_{12}];\ \langle k_{13},k_{12}\rangle.\texttt{outwhile}(e')\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
& Q_2[k_{12},k_{23}];\ k_{23}.\texttt{outwhile}(\texttt{true})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
& \langle k_{13},k_{23}\rangle.\texttt{inwhile}\{\ Q_3[k_{13},k_{23}]\ \})
\end{array}$$

$$\begin{array}{r}
\Gamma \vdash (k_{13} \dagger [\texttt{true}] \mid k_{23} \dagger [\texttt{true}] \mid Q_1;P_1 \mid Q_2;P_2 \mid P_3) \triangleright \{k_{12} \colon T_{12};![T_{12}]^* \circ T_{12}'^{\dagger};?[T_{12}']^*, \\
k_{13} \colon T_{13};![T_{13}]^* \circ ?[T_{13}']^{*\dagger}, \\
k_{23} \colon T_{23};![T_{23}]^* \circ ?[T_{23}']^{*\dagger}\}
\end{array}$$

By [INWHI1],

$$\begin{array}{rl}
(\nu k_{12},k_{23},k_{13}) & (k_{13} \dagger [\texttt{true}] \mid k_{23} \dagger [\texttt{true}] \mid \\
& Q_1[k_{13},k_{12}];\ \langle k_{13},k_{12}\rangle.\texttt{outwhile}(e')\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
& Q_2[k_{12},k_{23}];\ k_{23}.\texttt{outwhile}(\texttt{true})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
& \langle k_{13},k_{23}\rangle.\texttt{inwhile}\{\ Q_n[k_{13},k_{23}]\ \}) \\
\to (\nu k_{12},k_{23},k_{13}) & (Q_1[k_{13},k_{12}];\ \langle k_{13},k_{12}\rangle.\texttt{outwhile}(e')\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
& Q_2[k_{12},k_{23}];\ k_{23}.\texttt{outwhile}(\texttt{true})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
& Q_3[k_{13},k_{23}];\ \langle k_{13},k_{23}\rangle.\texttt{inwhile}\{\ Q_3[k_{13},k_{23}]\ \})
\end{array}$$

$$\begin{array}{r}
\Gamma \vdash (Q_1;P_1 \mid Q_2;P_2 \mid Q_3;P_3) \triangleright \{k_{12} \colon T_{12};![T_{12}]^* \circ T_{12}'^{\dagger};?[T_{12}']^*, \\
k_{13} \colon T_{13};![T_{13}]^* \circ T_{13}'^{\dagger};?[T_{13}']^*, \\
k_{23} \colon T_{23};![T_{23}]^* \circ T_{23}'^{\dagger};?[T_{23}']^*\} \\
\Gamma \vdash (Q_1;P_1 \mid Q_2;P_2 \mid Q_3;P_3) \triangleright \{k_{12} \colon\ \bot^{\dagger};\bot, k_{13} \colon\ \bot^{\dagger};\bot, k_{23} \colon\ \bot^{\dagger};\bot\}
\end{array}$$

Case $E[e] \to E[\texttt{false}]$

By [OUTWHI2],

$$\begin{array}{rl}
(\nu k_{12},k_{23},k_{13}) & (\langle k_{13},k_{12}\rangle.\texttt{outwhile}(e)\{\ Q_1[k_{13},k_{12}]\ \} \mid \\
& k_{23}.\texttt{outwhile}(k_{12}.\texttt{inwhile})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
& \langle k_{13},k_{23}\rangle.\texttt{inwhile}\{\ Q_n[k_{13},k_{23}]\ \}) \\
\to (\nu k_{12},k_{23},k_{13}) & (k_{13} \dagger [\texttt{false}] \mid k_{12} \dagger [\texttt{false}] \mid \mathbf{0} \mid \\
& k_{23}.\texttt{outwhile}(k_{12}.\texttt{inwhile})\{\ Q_2[k_{12},k_{23}]\ \} \mid \\
& \langle k_{13},k_{23}\rangle.\texttt{inwhile}\{\ Q_3[k_{13},k_{23}]\ \})
\end{array}$$

$\Gamma \vdash (k_{13} \dagger [\mathtt{true}] \mid k_{12} \dagger [\mathtt{true}] \mid \mathbf{0} \mid P_2 \mid P_3) \triangleright \{k_{12}\colon \varepsilon.\mathsf{end} \circ ?[T'_{12}]^{*\dagger}, k_{13}\colon \varepsilon.\mathsf{end} \circ ?[T'_{13}]^{*\dagger}, k_{23}\colon ![T_{23}]^{*} \circ ?[T'_{23}]^{*}\}$

By [INWHI2],

$$\begin{aligned}
&(\nu k_{12}, k_{23}, k_{13})\ (\langle k_{13}, k_{12}\rangle.\mathtt{outwhile}(e)\{\ Q_1[k_{13}, k_{12}]\ \} \mid \\
&\qquad k_{23}.\mathtt{outwhile}(k_{12}.\mathtt{inwhile})\{\ Q_2[k_{12}, k_{23}]\ \} \mid \\
&\qquad \langle k_{13}, k_{23}\rangle.\mathtt{inwhile}\{\ Q_n[k_{13}, k_{23}]\ \}) \\
\to &(\nu k_{12}, k_{23}, k_{13})\ (k_{13} \dagger [\mathtt{false}] \mid \mathbf{0} \mid \\
&\qquad k_{23}.\mathtt{outwhile}(\mathtt{false})\{\ Q_2[k_{12}, k_{23}]\ \} \mid \\
&\qquad \langle k_{13}, k_{23}\rangle.\mathtt{inwhile}\{\ Q_3[k_{13}, k_{23}]\ \})
\end{aligned}$$

$\Gamma \vdash (k_{13} \dagger [\mathtt{true}] \mid k_{12} \dagger [\mathtt{true}] \mid \mathbf{0} \mid P_2 \mid P_3) \triangleright \{k_{12}\colon \varepsilon.\mathsf{end}, k_{13}\colon \varepsilon.\mathsf{end} \circ ?[T'_{13}]^{*\dagger}, k_{23}\colon ![T_{23}]^{*} \circ ?[T'_{23}]^{*}\}$

By [OUTWHI2],

$$\begin{aligned}
&(\nu k_{12}, k_{23}, k_{13})\ (k_{13} \dagger [\mathtt{false}] \mid \mathbf{0} \mid k_{23}.\mathtt{outwhile}(\mathtt{false})\{\ Q_2[k_{12}, k_{23}]\ \} \mid \\
&\qquad \langle k_{13}, k_{23}\rangle.\mathtt{inwhile}\{\ Q_3[k_{13}, k_{23}]\ \}) \\
\to &(\nu k_{12}, k_{23}, k_{13})\ (k_{13} \dagger [\mathtt{false}] \mid k_{23} \dagger [\mathtt{false}] \mid \mathbf{0} \mid \mathbf{0} \mid \\
&\qquad \langle k_{13}, k_{23}\rangle.\mathtt{inwhile}\{\ Q_3[k_{13}, k_{23}]\ \})
\end{aligned}$$

$\Gamma \vdash (k_{13} \dagger [\mathtt{true}] \mid k_{12} \dagger [\mathtt{true}] \mid \mathbf{0} \mid P_2 \mid P_3) \triangleright \{k_{12}\colon \varepsilon.\mathsf{end}, k_{13}\colon \varepsilon.\mathsf{end} \circ ?[T'_{13}]^{*\dagger}, k_{23}\colon \varepsilon.\mathsf{end} \circ ?[T'_{23}]^{*}\}$

By [INWHI2],

$$\begin{aligned}
&(\nu k_{12}, k_{23}, k_{13})\ (k_{13} \dagger [\mathtt{false}] \mid k_{23} \dagger [\mathtt{false}] \mid \mathbf{0} \mid \mathbf{0} \mid \langle k_{13}, k_{23}\rangle.\mathtt{inwhile}\{\ Q_3[k_{13}, k_{23}]\ \}) \\
\to &(\nu k_{12}, k_{23}, k_{13})\ (\mathbf{0} \mid \mathbf{0} \mid \mathbf{0})
\end{aligned}$$

$$\Gamma \vdash (k_{13} \dagger [\mathtt{true}] \mid k_{12} \dagger [\mathtt{true}] \mid \mathbf{0} \mid \mathbf{0} \mid \mathbf{0}) \triangleright \{k_{12}\colon \varepsilon.\mathsf{end}, k_{13}\colon \varepsilon.\mathsf{end}, k_{23}\colon \varepsilon.\mathsf{end}\}$$

Finally, apply [BOT].

The result can be extended to $(P_1 \mid \ldots \mid P_n)$ by expanding the middle process from P_2 to $P_i \quad (2 \le i \le n)$.

Case [PASS] $(\mathtt{throw}\, k[k'];P_1) \mid (\mathtt{catch}\, k(k')\ \mathtt{in}\, P_2) \to P_1 \mid P_2$**.** The assumption is derived from:

$$\frac{\Gamma \vdash P_1 \triangleright \Delta'_1 \cdot k\colon \beta}{\Gamma \vdash \mathtt{throw}\, k[k'];P_1 \triangleright \Delta'_1 \cdot k\colon ![\alpha];\beta \cdot k'\colon \alpha}$$

and

$$\frac{\Gamma \vdash P_2 \triangleright \Delta'_2 \cdot k\colon \overline{\beta} \cdot k'\colon \alpha}{\Gamma \vdash \mathtt{catch}\, k(k')\ \mathtt{in}\, P_2 \triangleright \Delta'_2 \cdot k\colon ?[\alpha];\overline{\beta}}$$

and [BOT] with $\Delta'_i \prec \Delta_i$, [CONC] with $\Delta_1 \circ \Delta_2 \cdot k\colon \bot \cdot k'\colon \alpha = \Delta'$ and [BOT] with $\Delta' \prec \Delta$. Note that $k, k' \notin dom(\Delta_1, \Delta_2, \Delta'_1, \Delta'_2)$. By applying [BOT], [CONC] to P_1 and P_2, and then by [BOT], we obtain the required result.

Case [IF1],[IF2]**.** Trivial.

Case [DEF] $\texttt{def}\ D\ \texttt{in}\ (X[\tilde{e}k] \mid Q) \ \rightarrow\ \texttt{def}\ D\ \texttt{in}\ (P[\tilde{c}/\tilde{x}] \mid Q)$ with $\tilde{e} \downarrow \tilde{c}$ and $X(\tilde{x}k) = P \in D$**.** Simplifying the recursive definition to the single case, we set $D = (X(\tilde{x}\tilde{k}) = P)$. Then the assumption is derived from:

$$\dfrac{\Gamma \cdot X : \tilde{S}\tilde{\alpha} \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \tilde{k} : \tilde{\alpha} \qquad \dfrac{\Gamma X : \tilde{S}\tilde{\alpha} \vdash X[\tilde{e}\tilde{k}] \triangleright \Delta_1' \cdot \tilde{k} : \tilde{\alpha} \quad \Gamma \cdot X : \tilde{S}\tilde{\alpha}; \vdash Q \triangleright \Delta_2'}{\Gamma \cdot X : \tilde{S}\tilde{\alpha} \vdash X[\tilde{e}\tilde{k}] \mid Q \triangleright \Delta'' \cdot \tilde{k} : \tilde{\alpha} \quad \Delta'' \prec \Delta'}}{\Gamma \vdash \texttt{def}\ X(\tilde{x}\tilde{k}) = P\ \texttt{in}\ (X[\tilde{e}\tilde{k}] \mid Q) \triangleright \Delta' \cdot \tilde{k} : \tilde{\alpha}}$$

with $\Delta_0 = \Delta' \cdot \tilde{k} : \tilde{\alpha}$, $\Delta' = \Delta_1' \circ \Delta_2'$ and $\Delta_0 \prec \Delta$. Note that Δ_1' contains only $\perp$ or $\varepsilon.\mathsf{end}$. Then applying Substitution Lemma to P, we have:

$$\Gamma \cdot X : \tilde{S}\tilde{\alpha} \vdash P[\tilde{c}/\tilde{x}] \triangleright \tilde{k} : \tilde{\alpha}$$

Notice that $\tilde{k} \cap dom(\Delta_1') = \emptyset$, since $(\Delta_1' \circ \Delta_2') \cdot \tilde{k} : \tilde{\alpha}$ is defined. Then by Weakening, we have:

$$\Gamma \cdot X : \tilde{S}\tilde{\alpha} \vdash P[\tilde{c}/\tilde{x}] \triangleright \Delta_1' \cdot \tilde{k} : \tilde{\alpha}$$

Now by [CONC], we have

$$\Gamma \cdot X : \tilde{S}\tilde{\alpha} \vdash P[\tilde{c}/\tilde{x}] \mid Q \triangleright \Delta'' \cdot \tilde{k} : \tilde{\alpha}$$

Finally by [BOT] ($\Delta'' \prec \Delta'$), then by [DEF], we obtain:

$$\Gamma \vdash \texttt{def}\ X(\tilde{x}\tilde{k}) = P\ \texttt{in}\ (P[\tilde{c}/\tilde{x}] \mid Q) \triangleright \Delta' \cdot \tilde{k} : \tilde{\alpha}$$

Then we can apply [BOT] to obtain Δ, as desired.

□

4.6 Progress property

We can now model any process (composition of processes) that uses a ring topology, and show that they are deadlock free if they conform to our definition of well-formed topology (Definition 1). An exception is when there are shared names in the process P, composition with other processes might change the well-formed topology property of the process thus making our deadlock-free claim invalid.

The proof uses subject reduction theorem proven above (§4.5.3).

Theorem 1. *Suppose* $\Gamma \vdash P \triangleright \Delta$ *and P is under a well-formed topology* without shared names *Then P is deadlock free*

$$\textit{ie. Suppose } P \ \rightarrow^*\ P' \textit{ then} \begin{cases} \textit{either} & P' \equiv \mathbf{0} \\ \textit{or} & \exists Q\ (P' \ \rightarrow\ Q) \end{cases}$$

Proof. Let P be a process under well-formed topology and do not have shared names. Given the typings are correct, under subject reduction, no process will reduce to a deadlock state so P is deadlock free in all cases we have shown in above proof. □

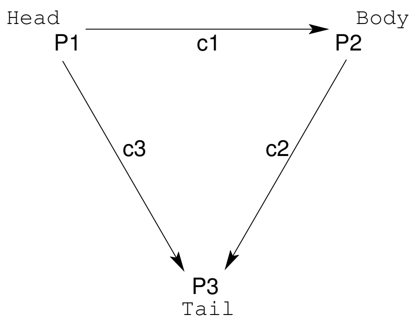


Fig. 4.11: *Shared channels c_1, c_2, c_3 between 3 n-body processes*

4.7 Correctness proof for n-body simulation

In Table 3.1 from the previous section, we have shown the session declarations of the ring topology used in our n-body implementation.

4.7.1 N-body simulation in session calculus

Fig. 4.11 shows the shared channels between the processes in the n-body simulation. The processes can be represented in session calculus by

$$P_1 \equiv \texttt{request}\ c_1(k_{12})\ \texttt{in}\ \texttt{request}\ c_3(k_{13})\ \texttt{in}\ k_{12}?(int)\ \texttt{in}\ \langle k_{12}, k_{13}\rangle.\texttt{outwhile}(e)\{\ Q_1\ \}$$
$$P_2 \equiv \texttt{request}\ c_2(k_{23})\ \texttt{in}\ \texttt{accept}\ c_1(k_{12})\ \texttt{in}\ k_{23}?(int)\ \texttt{in}\ k_{12}![int]; k_{23}.\texttt{outwhile}(k_{12}.\texttt{inwhile})\{\ Q_2\ \}$$
$$P_3 \equiv \texttt{accept}\ c_2(k_{23})\ \texttt{in}\ \texttt{accept}\ c_3(k_{13})\ \texttt{in}\ k_{23}![int]; \langle k_{13}, k_{23}\rangle.\texttt{inwhile}\{\ Q_3\ \}$$

$$Q_1 \equiv \langle k_{12}, k_{13}\rangle.\texttt{outwhile}(e)\{\ k_{12}![\texttt{Particle}[]] \mid k_{13}?(\texttt{Particle}[])\ \texttt{in}\ \mathbf{0}\ \}$$
$$Q_2 \equiv k_{23}.\texttt{outwhile}(k_{12}.\texttt{inwhile})\{\ k_{23}![\texttt{Particle}[]] \mid k_{12}?(\texttt{Particle}[])\ \texttt{in}\ \mathbf{0}\ \}$$
$$Q_3 \equiv \langle k_{13}, k_{23}\rangle.\texttt{inwhile}\{\ k_{13}![\texttt{Particle}[]] \mid k_{23}?(\texttt{Particle}[])\ \texttt{in}\ \mathbf{0}\ \}$$

The typing of the processes are

$$\Gamma \vdash P_1 \triangleright \{k_{12}\colon\ ?[int]; ![![![\texttt{Particle}[]].\mathsf{end}]^*]^*.\mathsf{end},\ k_{13}\colon\ ![![?[\texttt{Particle}[]].\mathsf{end}]^*]^*.\mathsf{end}\}$$
$$\Gamma \vdash P_2 \triangleright \{k_{23}\colon\ ?[int]; ![![![\texttt{Particle}[]].\mathsf{end}]^*]^*.\mathsf{end},\ k_{12}\colon\ ![int]; ?[?[?[\texttt{Particle}[]].\mathsf{end}]^*]^*.\mathsf{end}\}$$
$$\Gamma \vdash P_3 \triangleright \{k_{13}\colon\ ?[?[![\texttt{Particle}[]].\mathsf{end}]^*]^*.\mathsf{end},\ k_{23}\colon\ ![int]; ?[?[?[\texttt{Particle}[]].\mathsf{end}]^*]^*.\mathsf{end}\}$$

Reduction

To prove that the our program, $((\nu k)\ _{12}, k_{23}, k_{13})(P_1 \mid P_2 \mid P_3)$ is deadlock free, it needs to satisfy the preconditions laid out in progress property (§4.6).

- Process is under a well-formed topology
- Process do not have shared names

We first inspect the typing of the process from the end,

$$\Gamma \vdash (k_{12}!\texttt{[Particle[]]} \mid k_{13}?(\texttt{Particle[]})\ \texttt{in}\ \mathbf{0}) \triangleright \{k_{12}\colon ![\texttt{Particle}[]].\mathsf{end},\ k_{13}\colon ?[\texttt{Particle}[]].\mathsf{end}\}$$
$$\Gamma \vdash (k_{23}!\texttt{[Particle[]]} \mid k_{12}?(\texttt{Particle[]})\ \texttt{in}\ \mathbf{0}) \triangleright \{k_{23}\colon ![\texttt{Particle}[]].\mathsf{end},\ k_{12}\colon ?[\texttt{Particle}[]].\mathsf{end}\}$$
$$\Gamma \vdash (k_{13}!\texttt{[Particle[]]} \mid k_{23}?(\texttt{Particle[]})\ \texttt{in}\ \mathbf{0}) \triangleright \{k_{13}\colon ![\texttt{Particle}[]].\mathsf{end},\ k_{23}\colon ?[\texttt{Particle}[]].\mathsf{end}\}$$

When the three processes are composed, all of the sessions have a dual, satisfying the conditions for $Q_1, Q_i, and Q_n$ in Definition 1

$$\Gamma \vdash Q_1 \triangleright \{k_{1,2}\colon T_{1,2},\ k_{1,n}\colon T_{1,n}\}$$
$$\Gamma \vdash Q_i \triangleright \{k_{i,i+1}\colon T_{i,i+1},\ k_{i-1,i}\colon T'_{i-1,i}\}$$
$$\Gamma \vdash Q_n \triangleright \{k_{1,n}\colon T_{1,n}{}',\ k_{n-1,n}\colon T_{n-1,n}{}'\}$$

and $\overline{T_{i,j}} = T'_{i,j}$.

Given above, the subprocesses Q_1, Q_2, Q_3 is under a well-formed topology by matching the structures in the definition.

We can also show that the processes P_1, P_2, P_3 are under a well-formed topology, after the sessions are established with `request` and `accept` pairs and node information exchange (send and receive of a single `int`). Shared names do not exist in the process after the link phase.

Therefore, by progress property, the n-body implementation is deadlock free.

4.8 Summary

In this chapter we formalised the multichannel **`inwhile`** and **`outwhile`** construct in session calculus. An updated session calculus and session typing system is presented. We have also included a proof for subject reduction of `inwhile outwhile`, and by that shown a well-formed ring topology will have progress property and never deadlocks.

Chapter 5

Testing and Evaluation

In this chapter we will first discuss some failed attempts (§5.1) and testing results (§5.2) that influenced the current design of SJ applications on Axel.

Then we will look at the benchmark results of our n-body implementation with SJ comparing our results to non-session based message passing solution such as MPJExpress (§5.3.2). We will also look at the performance of our C translation, and compare the results with ordinary SJ and SJ with FPGA (§5.3.2).

5.1 Alternative designs

5.1.1 SJ and acceleration hardware allocation

Current design The current design of applications on cluster maps 1 SJ executable to 1 hardware accelerator. Multiple SJ executables can be run on a single node to use multiple hardware. Such as node using both FPGA and CPU shown in the example Fig 5.1. This approach is very simple and the class design can be minimalistic, with implementations for specific hardware encapsulated in a single class. eg. `FPGAHead` is a `Head` node that uses FPGA. `CPUBody` is a `Body` node that uses C on CPU. `JavaTail` is a `Tail` node that uses SJ/Java on CPU. We can execute the three nodes to have a hybrid execution with FPGA and SJ.

SJ as a coordinator We have previously considered using a single SJ application running on a node to be a coordinator between hardware accelerators and other nodes. This structure can allow dynamic load balancing if the hardware accelerators are of different performance (eg. coordinating both a GPU and a FPGA connected to the same node), especially since SJ will be in a central position that overlooks all aspects of inter-node and inter-component communications.

Typically there will be a single complicated function that we wish to accelerate. In the case of our n-body simulation, the said function is `computeForces()`. *If a SJ program controls more than one hardware accelerators*, then it would make sense to have `computeForces()` transparently

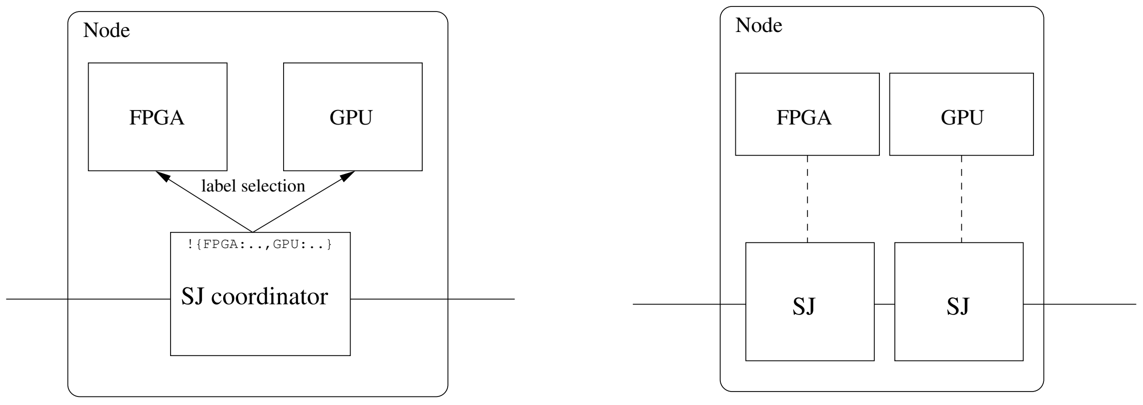


Fig. 5.1: *Left: SJ as a coordinator, Right: current design*

handle load balancing between the hardware accelerators based on their workload or performance. There might be a chance to exploit some advanced session programming constructs such as *label selection* if the communication between hardware accelerators and main SJ components is also session-typed.

The reason this is ultimately not implemented is because the lack of advantage over ordinary method call. We currently only have a single type of hardware accelerator implemented (FPGA), and the performance compared with native implementations are far from good; or in other words, we do not have spare performance for features such as a load balancer. In the current toolchain from Axel's SDK, input values are partitioned to different hardware components based on a static XML configuration file. This has worked well so far on applications built for Axel, with [**?**] showing good results with a 1/3 GPU and 2/3 FPGA manual partition.

Dynamic load balancing based on SJ, however, remains a novel idea for future work.

5.1.2 Communication medium

Explicit SJ communication The original proposal was to introduce label selection to the accelerator - CPU communication, so in our main SJ program we can demonstrate using two labels for the two compute tasks, `computeForces()` invoked in every iteration and `computePositions()` invoked in every completed ring to update the positions of the particles. Because of reasons outlined in 3.2.1 that FPGA is more suitable to accelerate a single task, we instead. Seeing that there are no extra benefits on using explicit communication in accelerator - CPU link, we reverted to using an implicit (class based method call) communication between *CPU part* and *FPGA part* as in current design.

However if **SJ as a coordinator** were implemented, most likely the communication medium will be in SJ.

Shared memory between SJ and hardware accelerators This was attempted but was later abandoned. There are many advantages in using shared memory (SHM) to share data between

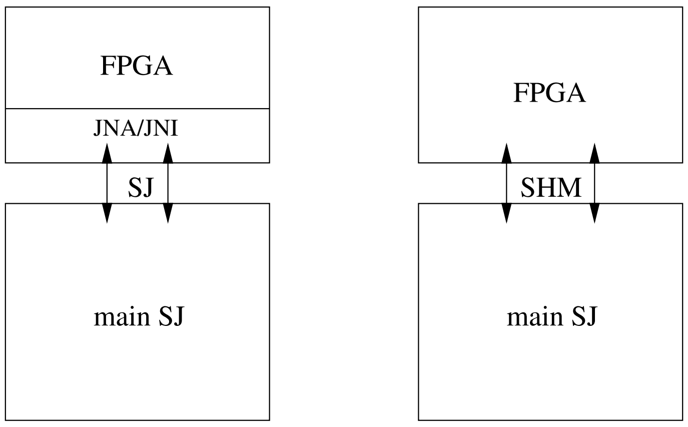


Fig. 5.2: *Left: using SJ to share data, Right: using shared memory to share data*

two different processes. Shared memory is efficient, and is the method which data is exchanged between C code and FPGA in the Axel SDK [**?**, **?**]. Some parts of FPGA memory is mapped to main memory to pass simple function arguments (such as size of array FPGA should expect) and o kick start the computation on FPGA.

If Java have a robust mechanism to access shared memory directly, this will save us a lot of effort (and overhead) passing data as function arguments, then through a cross-language library (JNA) to communicate with the FPGA. As expected from the design principals of Java, SHM with the host operating system is not possible without the use of *Java Native Interface* (JNI), because of the closed nature of the Java Virtual Machine. This offers no obvious advantage over our JNA-based solution.

5.2 Pre-implementation tests: inner product

At the initial stage of the project, we wish to run a simple algorithm to check that all the libraries work as intended and the communication and the overall design are correct.

Two of the main features we wished to examine were methods of using JNA and the magnitude of overhead when using JNA to access native functions.

Inner product was the algorithm used for this test, where all the computing nodes are first loaded with $a_i \; i \in \{1..n\}$. At each step, $\sum\limits_{i=1}^{n} a_i \times b_i$ where $b_i \; i \in \{1..n\}$ is received from a neighbour node.

5.2.1 JNA direct mapping

The JNA project states that there are two methods of using the library. **interface mapping** and **direct mapping**. The usage of the two methods are quite similar, and the developers of JNA

```java
public class CPUInnerProduct extends InnerProduct {
    public CPUInnerProduct() {
        Native.register(System.getProperty("user.dir")+"/lib/libinp_cpu.so");
    }

    // Direct mapped function
    public static native int innerproduct(int[] a, int[] b, int size);
}
```

Listing 5.1: *JNA direct mapping example*

library strongly encourages the use of direct mapping for high performance applications [1] because of the lower native function calling overhead.

To tell the difference between the two methods, we should first remember that JNA analyses native libraries at *runtime*.

For interface mapping, a Java interface needs to be supplied. For example, `LibExample` interface in Listing A.2 specifies what JNA should expect in our supplied shared library. In our example, the shared library is `libexample.so` in the calling directory. At runtime, JNA analyses the Library interface and instantiate any new classes for our declared type. `LibExample` does not use any external classes or datastructure, but it is common that some parameters map to a `class` or C `struct`. For example, our implementation of n-body simulation uses a `Particle` class to represent a particle.

Instead of providing a subclass of JNA's `Library` interface, the direct mapping method allows taking the function signature and declare them directly as a **`native static`** method, given there is a build-in mapping between chosen *primitives* or *arrays of primitives* This is much more convenient for simple datatype than normal interface mapping; However, this method do not support all function parameter and return types. In particular, arrays of Pointer-based classes are not allowed in direct mapped methods.

5.2.2 JNA interface mapping and direct mapping

Fig. 5.3 shows the performance when direct mapping was compared to interface mapping method and native SJ. From the figure it can be seen that direct mapping has a slight performance edge over interface mapping. *CPU* in the legend refers to computation code written in C and runs in the CPU.

5.2.3 Execution in CPU and FPGA

Next, we compare the performance of running inner product in CPU and FPGA.

[1] `https://jna.dev.java.net/#direct`

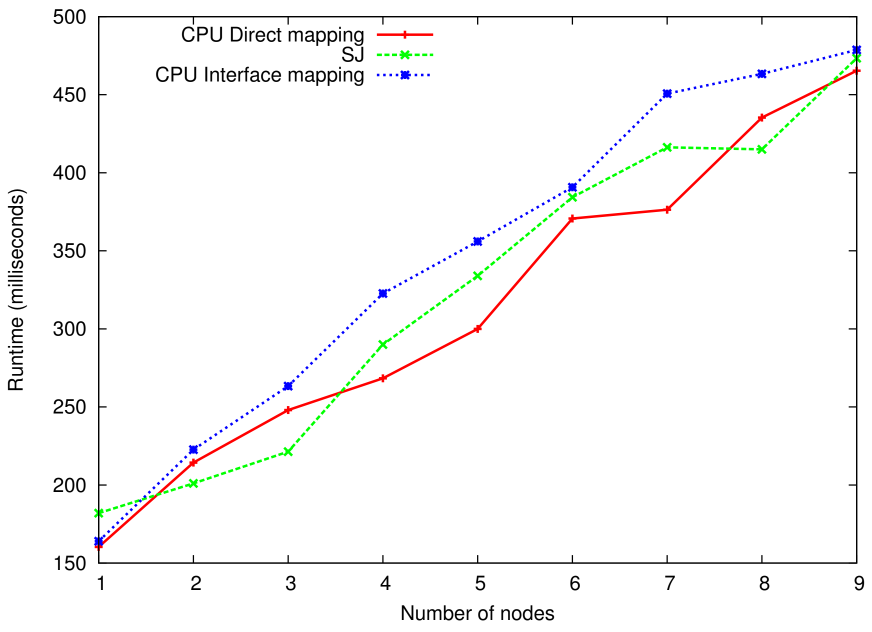


Fig. 5.3: *JNA direct mapping shows better performance over interface mapping*

In Fig. 5.4 FPGA shows a much worse performance than either SJ or CPU implementation. This can be explained by the lack of complex operations and pipelinable operations. The calculation of inner product involves n multiplications and $(n-1)$ summation steps, so it is of O(n) in each node. Our main implementation to run on the cluster, the n-body simulation, calculates the aggregate forces between a particle and other $(n-1)$ particles. This is repeated for n particles, and requires $n \times (n-1)$ operations in total. N-body simulation is therefore more complex than inner product at O(n^2).

When a simple algorithm is implemented on the FPGA, it spends a low proportion of time in computing the results, but a high proportion of time in the transfer of data to and from the FPGA memory which is separate from the main memory. This might overweigh all benefits of using a fast hardware accelerator, since in an ordinary microprocessor, the data can be accessed directly and do not need the extra data transfer.

It is possible that with a big enough problem size, the total computation time in FPGA that includes data transfer will be less than total computation time in microprocessor. But do remember when the problem size is increased, the data transfer time will increase accordingly. If the total computation time increases in a rate equal or lower than which a CPU scales, as we saw in Fig. 5.3, then FPGA might not be a feasible solution for your task.

Nonetheless, the main reason for this test is to verify that FPGAs can be operated from SJ and calculates results correctly. It had been shown in [**?**] that n-body is a viable algorithm to run on FPGAs.

The conclusion of the initial testing with inner product is

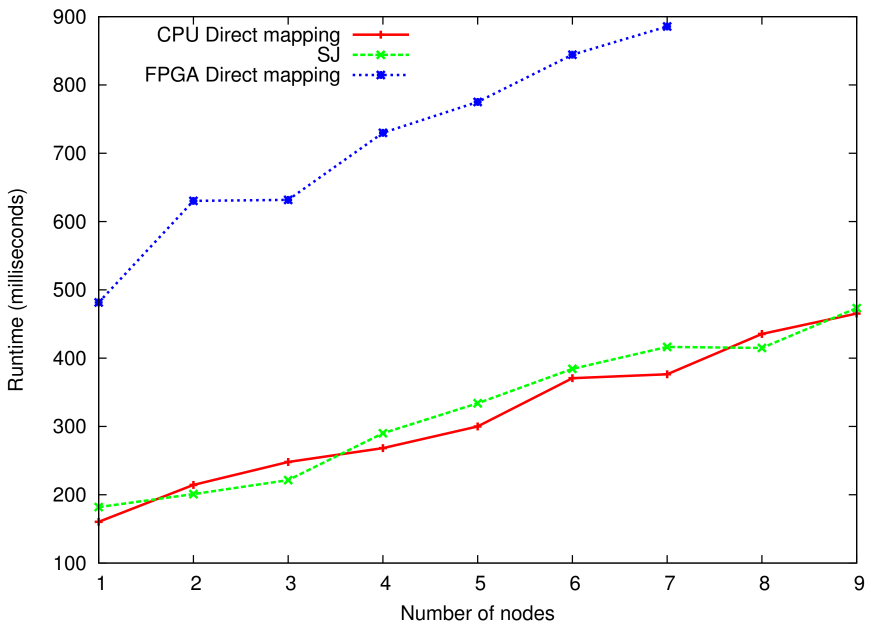


Fig. 5.4: *FPGA performance of innerproduct is much worse than JNA direct mapping and native SJ implementation*

- Direct mapping, as the authors of JNA have suggested, yields better performance.
- It is necessary that the main computation function in the parallel algorithms to be sufficiently complex to overcome the data transfer overhead

5.3 Benchmarks

This section contains all benchmark results and comparison between different implementations with Axel and SJ.

	Computation	Communication
SJ	Java	SJ
SJ + C	C with JNA	SJ
SJ + FPGA	FPGA with JNA	SJ
MPJExpress	Java	MPJExpress
Translated C	C	TCP-sockets, translated from SJ

Table 5.1: *Implementations which we will compare*

5.3.1 Benchmark methodology

The initial particles configurations came from the *Dubinski 1995 data set* available on `http://bima.astro.umd.edu/nemo/archive/`. Each node will load their partition of the particles, the starting offset of the particle indices are calculated by the node's position in the ring.

`Head` will be node0 and `Tail` will be node8, the number of particles each node loads is specified as a command-line argument.

This allows flexible assignment of number of particles to each node such that more particles can be assigned to a node if the node runs on faster hardware.

For all implementations with SJ, 5 warm-up iterations are run before timing begins. This allows the Java *Just-in-time* (JIT) compiler to optimise the code for a (marginally) better performance.

To run the implementations, the SJ application is launched in each of the nodes involved in the computation. We have put together *sessionj-tools*, a set of Perl scripts [2] to automatically resolve hosts and port numbers of each component and connect the nodes in the correct order.

5.3.2 Benchmark results

SJ-based implementation

First we present the total runtime of three implementations of n-body simulation with SJ, some of the results are shown in Table 5.2 and plotted on Fig. 5.5.

SJ A pure SJ implementation that does not involve the JNA library or acceleration hardware.

SJ + C An implementation that uses JNA library to bridge SJ with C. The main computation function is implemented in C that runs on the CPU.

SJ + FPGA An implementation that uses JNA library to bridge SJ with C. The main computation function marshals the data from SJ and forwards to and from the FPGA using DMA.

MPJExpress An implementation of the MPI standard in pure Java. This is our candidate for non-session based message passing framework. In previous comparisons in [**?**], SJ performs competitively with MPJExpress.

The graph in Fig. 5.5 shows that with an increasing number of particles, the performance of the FPGA implementation starts to be more efficient than pure SJ. Runtime for the SJ + FPGA combination overtook SJ when the total particle number is over 33000. We have discussed the importance of the complexity of the algorithm and choosing suitable problem size in §5.2.3, this is the point where the problem size is big enough for FPGA to be feasible. The best performance of

[2]Full details of the miniproject can be found on `http://www.doc.ic.ac.uk/~cn06/sessionj-tools/`

Runtime (ms)		Implementation			
		SJ	SJ + CPU	SJ + FPGA	MPJExpress
Total number of particles	17600	6534	13861	9129	**4450**
	19800	7845	15973	10064	**5393**
	22000	10037	17750	11047	**6896**
	24200	10808	19159	13801	**8708**
	28600	13354	21571	14558	**12497**
	30800	**14819**	24141	15627	15979
	33000	17644	24057	**15801**	18747
	35200	19567	23747	**16602**	22801
	37400	21246	23310	**17744**	25900
	39600	23750	25262	**18055**	27054
	41800	26743	30150	**18242**	30266
	44000	29522	32352	**19145**	30950

Table 5.2: *A partial table of results showing the crucial point when runtime of SJ+FPGA implementation overtakes SJ and MPJExpress*

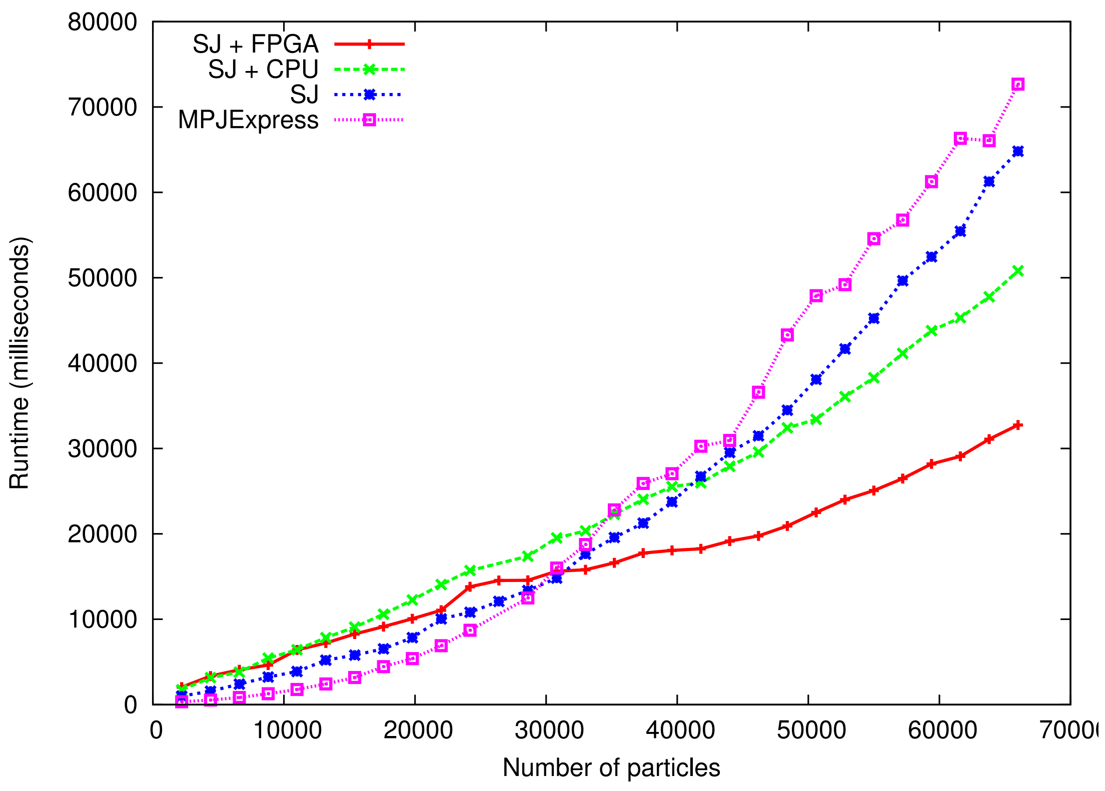


Fig. 5.5: *Runtime results against number of particles in 11 nodes over 5 iterations*

SJ + FPGA is when the number of particles reach maximum in our benchmark, which received almost 2 times speedup compared to SJ implementation. (Fig. A.1 in the appendix shows a complete graph of speedup against the same x-axis)

For the SJ + C version, the performance is worse than SJ. This is expected since the communication in SJ + C version is identical to that of SJ version, and the calculation uses identical hardware. The towards can only be explained by other minor factors such as JVM activities or network latency.

Comparison with C-translation

While the performance of the SJ + FPGA arrangement fared well with the SJ counterpart, the results of SJ implementations are dismal compared to a native C implementation as shown in Fig. 5.6. Speedup compared to SJ is 7 times maximum, and the average speedup is 5 times. Again, Fig. A.1 shows the speedup in more details.

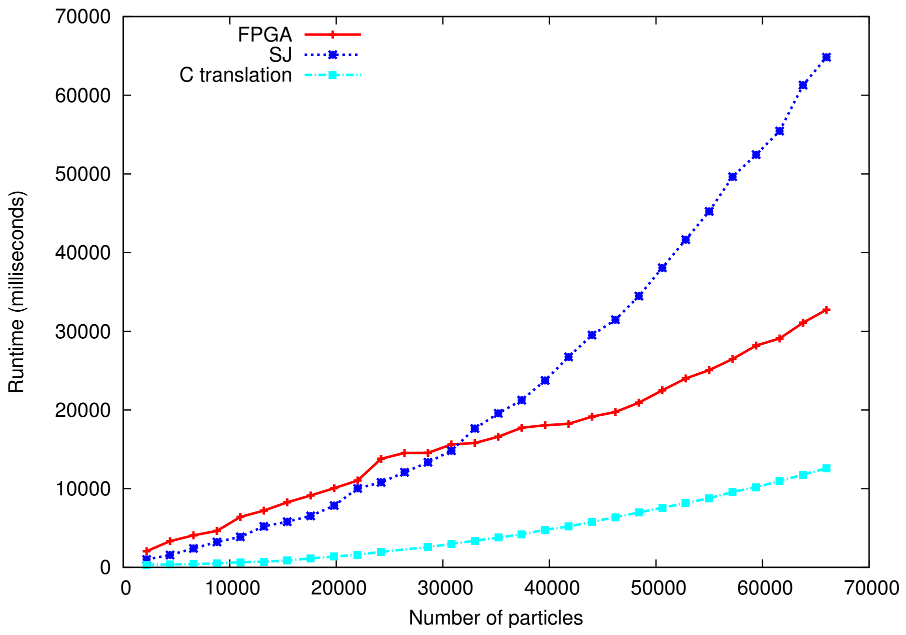


Fig. 5.6: *Comparison of SJ+FPGA and its C-translation*

We need to keep in mind again that the two programming languages are very different in terms of intended use and design, which we went into details in §3.3.1.

Sources of overhead Fig. 5.7 shows the flow of data during a call to the main computation function, `computeForces()` in SJ. Inputs from SJ are passed to the shared library as arguments to `compute_forces()` in C. Inside `compute_forces()`, inputs are written to and results read from the FPGA's memory. Then the results are forwarded back to the SJ `computeForces()`.

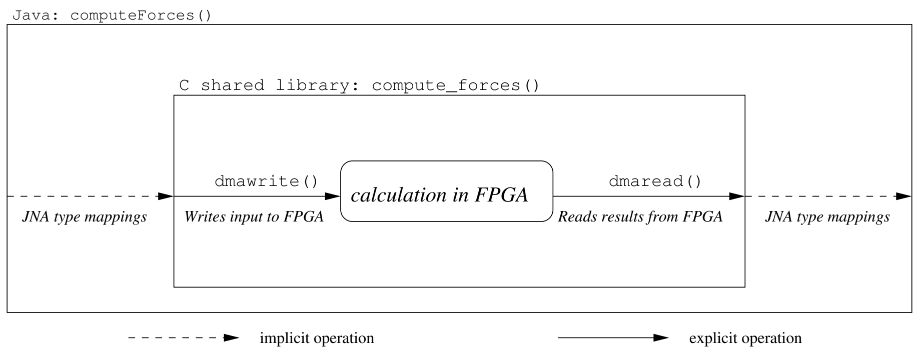


Fig. 5.7: *Flow of data between SJ and FPGA in a single* `computeForces()` *call*

In comparing the runtime of SJ+FPGA version and C-translation, the differences between the two are the time spent on conversion to and from the two languages since the communication structure of the program are identical. The conversion includes data type and format translation, as well as copying the data from JVM memory space to main memory.

Fig. 5.8 shows a comparison of time spent in the main computation (ie. `compute_forces()`), and time spent in `computeForces()` which includes all the aforementioned conversion overhead. This microbenchmark was compiled using the same configuration as in the execution of Fig. 5.6. Note that the numbers shown in the graph are average *per call* to `computeForces()`.

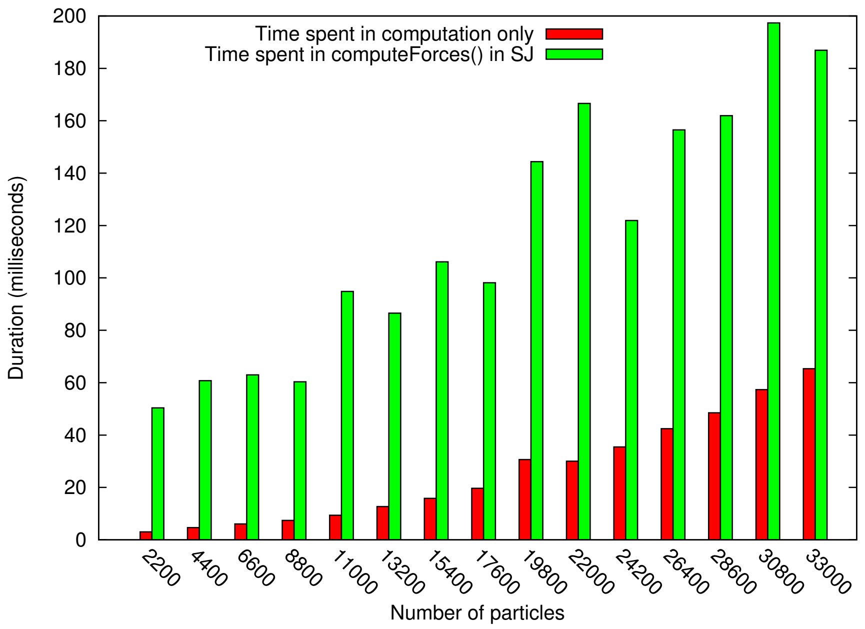


Fig. 5.8: *Graph showing the actual execution time in FPGA and the execution time from SJ's perspective*

From Fig. 5.8, the overhead (ie. differences of the bars) overshadows the time spent on computation. (green bars corresponds to outer box in Fig. 5.7, and red to inner box) Moreover, with the more particles used in the simulation, the efficiency shows a slow improvement. Interpolating the results, if we can increase the input size indefinitely, the proportion of overhead would eventually be small enough to be negligible. Whether using that input size is realistic is another question - the duration of the calculation will be very long, judging from the results that computation time with 33000 particles is about a third of the total function call time. The efficiency of this function call is definitely less than 50%.

5.3.3 Comparing with Axel's implementation

In [**?**], the implementation of n-body simulation for 81920 particles on FPGA is quoted to have $T_{comp} = 5.62s$ and CPU $T_{comp} = 99.3s$, a total of 17.7 times improvement on computation time.

While we wish to compare the performance of our implementation directly, we have a slight difference in specific algorithm details. In this project, we have chosen to use a 2D n-body simulation, ie. particles in the universe we simulate all lies on a 2D plane. This choice is based on previous SJ parallel algorithm works [**?**, **?**]. In the Axel implementation, 3D n-body simulation is used instead.

3D n-body simulation should be seen as an advantage for the FPGA implementation. As shown in Listing 2.5, calculation on an extra axis can be parallelised giving a better performance than CPU. If the parallel efficiency is 100%, we would be seeing 3 times speedup on 3D n-body with FPGA but only 2 times speedup on a 2D n-body simulation.

The speedup we can get from SJ version is The speedup of is C-translation from SJ implementation is on average 5 (See Fig. A.1). and on Fig. 5.6 the results of C-translation interpolates to about $20s$. If we take into account the 2D/3D differences above, the runtime of Axel's FPGA implementation $5.62s \times 3/2$ is still quite a lot faster than our c-translation.

It should be reminded that this implementation runs on a different configuration and runs a simplified n-body simlulation algorithm. If the *same* FPGA implementation is used, the comparison would be TCP sockets/Session sockets (c-translation) against MPI. We have shown that the Java implementation of MPI, MPJExpress compares competitively with SJ implementation. If the comparison environment is the same, it should be expected that similar results would hold for C-translations and MPI.

5.3.4 Benchmark results conclusion

In conclusion, despite the performance improvement of SJ+FPGA over vanilla SJ implementation of our n-body simulation, the biggest bottleneck of the design lies in the conversion library JNA. The translation of SJ implementation to C has eliminated the need for such *runtime* conversion library, and shows a much improved performance over versions of code that relies on Java, yet the performance still could not match implementation with MPI using the Axel toolchain.

5.4 Summary

In this chapter, we had some discussion on previously planned and alternative application structure. Next, we looked at an *inner product implementation*, which it showed that the JNA direct mapping method is a better way to build cross-language applications such as our implementation. Finally, we showed some benchmark data of SJ against different implementation of the same n-body algorithm, most notably, SJ + FPGA combination, MPJExpress - a pure Java MPI implementation, and a manual C-translation of the SJ + FPGA implementation. Both of SJ + FPGA and C-translation showed big improvements over native SJ, with SJ + FPGA at 2 times speedup and C-translation about 5 times speedup. We also identified the overhead of the design, JNA library, by comparing the duration of the main function call with the main computation time. It showed that only less than a third of the function call was performing useful computation.

Chapter 6

Conclusion

In our design goals outlined in §3.1, we stated that our main design criteria are *efficiency*, *safety* and *readability*. Using SJ, We have successfully shown that all the design goals are met:

Efficiency In our implementation of n-body simulation with SJ, the benchmark result shows with hardware acceleration of FPGA the performance of the simulation are improved. (§5.3.2)

Safety As the n-body simulation is designed in SJ, it is free from communication errors. Furthermore, we have proved that deadlocks Will not happen throughout the execution of the application from the global view of our implementation. (§4.7)

Readability We use SJ as the main design instrument for our implementations on Axel. SJ is a high level language that only exposes a minimal set of primitives for communications and makes use of object-oriented features to structure program design. §2.4

We also looked at a version of our n-body simulation translated to C. §3.4 The implementation shows potential of C as a target language for parallel designs in SJ. A SJ communication primitives library in C was developed as a result of the translation. (§3.4)

We have formalised a multichannel `inwhile` and `outwhile` construct used for designing parallel algorithms. We also derived a definition for well-formed ring topology as part of the formalisation above, and delivered a deadlock-free property for all processes under the topology. This involves a new mechanical proof technique that avoided the use of complex formalism such as global type (multiparty session type) and shared input queue [**?**, **?**] to model the multichannel `inwhile` and `outwhile` semantics.

We wish formalisation of the multichannel SJ constructs will take SJ further in the field of parallel programming, given the extra confidence of a deadlock free prove. The results of our formalisation is built upon a lot of previous and ongoing session types research [**?**, **?**, **?**]. Partial session types, sequential composition, well-formedness of process structures are all additions not found in [**?**].

6.1 Future work

- **Multiparty session types for SJ**. We have shown global communication safety with our n-body simulation as a theoretical proof. If we have multiparty session types for SJ, we would be able to show global communication safety by asserting the property from the SJ framework without a separate proof.

- **Automatic translator**. The SJ primitives library in C and the manual C translation has shown prospects of parallel design with a SJ based approach. Avoiding the huge overhead of *runtime* translation between the two language and instead providing an SJ *runtime* in C is a much preferred approach than mixing language. If further work can extend this approach and automate the translation, we can have best of session types and HPC programming.

- **Generalise approach to GPU**. Of all the three computing elements available on Axel, GPUs are the ones that we have not implemented a SJ version. As described in previous sections, our approach is *designed* to use with different hardware and implementations. This will extend SJ to a wider range of acceleration hardware.

- **Full Session C++**. X10 programming language [**?**] from IBM is a research language for parallel applications in the *Partitioned Global Address Space* (PGAS) family of languages. PGAS is a parallel programming model that essentially aggregates distributed memory to a global address space and exploits locality of reference in memory access for performance. There are some interest from the SJ community to compare the two languages [**?**, **?**]. However, an interesting feature from an implementer point of view in the language is a multi language code generator in X10 compiler. X10 can generate Java code and C++ code in their compiler[1]. SJ uses the same *polyglot compiler framework* as the X10 project, and it looks like it is possible to add a similar extension to the SJ compiler. If we are able to develop a representation of sessions in C++, and incorporate this information in the runtime environment, then we could possibly get a complete session-based C++ language.

- **Integrating SJ into heterogeneous cluster toolchain** As it stands, SJ cannot be used directly as a part of the Axel toolchain.

  The toolchain uses MPI as the inter-node communication tool. All the code for each hardware accelerator (eg. `fpga.c`, `gpu.c`, `cpu.c`) is compiled separately by their respective building tool, then linked to a single executable by the MPI compiler `mpicc`. `mpirun` is then invoked on the executable and the partitioning information (an XML configuration file we briefly mentioned in §2.5.2) is supplied as part of the arguments.

  To fit SJ into the toolchain, it is best that we use SJ as a communication tool to replace MPI. The main application should be written in SJ, but since SJ/Java cannot perform a link operation with natively compiled code, there are two possible ways to proceed:

  1. Adopt the methods described in this project and use JNA to allow SJ code interoperate with the precompiled executables. Alternatively go one step further by translating the

[1]`http://docs.codehaus.org/display/XTENLANG/X10+Compiler+Overview`

SJ code to C as in §3.4. Note that in this arrangement, the architecture described is exactly **SJ as a coordinator** we discussed in §5.1.1.

Dynamic load balancing could be a feature in this arrangement, eliminating the need to specify the partition split before execution.

2. Session C++ we have just proposed would be an ideal candidate with both *performance* and *communication safety*: A session types based language that can be used **natively** with the heterogeneous components as an inter-component coordinator, and a **communication-safe** message passing framework for inter-node communication.

Bibliography

[1] C. Austin and M. Pawlan. *Advanced Programming for the Java 2 Platform with CD-ROM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[2] Axel project website. `http://cc.doc.ic.ac.uk/projects/prj_axel/`. Accessed on 2/6/2010.

[3] A. Bejleri, R. Hu, and N. Yoshida. Session-based programming for parallel algorithms: Expressiveness and performance. In *PLACES'09*, 2009. `http://www.doc.ic.ac.uk/~ab406/parallel_algorithms.html`.

[4] A. Bejleri and N. Yoshida. Synchronous multiparty session types. *Electron. Notes Theor. Comput. Sci.*, 241:3–33, 2009.

[5] L. Cardelli and P. Gardner. Membrane computing and biologically inspired process calculi. Slides available at `http://www.lucacardelli.name/Talks/2009-12-04PiintheSky(ImperialLecture).pdf`, 2009. Accessed on 13/1/2010.

[6] P. Collingbourne and P. Kelly. Inference of session types from control flow. In *FESCA*, ENTCS. Elsevier, 2008. To appear.

[7] CUDA homepage. `http://www.nvidia.co.uk/object/cuda_what_is.html`. Accessed on 13/1/2010.

[8] M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On progress for structured communications. In G. Barthe and C. Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2007.

[9] M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, and N. Yoshida. Objects and session types. *Inf. Comput.*, 207(5):595–641, 2009.

[10] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys'06*, ACM SIGOPS, pages 177–190, 2006.

[11] libffi: A Portable Foreign Function Interface Library. `http://sourceware.org/libffi/`. Accessed on 30/5/2010.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:36–47, 1999.

[14] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381, pages 22–138, 1998.

[15] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *In Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM Press, 2008.

[16] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP '10*, 2010. To appear.

[17] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In J. Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[18] Jacuzzi homepage. `http://jacuzzi.sourceforge.net`. Accessed on 13/1/2010.

[19] JCUDA homepage. `http://www.jcuda.org`. Accessed on 13/1/2010.

[20] JNA homepage. `https://jna.dev.java.net/`. Accessed on 13/1/2010.

[21] Y. Kryftis. Session-based programming for message-passing-based parallel algorithms. Master's thesis, Imperial College London, 2009.

[22] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[23] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[25] G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan. 1998.

[26] MPI: A Message-Passing Interface Standard, Version 2.1. `http://www.mpi-forum.org/docs/mpi21-report.pdf`, 2008. Accessed on 13/1/2010.

[27] M. Neubauer and P. Thiemann. An implementation of session types. In *In PADL, volume 3057 of LNCS*, pages 56–70. Springer, 2004.

[28] M. Sackman and S. Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, June 2008.

[29] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[30] The computer language benchmark game. `http://shootout.alioth.debian.org`. Accessed on 13/1/2010.

[31] SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`. Accessed on 13/1/2010.

[32] K. H. Tsoi and W. Luk. Axel: a heterogeneous cluster with FPGAs and GPUs. In *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 115–124, New York, NY, USA, 2010. ACM.

[33] X10 homepage. `http://x10-lang.org/`. Accessed on 11/6/2010.

[34] N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In C.-H. L. Ong, editor, *FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.

[35] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Appendix A

Appendix

A.1 Java Native Interface (JNI) example

This example is a C program from the official JNI Tutorial [**?**, Chapter 5].

```c
#include <jni.h>
...

JNIEXPORT jbyteArray JNICALL Java_ReadFile_loadFile(JNIEnv * env, jobject jobj,
    jstring name)
{
   caddr_t m;
   jbyteArray jb;
   jboolean iscopy;
   struct stat finfo;
   const char *mfile = (*env)->GetStringUTFChars(env, name, &iscopy);
   int fd = open(mfile, O_RDONLY);

   if (fd == -1) printf("Could not open %s\n", mfile);
   lstat(mfile, &finfo);
   m = mmap((caddr_t) 0, finfo.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
   if (m == (caddr_t)-1) {
      printf("Could not mmap %s\n", mfile);
      return(0);
   }
   jb = (*env)->NewByteArray(env, finfo.st_size);
   (*env)->SetByteArrayRegion(env, jb, 0, finfo.st_size, (jbyte *)m);
   close(fd);
   (*env)->ReleaseStringUTFChars(env, name, mfile);
   return (jb);
}
```

Listing A.1: *Example C function that uses JNI from [**?**]*

JNI provides a complete mapping between native datatype and Java datatype, such as `jstring` and `jbyteArray`. The `JNIEnv *env` pointer is the core feature of JNI, which gives the native program access to the execution environment and data in the JVM. It also helps keeping track of references for the automatic garbage collection mechanism amongst other metadata, therefore `ReleaseStringUTFChars` is issued to notify the garbage collector before the function in Listing A.1 returns.

The *Java Development Kit* (JDK) comes with a tool `javah` to generate C header and stub files as in Listing A.1, but working with code in JNI is still cumbersome and generally considered difficult.

A.2 Java Native Access (JNA) example

```
/* gcc -shared -o libexample.so libexample.c */
int sum(int x, int y)
{
   return x + y;
}
```

Listing A.2: *Source of shared library* `libexample` *in C*

```
package libexample;
import com.sun.jna.Library;

public interface LibExample extends Library {
   int sum(int x, int y);
}
```

Listing A.3: *Java interface for* `libexample`

```
package libexample;
import com.sun.jna.Native;

public class Example {
   public static void main(String args[]) throws Exception {
      LibExample libexample = (LibExample) Native.loadLibrary(
              System.getProperty("user.dir")
              + "/libexample.so", LibExample.class);
      System.out.println("Sum is "+libexample.sum(42, 77));
   }
}
```

Listing A.4: *Java code that uses the sum function in* `libexample`

```
package SJExample;
import sessionj.runtime.*;
import sessionj.runtime.net.*;
```

```java

public class Client {
   final noalias protocol p_client { cbegin.!<int>.!<int>.?(int) }

   public void run(int port) {
      final noalias SJService svc = SJService.create(
            p_client, "localhost", port);
      final noalias SJSocket sock;

      try (sock) {
         sock = svc.request();
         sock.send(42);
         sock.send(77);
         int result = sock.receiveInt();
         System.out.println("Server replies: "+result);
      } catch(SJIncompatibleSessionException ise) {
         System.err.println("[C] Non-dual behaviour: " + ise);
      } catch(SJIOException sioe) {
         System.err.println("[C] Communication error: " + sioe);
      } finally { /* Close socket */ }
   }

   public static void main(String argv[]) throws Exception {
      int port = Integer.parseInt(argv[0]);
      Client client = new Client();
      client.run(port);
   }
}
```

Listing A.5: *SJ code similar to `Example` class in the JNA-Java example*

```java
package SJExample;
import sessionj.runtime.*;
import sessionj.runtime.net.*;

import com.sun.jna.Native;
import libexample.LibExample;

public class Server {
   final noalias protocol p_server { sbegin.?(int).?(int).!<int> }

   public void run(int port) {
      final noalias SJServerSocket svr;
      final noalias SJSocket sock;

      /**
       * Get an instance of LibExample
       */
      String abspath = System.getProperty("user.dir")+"/libexample.so";
```

```
        LibExample libexample = (LibExample) Native.loadLibrary(
                abspath, LibExample.class);

        try (svr) {
            svr = SJServerSocketImpl.create(p_server, port);
            try (sock) {
                sock = svr.accept();
                int x = sock.receiveInt();
                int y = sock.receiveInt();
                int result = libexample.sum(x, y);
                sock.send(result);
            } catch(SJIncompatibleSessionException ise) {
                System.err.println("[S] Non-dual behaviour: " + ise);
            } catch(SJIOException sioe) {
                System.err.println("[S] Communication error: " + sioe);
            }

        } catch(SJIOException sioe) {
            System.err.println("[S] Communication error: " + sioe);
        } finally { /* Close socket */ }
    }

    public static void main(String argv[]) throws Exception {
        int port = Integer.parseInt(argv[0]);
        Server server = new Server();
        server.run(port);
    }
}
```

Listing A.6: *SJ code similar to* `Example` *class in the JNA-Java example*

This is a full code listing of a JNA example, where the Java code invokes a function in a C shared library (`libexample`) to add two numbers. `libexample` is similar to SumServer/Client (Listing 2.1).

A.3 Comparison of SJ and C-translation implementation

This section compares SJ implementation and its C-translation of the same SJ implementation. There is almost a line-by-line correspondence between the two versions.

```
//
//$ bin/body left-port body-host right-port input-size

... // includes, macros

volatile uint32_t* fpgaReg;
volatile uint8_t* fpgaMem;

```

```
int main(int argc, char **argv)
{
    ... // Variable declarations

    signal(SIGPIPE, &sigpipe_handler);
    signal(SIGSEGV, &sigsegv_handler);

    prepare();

    // Protocol: cbegin.?(int).![![!<Particle[]>]*]*
    next_fd = client_socket(argv[2], atoi(argv[3]));
    // Protocol: sbegin.!<int>.?[?[?(Particle[])]*]*
    prevnode_fd = server_socket(atoi(argv[1]));
    prev_fd = accept_connection(prevnode_fd);

    // # of nodes
    recv_int(next_fd, &nr_of_nodes); // ?(int)
    ++nr_of_nodes;
    send_int(prev_fd, &nr_of_nodes); // !<int>

    size = atoi(argv[4]);

    particles = (particle_t *) malloc(sizeof(particle_t)*size);
    temp_particles = (particle_t *) malloc(sizeof(particle_t)*size);
    pvs = (particlev_t *) malloc(sizeof(particlev_t)*size);

    init(particles, pvs, size);

    outer_loop_index = 0;
    OUTWHILE(inwhile(&prev_fd, 1), &next_fd, 1) {

        // This round
        memcpy(temp_particles, particles, size * sizeof(particle_t));

        // Pump particles through the ring
        OUTWHILE(inwhile(&prev_fd, 1), &next_fd, 1) {

            // !<Particle[]>, Send particles into ring
            send_particles(next_fd, temp_particles, size);
            compute_forces(particles, temp_particles, pvs, size);
            // ?(Particle[]), Receive from the other end of ring
            recv_particles(prev_fd, temp_particles);

        }

        compute_forces(particles, temp_particles, pvs, size);
        compute_positions(particles, pvs, outer_loop_index, size);

        ++outer_loop_index;
    }

    // These are done by SJ automatically
    close_socket(prev_fd); close_socket(prevnode_fd); close_socket(next_fd);
    free(particles); free(temp_particles); free(pvs);

    finish(); // Finalise FPGA etc.

    return EXIT_SUCCESS;
}
```

Listing A.7: *C translation of SJ n-body Worker node*

```
//sessionjc − cplib : ./jna.jarsrc/nbody/Body.sj − dlib//


... // imports, package declarations

public class Body {
    //                                  #nodes
    final noalias protocol p_prev { sbegin.!<int>.?[?[?(Particle[])]*]* }
    final noalias protocol p_next { ^(p_prev) }
    NBody nbody;

    public Body(NBody nbody) {
        this.nbody = nbody;
    }

    public void run( ... ) throws ClassNotFoundException {
        ... // Variable declarations

        nbody.prepare();

        try (prevNode) {
            prevNode = SJServerSocketImpl.create(p_prev, listenPort);
            try (prev,next) {

                next = nextNode.request();
                prev = prevNode.accept();

                // # of nodes
                nodeIndex = next.receiveInt();
                prev.send(nodeIndex+1);

                particles = new Particle[size];
                pvs = new ParticleV[size];

                nbody.init(particles, pvs, nodeIndex);

                next.outwhile(prev.inwhile()) {

                    // This round
                    Particle[] tempParticles = new Particle[size];
                    System.arraycopy(particles, 0, tempParticles, 0, size);

                    // Pump particles through ring
                    next.outwhile(prev.inwhile()) {
                        next.send(tempParticles);
                        nbody.computeForces(particles, tempParticles, pvs);
                        tempParticles = (Particle[]) prev.receive();
                    }

                    nbody.computeForces(particles, tempParticles, pvs);
                    nbody.computePositions(particles, pvs, i);

                    ++i;
                }
            } catch (SJIncompatibleSessionException ise) {
            } catch (SJIOException sioe) {}
        } catch (SJIOException sioe) {
        } finally { // Close socket
            nbody.finish(); // Finalise FPGA etc.
        }

    }

}
```

Listing A.8: *SJ n-body Worker*

A.4 SJ + FPGA speedup over SJ implementation

Fig A.1 shows the speedup calculated from runtime results from Fig 5.5. We could see that the speedup increases as we increase the problem size (number of particles).

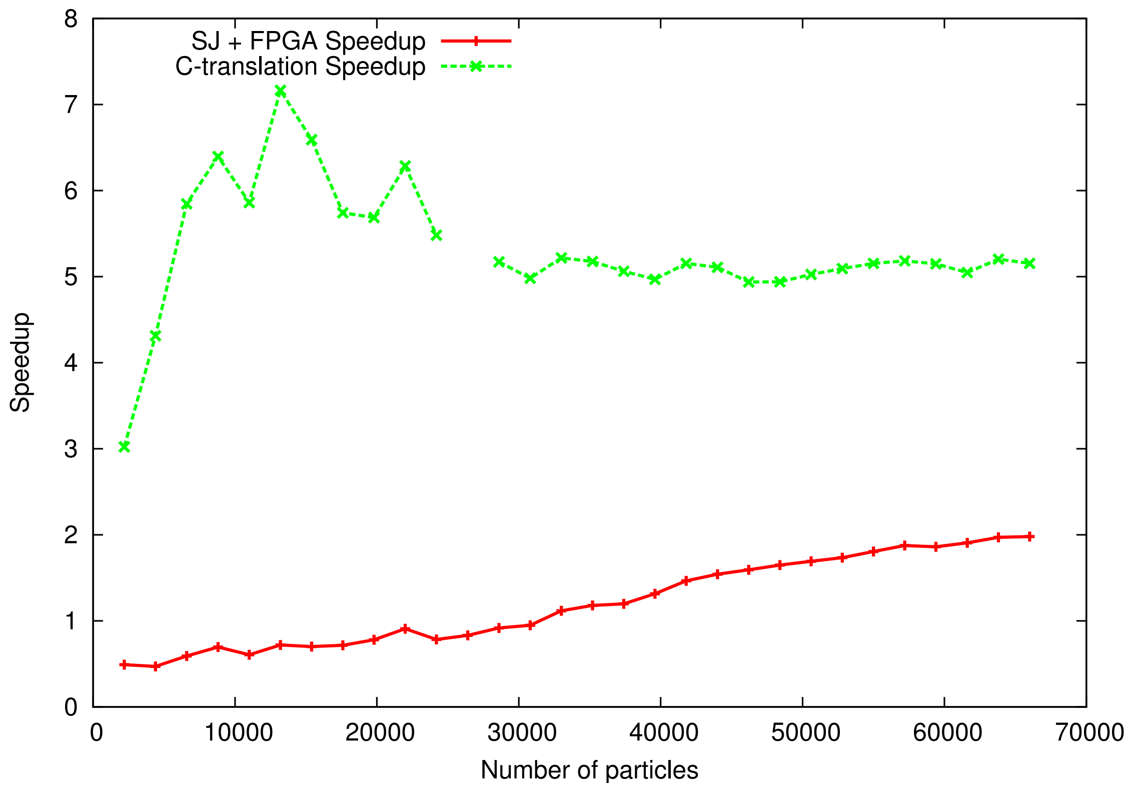


Fig. A.1: *Speedup of SJ + FPGA, C translation vs. SJ in 11 nodes over 5 iterations*