University of Hertfordshire

Data-Driven Self-Tuning in a Coordination Programming Language

by
Maksim Kuznetcov

A thesis submitted to the University of Hertfordshire
in partial fulfilment of the requirements of the degree of

MSc by Research

January 2015

Abstract

Coordination programming is a paradigm for managing composition, communication, and synchronisation of concurrent components. AstraKahn is a new dataflow coordination language based on Gilles Kahn's model of process network with some significant refinements.

AstraKahn provides a mechanism of implicit data parallelism that is expected to rely on self-tuning, i.e. adaptive optimisation of execution parameters in order to improve the performance of the program. This is achieved by providing a programmer with a number of special network primitives that allow an AstraKahn runtime system to extract optimisation parameters and adjust them while monitoring the performance of execution.

In this thesis, we present the architecture of an AstraKahn prototype including a compiler and a runtime system. On the runtime system level the built-in compound network primitives are constructed from simple ones. This approach allows us to make the implementation clear and easily extensible.

As a minor contribution we present a number of potential self-tuning heuristics for a simple network pattern. Also, for illustrative purposes, a practical application of the morphism pattern is presented. The particle-in-cell problem, whose parallelisation requires load-balancing, is formulated this way.

Contents

Chapter 1

Introduction

1.1 Coordination Programming

Coordination programming is a paradigm for managing composition, communication, and synchronisation of concurrent programs, generally called *components*. A *coordination language* is a programming language for such coordination [1].

While there are several approaches to coordination programming, in this work we focus on *dataflow* coordination, in which the execution flow is defined in terms of availability and/or mutation of data.

The approach was implemented first in the coordination language *Linda* [2] which allows several sequential components to communicate via shared associative, concurrently accessed memory storage, called the *tuple space*. The components can read, write, and remove tuples of objects from the tuple space; they can also wait for a particular tuple and resume execution as soon as it appears in the tuple store. The ideas of Linda found a new life in a recent project at Intel, called *Concurrent Collections* (CnC) [3, 4]. Like Linda, CnC uses content-addressable storage called a *tag collection* and supports dataflow synchronisation.

Another possible aggregation mechanism (or *glue* between the components) is *streaming networks*. One of the recent representatives of the approach is S-Net [5]. Here the concept of coordination is fully developed in that the application is represented as two separate programs: a collection of *boxes* with well-defined interfaces and a coordination program written in a specially constructed language that connects and coordinates those interfaces. The boxes are single-input single-output pure functions of streams written in a conventional language and oblivious to concurrency concerns. They are connected into a streaming network which (i) has a static topology where the connections are irregular, and (ii) can dynamically evolve, but only according to regular patterns where the dependencies between boxes remain statically analysable. Each box waits for an input message, performs some processing, and outputs zero or more messages into its output stream. The boxes are executed concurrently by the S-Net runtime system.

1.2 Data Parallelism

S-Net boxes process messages sequentially, one after another. However, in some cases it is possible to apply a box to several messages in parallel. In S-Net it can be implemented with an *index split combinator*: a stream splitter that specifies an integer tag and creates parallel instances of the box for every message with the unique tag (Figure 1.1).

This approach requires a programmer to write additional code for extending certain

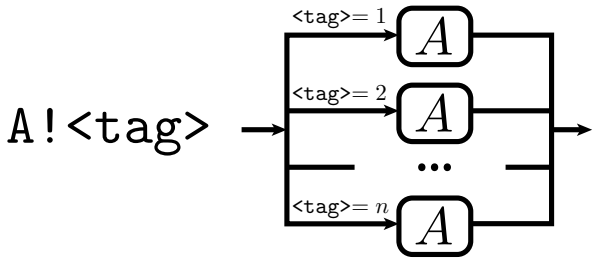


Figure 1.1: S-Net: index split combinator and the resulting network.

messages, which are eligible for parallel processing, with the tag. The number of the unique tags defines the number of parallel instances thus explicitly regulating the level of parallelism. Furthermore, the combinator does not guarantee the order of output messages. If the order is essential, the sorting must be performed explicitly.

Another approach called *concurrent box invocation* was introduced in the new S-Net runtime system Front [6]. The runtime system executes each box in a number of parallel contexts activated upon receiving input messages. The processed messages from these contexts are sent to the collector, where they may be sorted, and then to the output stream (Figure 1.2). For each box the maximum number of parallel contexts and the need for message sorting are configured manually.

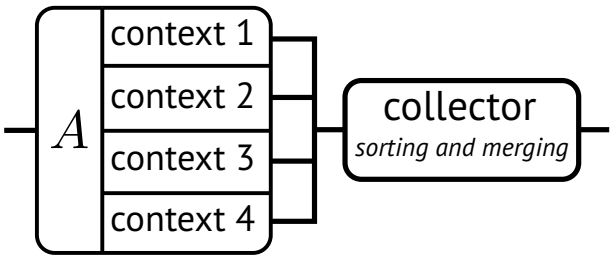


Figure 1.2: Front: a box is executed in four parallel contexts

The latter approach is less explicit, but it still requires the programmer to configure the level of concurrency for each box. The optimal settings may strongly depend on the hardware, input data, and/or dynamic properties of algorithm implemented by the network.

The number of concurrently processed messages (or *data granularity*) is also configured manually. However, it is generally unclear how many messages are sufficient to utilise the available resources fully. Too many messages may cause an overhead, while having too few of them may prevent potential pipeline parallelism.

1.3 AstraKahn

AstraKahn is a new coordination language [7] that inherits some concepts of S-Net with a number of refinements. Those include stream structuring, classes of boxes, limited capacity channels, and a well-defined protocol of box execution that prevents a box from holding its state when it is blocked. A detailed description of the language is given in Chapter 2.

AstraKahn provides an implicit kind of box parallelism called *proliferation*. It is conceptually similar to the one in Front. Unlike Front, however, the concurrency levels of boxes are expected to be set automatically based on the information collected by

monitoring the network and analysing resource utilisation.

We refer to this form of runtime adaptivity as *self-tuning*. The mechanism can also govern the granularity of data parallel problems. Using a build-in pattern called *morphism* a programmer can define problem decomposition with granularity adjusted at runtime based on the observed network performance.

The morphism pattern may also be useful for problems where a straightforward data decomposition results in imbalances of the workload assigned to different processors. Normally, specific load-balancing techniques are developed for these problems in order to benefit from data parallelism. When such techniques suffer from the compromise between the negative effect of computation imbalance and the overhead of the balancing, one can delegate the decision to the runtime system by implementing the problem using morphisms.

1.4 Contributions

The challenge of self-tuning can only be addressed with new heuristics and well-chosen assumptions about program behaviour. The first step, however, should be to implement the AstraKahn prototype as a test bed on which to run an example application while attempting several adaptation strategies. Since the language itself is not yet stable, the prototype has to be easily extensible. It is such a prototype that the present work is attempting to develop.

The main contributions of the thesis are the following

- An up-to-date definition of AstraKahn is presented (Chapter 2). While the language was first defined by Shafarenko in his preliminary report [7], during the course of research a number of amendments and clarifications were made. They include network and synchroniser description languages, practical representation of morphisms, and definition of the fixed point series;
- An implementation of AstraKahn compiler and runtime system prototype is described (Chapter 3). We discuss and motivate a number of design decisions. Although the implementation itself is still under development, the core language and runtime system components, as well as a number of extensions, are completed [8].

In addition, as a minor contribution, a number of potential self-tuning heuristics for a simple network pattern have been proposed (Chapter 4). We intend to test them as soon as the AstraKahn prototype has been implemented. Also, for illustrative purposes, a practical application of morphism pattern is presented. The particle-in-cell problem, whose parallelisation requires load-balancing, was formulated in AstraKahn using the morphism pattern (Chapter 5).

Chapter 2

AstraKahn

This chapter describes AstraKahn, a coordination language, which forms the basis of our further discussion. The language, being a successor of S-Net, was designed with a special focus on progress control and adaptive data parallelism.

This description below is based on a preliminary report [7] written by Shafarenko. Note that both the report and this chapter reflect the current state of the AstraKahn project, and that neither is guaranteed to reflect the final version that is still forthcoming.

2.1 Overview

AstraKahn is a coordination language: it treats a program as a network of computational components (or *vertices*), and defines a controlling agent called *coordinator* that is responsible for execution, communication, and synchronisation of the components. The vertices are communicating via channels carrying segmented message sequences. There are two types of vertices in AstraKahn: boxes and synchronisers:

- *Boxes* perform computations on messages. A box has a stateless function. While executing, the box reads input messages, applies its function to them, and sends the result to the output channels. Boxes can have one (in some cases – two) input channel and any number of output ones.
- *Synchronisers* can have any number of input and output channels. They also maintain an internal state: a synchroniser is organised as a finite state machine in which transitions are triggered on receiving messages. It may also have a storage for input messages. Although synchronisers do not perform any substantial computations on messages, they are used for various "housekeeping" operations such as message combining, routing, channel merging, etc.

The AstraKahn network is constructed from an algebraic wiring expression, where operands are vertices, with four operators corresponding to wiring patterns that are sufficient to construct arbitrary connection topology. Hierarchical network construction is also supported: one can declare a network as a single compound vertex and use it repeatedly in other wiring expressions. AstraKahn provides a number of built-in extensions constructed this way that, in particular, provide a programmer with transparent parallelism and self-tuning mechanisms.

In the following description AstraKahn is divided into two parts: the *core*, which includes boxes and synchroniser along with static connection operators, and the *extensions*, which are defined over the elementary vertices.

2.2 Core

2.2.1 Messages and Channels

A unit of data in AstraKahn is *message*. In the original concept the messages have types based on a term algebra defined by the Message Definition Language [7, 9]. However, in this work this characteristic of messages is not used. Hence, for simplicity, we will assume that all messages have a *record* type, i.e. can be represented as a collection of label-value pairs. We also do not restrict the type of data that can be associated with labels.

Vertices in an AstraKahn network communicate by sending messages via unidirectional FIFO channels. Each channel is shared between exactly two vertices. The channel is *input* from a standpoint of the vertex connected to its read end, and *output* for the one connected to the write end. Channels are assigned names that are used to set up connections between vertices (see Section 2.2.4).

The channels are *segmented*: they carry homogeneous sequences consisting of messages or other sequences with the nesting *depth* being the same for each message. Denote the depth by d and mark the sequences by brackets, for example:

$$((m_1\ m_2\ m_3)(m_4\ m_5))((m_6\ m_7)),\quad d = 2 \tag{2.1}$$

Since d is constant for a given channel, the number of initial and final brackets is d, whereas all other brackets occur in the following combinations:

$$\underbrace{)\ldots)}_{k}\underbrace{(\ldots(}_{k}\quad k \le d$$

We call such combinations *segmentation marks* and denote by $\sigma_k,\ k > 0$. The end of the top-level sequence is marked by a special symbol σ_0. For example, sequence (2.1) becomes

$$m_1\ m_2\ m_3\ \boldsymbol{\sigma_1}\ m_4\ m_5\ \boldsymbol{\sigma_2}\ m_6\ m_7\ \boldsymbol{\sigma_0},\quad d = 2$$

where d is a channel property and σ_k are messages of a special type that are distinct from *data messages*.

Normally AstraKahn channels are *bounded*: only a limited number of messages can be carried by it at any time[1]. The capacity of each channel is assigned individually by the coordinator. If an output channel is full, it becomes *blocked* and cannot accept further messages. The coordinator prevents situations when a vertex has to send messages to blocked channels.

2.2.2 Boxes

A box is a vertex equipped with a *pure function*: it does not depend on any external information or stored values that can be changed between different invocations, nor can it cause any side effect. In other words, boxes in AstraKahn are self-contained and *stateless*. Normally a box applies its function to input messages and sends out the result. The purity of the box function implies that the output messages do not depend on any previously processed messages. The statelessness of the box also makes it, in a sense, "ephemeral": it is neither required to have a persistent location nor to be active constantly. Since the

[1] There is one exception: a wrap-around channel, see Section 2.2.4

output of a box is fully determined by its function and incoming messages, it can be instantiated when and where needed for a given input message.

AstraKahn does not define how a box function is programmed, it can potentially be written in any programming language. There are three categories of boxes in AstraKahn identified by their behaviour with respect to input messages.

Transductor

A transductor has one input channel and one or more output channels. It produces at most one message on each output channel in response to an input message. Segmentation marks are forwarded unchanged to all output channels, i.e. the depth of the input and each of the output channels of the transductor are the same. Having a message in its input channel, a transductor runs only if all output channels are unblocked.

Inductor

An inductor also has one input channel and one or more output channels. It responds to a single message with a sequences of messages on all output channels. The inductor replaces each σ_k, $k > 0$ with σ_{k+1}, and puts σ_1 between output sequences produced from consecutive input messages. Hence, it encloses each produced sequence in a pair of brackets thus increasing the depth of the output sequence by one, for example:

$$\begin{array}{rcccl} m_1\ m_2\ (d=1) & \rightarrow & \mathrm{I} & \rightarrow & m'_{11}\ m'_{12}\ m'_{13}\ \sigma_1\ m'_{21}\ m'_{22}\ m'_{23}\ \sigma_0\ \ (d=2) \\ (m_1\ m_2) & \rightarrow & \mathrm{I} & \rightarrow & ((m'_{11}\ m'_{12}\ m'_{13})(m'_{21}\ m'_{22}\ m'_{23})) \end{array}$$

The inductor runs only if each output channel is ready to accept a message. It reads an input message and sends the first message of the result, at most one message per output channel. Then the inductor checks if it is able to continue execution. If so, the production continues. Otherwise, if some output channels happens to be blocked and the inductor has not finished its work, it generates a *continuation message*, i.e. a message which, if fed to it at a later time when the channels are unblocked, would cause the work to continue as if no interruption had taken place. The continuation replaces the current input message and the inductor stalls until it is able to run again.

Note that it is the box programmer's obligation to ensure the generation of a valid continuation after each production that suspends execution.

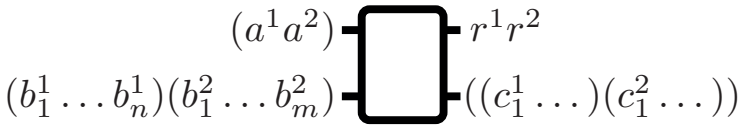
Reductor

Reductors, as opposed to inductors, turn a sequence of messages into a single message. Normally a reductor performs the following computation:

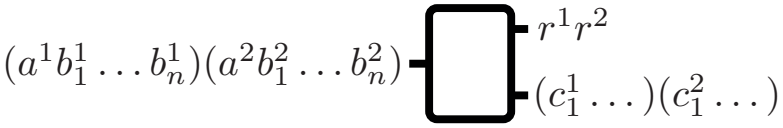
$$r = a \oplus b_1 \oplus b_2 \cdots \oplus b_n \tag{2.2}$$

where the message a is an initial term of reduction, $b_1, \ldots, b_n$ is the sequence of messages to be reduced, and $\oplus$ is a binary operator defined by a reductor function $\oplus : \tau_a \rightarrow \tau_b \rightarrow \tau_a$ where $\tau_\bullet$ is a type of the corresponding term. The expression is evaluated from left to right, hence it is essential for $\oplus$ to be left-associative.

If $\tau_a \neq \tau_b$, the reductor is *dyadic* and has two input channels, separately for a and $b_1, \ldots, b_n$. Otherwise, reductors with $\tau_a = \tau_b$ are called *monadic* and have a single input channel for all terms. Both types of reductor can have one or more output channels: the first one is used for reduction results, whereas the rest are auxiliary and could be used for messages associated with partial results of the reduction, e.g. errors, flags, etc.



(a) Dyadic reductor.



(b) Monadic reductor.

Figure 2.2: Reductors.

A reductor runs only if all its output channels are unblocked and both a and at least one of b_i are available in the input channel(s). The reductor reads a and b_i, computes an intermediate result a', optionally sends messages to auxiliary channels, and attempts to continue reduction. If either b_{i+1} is unavailable or an output channel is blocked, the reductor replaces a with the partial result a' and stalls until it is able to start the reduction again. Note that the scheme is similar to the inductor where a' plays the role of the continuation message.

The reductor treats the segmentation marks as follows: if σ_k is read in place of the initial term a, the segmentation mark of incremented depth is forwarded to all auxiliary channels:

$$\sigma_k \xrightarrow{a} \begin{cases} \sigma_{k+1}, & k > 0 \\ \sigma_0, & k = 0 \end{cases}$$

On the other hand, if σ_k appears in place of b_i, it is treated as the end of the term sequence. In this case the last a' is sent to the first output channel followed by the segmentation marks with decremented depth:

$$\sigma_k \xrightarrow{b} a' \left\{ \begin{array}{rl} \sigma_{k-1}, & k > 1 \\ \varepsilon, & k = 1 \\ \sigma_0, & k = 0 \end{array} \right\}$$

If the operator $\oplus$ is commutative and associative, i.e.

$$r = a \oplus b_1 \oplus b_2 \cdots \oplus b_n = a \oplus b_{\pi(1)} \oplus b_{\pi(2)} \cdots \oplus b_{\pi(n)}$$

for any permutation π, the reductor called *unordered*, otherwise it is *ordered*.

In the case of monadic reductor the operator can also be associative but not commutative, i.e. such that the result of expression does not depend on the placement of parenthesis. These reductors called *monadic segmented*.

For unordered and segmented reductors the reduction can be performed in parallel, see Section 2.4.2 for details.

2.2.3 Synchroniser

Unlike boxes, synchronisers are *stateful*, and do not perform any substantial computations on messages. Instead, they can store and forward messages, combine them with one another, and augment them with some simple auxiliary data.

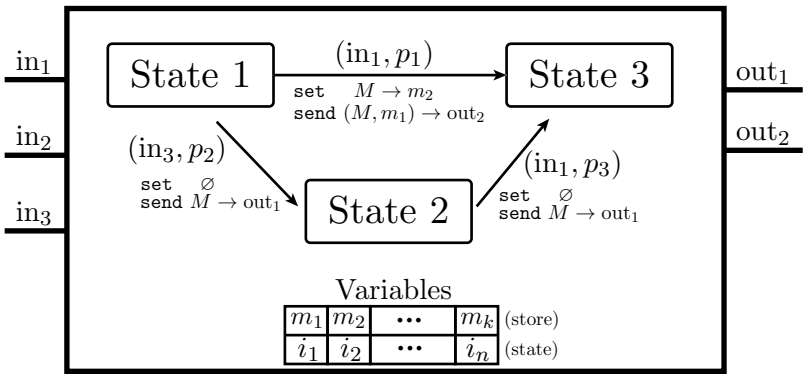


Figure 2.3: Structure of synchroniser in AstraKahn

The output of a synchroniser may depend on a potentially unlimited history of previously read input messages. This makes the synchroniser a powerful tool for various operations on message sequences coming from several channels, for message routing and construction of compound network structures.

Overview

The synchroniser is organised as a finite state machine, its high-level structure outlined in Figure 2.3. A programmer declares a number of states and transitions between them; at any moment the synchroniser is in one of the states. He can also specify a number of named registers for integer values (called *state variables*) and input messages (called *store variables*).

Each transition is associated with some input channel and is triggered upon receiving a message from the channel. A transition can also have a message pattern and a predicate defined on the content of message and state variables. Once the pattern is matched and the predicate is true, the synchroniser switches its state and executes the programmer-defined statements associated with the transition. It can

- **set** new values for store and/or state variables;
- **send** messages to one or more output channels. The synchroniser can send the currently read message, stored ones, or a combination of those, possibly augmented with integer values computed from state variables;

Synchroniser Declaration

⟨*synch_decl*⟩ ::= '`synch`' ⟨*synch_name*⟩ '`(`' ⟨*ports_decl*⟩ '`)`'
'`{`' ⟨*variable_decl*⟩* ⟨*state_decl*⟩$^+$ '`}`'

⟨*synch_name*⟩ ::= ⟨*identifier*⟩

In what follows, by *identifier* we mean a string satisfying the regular expression `[A-Za-z][A-Za-z0-9_]*`.

Ports

The synchroniser can have any number of input and output ports. Their names are declared in a header of the `synch` declaration:

⟨*ports_decl*⟩ ::= ⟨*input_port*⟩ ['`,`' ⟨*input_port*⟩]* '`|`' ⟨*output_port*⟩ ['`,`' ⟨*output_port*⟩]*

⟨*input_port*⟩ ::= ⟨*port_name*⟩ [‘:’ ⟨*depth_input*⟩]
⟨*output_port*⟩ ::= ⟨*port_name*⟩ [‘:’ ⟨*depth_output*⟩]
⟨*depth_input*⟩ ::= ⟨*integer*⟩ | ⟨*identifier*⟩
⟨*depth_output*⟩ ::= ⟨*integer*⟩
| ⟨*identifier*⟩ [(‘+’ | ‘-’) ⟨*integer*⟩]
⟨*port_name*⟩ ::= ⟨*identifier*⟩

These port names are used both externally (for wiring) and internally (in transition declarations). Each port can optionally be associated with the channel bracketing depth. If specified as an integer, a fixed constraint on the channel depth is created and used in components' reconciliation. If specified by an identifier, it becomes a global *depth constant* with the value of the actual bracketing depth. Furthermore, for output channels the depth can be constrained by an input's depth differing by a constant shift.

For example, a synchroniser with an output channel whose depth depends on the depth of an input's one has the following header:

```
synch SomeSync (foo:0, bar:d | zoo:d-1) { ... }
```

Variables

Store and state variables are declared in the beginning of the `synch`'s body

⟨*variable_decl*⟩ ::= ‘`store`’ ⟨*identifier_list*⟩ ‘`;`’
| ‘`state`’ ⟨*type*⟩ ⟨*init_decl*⟩ [‘`,`’ ⟨*init_decl*⟩]* ‘`;`’
⟨*type*⟩ ::= ‘`int`’ ‘`(`’ ⟨*width*⟩ ‘`)`’
| ‘`enum`’ ‘`(`’ ⟨*enumerator*⟩ [, ⟨*enumerator*⟩]* ‘`)`’
⟨*init_decl*⟩ ::= ⟨*identifier*⟩ [‘`=`’ ⟨*int_exp*⟩]
⟨*width*⟩ ::= ⟨*integer*⟩
⟨*enumerator*⟩ ::= ⟨*identifier*⟩ [‘`=`’ ⟨*integer*⟩]
⟨*identifier_list*⟩ ::= ⟨*identifier*⟩ [‘`,`’ ⟨*identifier*⟩]*

A state variable can be either an unsigned integer with a fixed bit-width or a C-style enumeration where the enumerators stand for integer constants. These variables can be initialised with a constant integer expression, default initial value is 0.

```
store operand;
state enum(NOP, LD, ST, ADD, MUL) opcode;
state int(32) pc = 1;
```

Store and state variables have global scope (i.e. they can be accessed from any transition) and retain their values over state transitions.

States and Transitions

⟨*state_decl*⟩ ::= ⟨*state_name*⟩ ‘`{`’ ‘`on:`’ ⟨*transition*⟩$^+$ [‘`elseon:`’ ⟨*transition*⟩$^+$]* ‘`}`’
⟨*state_name*⟩ ::= ⟨*identifier*⟩

Transitions from a state are declared inside the state body. Each synchroniser must have an initial state named `start`.

⟨*transition*⟩ ::= ⟨*port_name*⟩ [‘`.`’ ⟨*msg_pattern*⟩] [‘`&`’ ⟨*predicate*⟩]
‘`{`’ ⟨*statements*⟩ ‘`}`’

⟨*msg_pattern*⟩ ::= ⟨*bracket_pattern*⟩
| ⟨*variant_pattern*⟩
| [⟨*variant_pattern*⟩] ⟨*record_pattern*⟩
| ‘`else`’

⟨*bracket_pattern*⟩ ::= ‘`@`’ ⟨*identifier*⟩

⟨*variant_pattern*⟩ ::= ‘`?`’ ⟨*identifier*⟩

⟨*record_pattern*⟩ ::= ‘`(`’ ⟨*identifier_list*⟩ [‘`||`’ ⟨*tail*⟩]‘`)`’

⟨*tail*⟩ ::= ⟨*identifier*⟩

⟨*predicate*⟩ ::= ⟨*int_exp*⟩

Each transition is associated with some input channel and, optionally, a message pattern and a predicate. There are two kinds of pattern: segmentation mark and record [2]. An identifier in a segmentation mark pattern matches the bracketing depth, while the record pattern specifies the expected labels of an input message.

Once an input message matched some pattern, identifiers from the pattern become *local variables* (i.e. visible only within the current transition) initialised with matched values. For example

```
a.@d {...}              # Once received ))((, a local d=2 is created

a.(z, y || x) {...}     # Once received {z: 1, y: 2, w: 3, v: 4}, locals
                        # z=1, y=2, and x={w: 3, v: 4} are created
```

The predicate is an integer expression possibly containing local, state, and global integer variables. The value of the expression is interpreted as Boolean. Once the predicate is evaluated to `True`, the transition fires.

If there are more than one transition matching an input message and satisfying the predicates, one of them is chosen non-deterministically, based on a fairness policy. However, a label `elseon:` can be inserted to divide transitions into groups of descending priority. The groups are tested top-down. Once a ready transition is found in some group, it is executed and the search stops.

```
start {
  on:
    a.<p1> {...}   # if the message matches <p1> or both <p1> and <p2>

  elseon:
    a.<p2> {...}   # if the message matches <p2> only.
}
```

Transition Statements

The body of transition contains statements that are executed when the transition is triggered. Each of these statements is optional, but their order is fixed.

⟨*statements*⟩ ::= [⟨*set_stmt*⟩] [⟨*send_stmt*⟩] [⟨*goto_stmt*⟩]

[2] A *variant_pattern* specifies a variant type in MDL, which is not used in this thesis. See [9] for details.

The first statement allows one to set new values to store and/or state variables:

⟨*set_stmt*⟩ ::= '`set`' ⟨*assign*⟩ ['`,`' ⟨*assign*⟩]* '`;`'
⟨*assign*⟩ ::= ⟨*identifier*⟩ '`=`' (⟨*int_exp*⟩ | ⟨*data_exp*⟩)

The right-hand side expressions may include state, store, and local variables, as well as depth constants. The *int_exp* defines arithmetic and binary operations on integer constants and variables, and *data_exp* constructs a message of record type. The concrete syntax of the expressions will be presented later in the section.

The next statement defines sending messages to output ports. The message may be specified as a segmentation mark with an arbitrary depth or a data message of record type.

⟨*send_stmt*⟩ ::= '`send`' ⟨*dispatch*⟩ ['`,`' ⟨*dispatch*⟩]* '`;`'
⟨*dispatch*⟩ ::= ⟨*msg_exp*⟩ '`=>`' ⟨*port_name*⟩
⟨*msg_exp*⟩ ::= '`@`' ⟨*int_exp*⟩
| ['`?`' ⟨*identifier*⟩] ⟨*data_exp*⟩

Finally, the last statement defines a destination state(s) to which the synchroniser moves when the transition is taken.

⟨*goto_stmt*⟩ ::= '`goto`' ⟨*state_name*⟩ ['`,`' ⟨*state_name*⟩]* '`;`'

The statement can be omitted, in this case the synchroniser is remains in the current state. The `goto` statement can specify more than one potential destination states. In this case the synchroniser makes a single choice non-deterministically, prioritising the states where the synchroniser is more likely to trigger the next transition immediately.

Expressions

An integer expression may include standard arithmetic, binary, and comparison operators:

⟨*int_exp*⟩ ::= ⟨*iexpr*⟩
⟨*iexpr*⟩ ::= ⟨*integer*⟩
| ⟨*identifier*⟩
| '`(`' ⟨*iexpr*⟩ '`)`'
| ⟨*iexpr*⟩ '`+`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`-`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`*`' ⟨*iexpr*⟩
| ⟨*iexpr*⟩ '`/`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`%`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`<<`' ⟨*iexpr*⟩
| ⟨*iexpr*⟩ '`>>`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`|`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`&`' ⟨*iexpr*⟩
| ⟨*iexpr*⟩ '`^`' ⟨*iexpr*⟩ | '`-`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`<`' ⟨*iexpr*⟩
| ⟨*iexpr*⟩ '`>`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`==`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`!=`' ⟨*iexpr*⟩
| ⟨*iexpr*⟩ '`<=`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`>=`' ⟨*iexpr*⟩ | '`!`' ⟨*iexpr*⟩
| ⟨*iexpr*⟩ '`&&`' ⟨*iexpr*⟩ | ⟨*iexpr*⟩ '`||`' ⟨*iexpr*⟩

A data expression constructs a message of record type:

⟨*data_exp*⟩ ::= ⟨*atom_list*⟩ | '`(`' ⟨*atom_list*⟩ '`)`'
⟨*atom_list*⟩ ::= ⟨*atom*⟩ ['`||`' ⟨*atom*⟩]*
⟨*atom*⟩ ::= '`this`' | ['`'`'] ⟨*identifier*⟩ | ⟨*identifier*⟩ '`:`' ⟨*rhs*⟩

⟨*rhs*⟩ ::= ⟨*int_exp*⟩ | ⟨*identifier*⟩

The message is constructed from a sequence of the following atoms:

- A store variable;
- A key-value pair;
- The keyword `this` referring to the currently received message;
- An expansion: a store or state variable preceded by a single quote. The atom is expanded into a key-value pair where the key is named as the variable.

Putting it all together:

```
# Let the currently received message be {foo: 42}
# store k = {bar: -1}
# state m = 10, n = 100

(this || k || 'm | a: n)  # -> {foo: 42, bar: -1, m: 10, a: 100}
```

Parametrised Synchroniser

A programmer may want to parametrise some integer constants or identifiers in the synchroniser and initialise them with different values for its different instances.

In order to do this, one needs to replace actual identifiers or constants with the desired parameter's name, and put the following line before the `synch` declaration:

```
@<parameter's name> [= <default value>]
synch (...) {...}
```

When using the synchroniser in a network, a programmer will have to provide the values for its parameters (see Section 2.3).

2.2.4 Wiring

The process of connecting vertices with channels is called *wiring*. A bare vertex has a set of named placeholders for input and output channels called *ports*. When a vertex becomes a part of a network, ports are connected with correspondingly named channels. It follows that for any input and output ports to be connected by a channel, they have to be named identically.

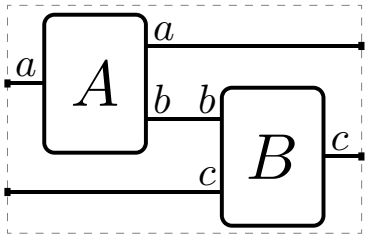
Wiring in AstraKahn is defined by an expression with vertices as operands and wiring patterns as operators. The network is constructed from the expression by its evaluation according to the rules of associativity and priority of operators.

Note that a channel can connect only two ports with the same name. If at any point of the network construction there appears a network with $n > 1$ identically named...

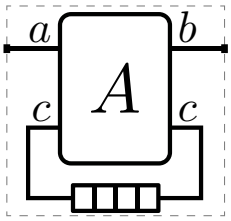
- ...input ports, they are plugged up with a 1-to-n *copier* that broadcast copies of input messages to each of the n output channels.
- ...output ports, they are plugged up with an n-to-1 *merger* that transfers all incoming messages to the single output channel non-deterministically.

AstraKahn provides three static wiring operators:

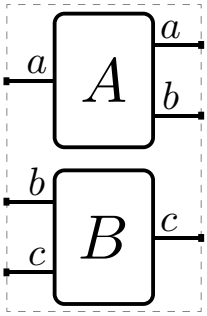
- **Serial connection** is denoted by the binary operator `..` and sets up the channels from free output ports of the left operand to identically named free input ports of the right operand;
- **Parallel connection** is denoted by the binary operator `||` and just places the operands "side by side" into a new subnetwork. The operator does not set up any new channels;
- **Wrap-around connection** is denoted by postfix operator \ and sets up the channels from free output ports of the operand to identically named free input ports of the operand. The operator creates cyclic connections. The channels of wrap-around connections are always unblocked, i.e. their capacity is limited only by the amount of available memory. This exception is made in order to avoid potential deadlocks: if such channels were of limited capacity, the network would stop execution once the channel is full and blocked.



(a) Serial



(b) Wrap-around



(c) Parallel

Figure 2.4: Wiring patterns in AstraKahn

2.3 Network Description Language

An AstraKahn program starts with a definition of the outermost net

⟨*net*⟩ ::= [‘`pure`’] ‘`net`’ ⟨*identifier*⟩ ‘`(`’ ⟨*input_ports*⟩ ‘`|`’ ⟨*output_ports*⟩ ‘`)`’
⟨*declaration*⟩*
‘`connect`’ ⟨*wiring*⟩ ‘`end`’

⟨*input_ports*⟩ ::= ⟨*identifier_list*⟩

⟨*output_ports*⟩ ::= ⟨*identifier_list*⟩

The `net` header defines its name and input and output ports. The `pure` keyword can be put before the declaration in order to specify that the net can act as a box (see Section 2.4.1).

While boxes are declared externally and have global scope, synchronisers and any nested networks (both built-in[3] and programmer-defined) are declared next to the header. The scope of these declarations is limited to the current net.

⟨*declaration*⟩ ::= ⟨*net*⟩ | ⟨*synchroniser*⟩ | ⟨*synchtable*⟩ | ⟨*morphism*⟩

The synchroniser declaration consists of its name and, optionally, the values of its external parameters:

⟨*synchroniser*⟩ ::= '`synch`' ⟨*identifier*⟩ ['`[`' ⟨*argument*⟩ ['`,`' ⟨*argument*⟩]* '`]`']
⟨*argument*⟩ ::= ⟨*identifier*⟩ '`=`' (⟨*string*⟩ | ⟨*integer*⟩)

The wiring expression is written using the operators defined in Section 2.2.4:

⟨*wiring*⟩ ::= ⟨*vertex*⟩
| '`(`' ⟨*wiring*⟩ '`)`'
| ⟨*wiring*⟩ '`..`' ⟨*wiring*⟩
| ⟨*wiring*⟩ '`||`' ⟨*wiring*⟩
| ⟨*wiring*⟩ '`\`'
| ⟨*wiring*⟩ '`*`'

The operators have properties listed below in the order of decreasing priority:

- `*` and `\` are postfix unary operators grouping from left to right, i.e.

$$a*\backslash = (a*)\backslash$$

- `..` is a binary associative operator
- `||` is a binary associative and commutative operator

In complex expressions without brackets operators `..` are evaluated from left to right:

$$a..b..c = ((a..b)..c)$$

The vertices are referred to by their names. Since boxes are declared as functions, the programmer needs to specify their category explicitly:

⟨*vertex*⟩ ::= ⟨*vertex_name*⟩ | ⟨*renaming_brackets*⟩ | ⟨*merger*⟩
⟨*vertex_name*⟩ ::= [⟨*category*⟩ '`:`'] ⟨*identifier*⟩
⟨*category*⟩ ::= '`t`' | '`i`' | '`do`' | '`du`' | '`mo`' | '`ms`' | '`mu`'

Port names are specified in declarations only for nets and synchronisers, whereas the boxes' ports by default are named `_1`, `_2`, etc., separately for input and output ports. One can change the default names by *renaming brackets*:

⟨*renaming_brackets*⟩ ::= '`<`' [⟨*renaming*⟩] '`|`' ⟨*vertex_name*⟩ '`|`' [⟨*renaming*⟩] '`>`'
⟨*renaming*⟩ ::= ⟨*identifier_list*⟩ | ⟨*kwarg_list*⟩
⟨*kwarg_list*⟩ ::= ⟨*kwarg*⟩ ['`,`' ⟨*kwarg*⟩]*
⟨*kwarg*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*identifier*⟩

For example:

[3]Syntax for synch-tables and morphisms is presented in Section 2.4.

```
<init,terms | SomeReductor | result,errors>
<_1=init,_2=terms | SomeReductor | _1=result>
```

Finally, a non-deterministic channel merger can be inserted as a wiring operand with the following syntax:

⟨*merger*⟩ ::= '<' ⟨*identifier_list*⟩ '| ~ |' ⟨*identifier_list*⟩ '>'

In the following example the merger receives messages from channels `a`, `b`, and `c`, and then copies them to `d` and `e`. As the merger is non-deterministic, no particular order of messages is guaranteed.

```
<a,b,c | ~ | d,e>
```

2.4 Extensions

AstraKahn allows one to combine boxes and synchronisers into named subnetworks, which in turn can be used repeatedly in larger networks. These subnetworks are called *nets* and also have a number of named input and output ports connected to constituent vertices.

Nets can be used not only by a programmer, but also by the coordinator in order to construct and manage built-in networked structures with a complex behaviour which comprise elementary vertices. The following sections describes such structures available in AstraKahn.

2.4.1 Pure Nets

Apart from the ordinary role of the net as a structural unit of AstraKahn networks, it can also, under some conditions, act exactly as an ordinary box in which input messages are processed by a network of vertices.

The network acting as a box is called *pure*. For the network to be pure the following conditions have to be satisfied:

- the number of input and output channels of the pure net must correspond to the ones in the box;
- the depth of all internal channels must be zero, i.e. $\sigma_k,\ k > 0$ are not allowed inside the pure net. The segmentation marks are handled by the coordinator: it forwards them to the output channel according to box's semantics and send σ_0 into the net to mark the end of the message sequence for inductors and reductors.

In order to ensure the statelessness of the pure net, the coordinator "resets" the net once it yields the result:

- for the transductor: after sending a message on all output channels;
- for the inductor: after the box has yielded a σ_0;
- for the reductor: after the box has encountered a σ_0.

The "reset" means that all internal vertices abort the execution, synchronisers are returned to their start state, and the content of all channels is flushed.

2.4.2 Parallel Boxes

Boxes process messages sequentially, one after another. However, in some cases it is possible to apply the box code to several messages in parallel. This allows one to increase

the processing rate of the box. In the following sections parallel transductor and reductor are discussed.

Transductor

A transductor simply applies its function to input messages one after another. Since the function is pure, there is no dependency between the computations. This allows one to replace a single transductor by a number of its copies and distribute input messages among them asynchronously, combining the results into the original output channels:

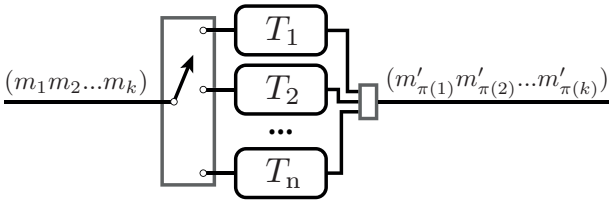


Figure 2.5: Parallel transductor.

Since transductor instances are asynchronous, the order in which messages are processed is not guaranteed. In order to preserve the semantics of a single transductor, additional components must be added after the parallel instances for retaining the original order of messages. This may reduce or cancel out the benefit of the data parallelism due to global synchronisation. However, it can be avoided if the order of messages after the transductor does not affect the computations. For example, if the transductor sends messages to an unordered reductor, the order of messages won't affect the result of reduction.

Since the topology of the network and components' properties are known statically, the compiler can mark the channels as 'ordered' or 'unordered'. This may be used for optimising out the sorting network after a parallel transductor.

That said, for the parallel transductor not to change the structure of message sequences, the messages must not be shuffled outside their brackets, i.e. a segmentation mark is always a barrier of such permutations.

Reductor

A reductor performs left-associative evaluation of expression (2.2). In the general case such evaluation is inherently sequential. However, for monadic reductors that are segmented or unordered it is possible to break the evaluation into independent computations:

$$r = b_0 \oplus b_1 \oplus b_2 \cdots \oplus b_n = (b_0 \oplus \cdots \oplus b_{p(1)}) \oplus (b_{p(1)+1} \oplus \cdots \oplus b_{p(2)}) \oplus (b_{p(m-1)+1} \oplus \cdots \oplus b_{p(k)})$$

where the monotonically increasing function $p(i)$ defines a partition of the original term sequence into k parts. The subexpressions can be evaluated in parallel, the result then can be partitioned and evaluated again until the final result is yielded. For example, for $n = 10$ and $k = 5$:

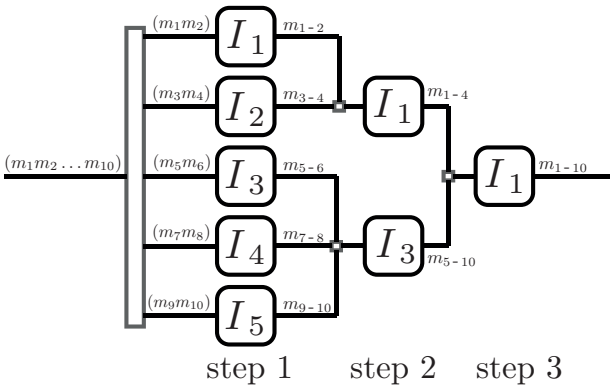


Figure 2.6: Sample message flow in a parallel reductor.

Since normally the reductor does not know the length of an incoming sequence of messages, the choice of the parameter k in general case can only be empirical. However, if the reductor is a part of a special construction (see Section 2.4.4), the coordinator knows the exact length of the sequence and can effectively allocate the resources to reduce it in parallel.

Proliferation

The parallel boxes can not be used directly by a programmer. Instead, it is the coordinator that creates parallel instances of boxes at runtime. This process is called *proliferation*. A number of parallel copies replacing a given box is called its *proliferation factor*.

Proliferation is a part of AstraKahn self-tuning strategy described in Chapter 4.

2.4.3 Synch-Table

Consider a single channel that carries interleaved homogeneous messages of several "logical" subsequences. The subsequences are distinguished by a combination of *indices*: integer values under certain labels within each message.

The *synch-table* is an extension of the synchroniser that turns it into a dynamic array of parallel instances associated with a declared set of indices contained in incoming messages. Syntactically, synch-tables are declared as follows:

⟨*synchtable*⟩ ::= '`tab`' '`[`' ⟨*index*⟩ ['`,`' ⟨*index*⟩]* '`]`' ⟨*synchroniser*⟩

⟨*index*⟩ ::= ⟨*identifier*⟩ '`[`' ⟨*integer*⟩ '`,`' ⟨*integer*⟩ '`]`'

For example:

```
tab [foo[a,b], bar[c,d] ... ] synch-declaration
```

where `foo[a,b]` means that the label named `foo` with an integer value within the limits `[a,b]` is expected to be found in an incoming message.

When the synch-table receives a message m containing all the declared indices, it dynamically instantiates the synchroniser passing the values of the indices as parameters:

```
S(foo=m[foo], bar=m[bar])
```

and routes the message to the instance. When the message has been processed, the instance persists and can accept further messages with the corresponding values of indices (Figure 3.10).

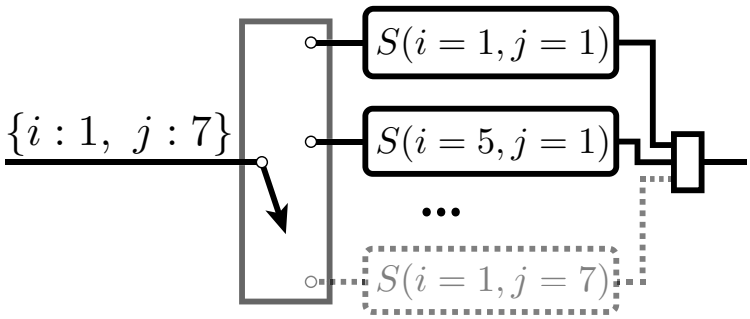


Figure 2.7: Synch-table.

Thus, each subsequence of messages is routed to an instance of the synchroniser with the indices representing corresponding the subsequence.

The output channels from all instances are merged into joint output channels corresponding to the synchroniser. Segmentation marks are broadcast unchanged to all synchronisers within the synch-table.

2.4.4 Morphisms

Since the proliferation mechanism regulates the throughput of individual vertices, it is efficient only if a sufficiently large number of messages are processed in parallel.

If the network implements a data parallel algorithm, it is possible to generate more messages by breaking down the initial problem into smaller ones by an inductor. After the messages are processed they can be assembled by a reductor to obtain the desired result.

While this pattern can easily be implemented manually, it is not generally clear how many messages are sufficient for to utilise parallel resources fully. Too many subproblems may cause a significant overhead that would cancel out benefits from the parallelism, while having too few of them may prevent potential pipeline parallelism.

However, if the program take sufficiently long time to run or is run repeatedly, the number of messages can potentially be changed automatically based on the information collected by monitoring the network and analysing resource utilisation. In order to self-tune the number of messages, AstraKahn makes it accessible to runtime system by supporting a built-in pattern for data parallel problems.

Definition (eager transductor)**.** A transductor is called *eager* if it responds to any input message with exactly one message on each of its output channels.

Consider a `1I` inductor P, an eager n`T` transductor T, and $\texttt{1R}_{M\alpha}$, $\alpha \in \{O, U, S\}$ reductors Q_i for $i \in [1, n]$. Express the boxes as mappings on messages:

$$\begin{aligned} P(k) &: m \to (m_1, m_2, ..., m_k) \\ T &: m \to r^1 \| ... \| r^n \\ Q_i(k) &: (r_1^i, ..., r_k^i) \to r, \quad i \in [1, n] \end{aligned}$$

where $\|$ separates messages on different channels. The inductor and reductors are parametrised with the number of messages they generate or reduce.

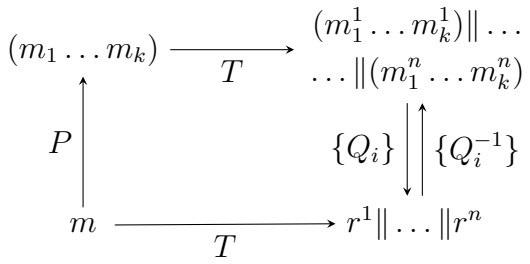
Let $Q_i(k)$ has the inverse mapping $Q_i^{-1}(k)$. In terms of boxes the inverse mapping can be defined by a `1I` inductor:

$$Q_i^{-1}(k) : r \to (r_1^i, r_2^i, ..., r_k^i)$$

Definition (loxomorphism)**.** The triplet $(P(k), T, Q(k))$ is called *loxomorphism* if

$$T \circ P = \left(\|_{i=1}^{n} Q_i^{-1}(k)\right) \circ T$$

or, equivalently, the following diagram is commutative:



The diagram can be interpreted as follows. Let an input message m to be processed by the transductor yield the result $r^1\|...\|r^n$. Instead of this the loxomorphism allows one to split the input message into a sequence $(m_1, m_2, ..., m_k)$ which can be processed by the transductor in parallel. The resulting sequence is the image of the intended result under $Q_i^{-1}(k)$ and can be reduced to it by $Q_i(k)$.

A loxomorphism in AstraKahn results in the following network:

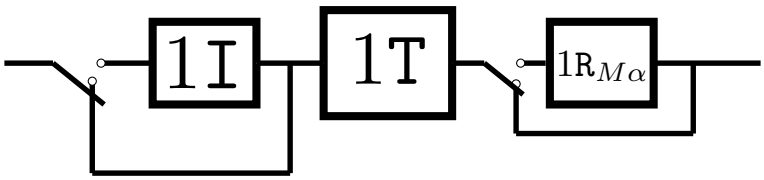


Figure 2.8: Morphism network

The coordinator can increase the level of concurrency at runtime by enabling the inductor and reductor for the given parameter k defining the number of messages that will be processed in parallel by the transductor. This process is called *fragmentation*.

The morphism is declared as follows:

⟨*morphism*⟩ ::= '`morph`' '`(`' ⟨*identifier*⟩ '`)`' '`{`' ⟨*morph_list*⟩ '`}`'
⟨*morph_list*⟩ ::= ⟨*morph*⟩ ['`,`' ⟨*morph*⟩]*
⟨*morph*⟩ ::= ⟨*inductor*⟩ '`/`' ⟨*transductor*⟩ '`/`' ⟨*reductor*⟩ '`;`'
⟨*inductor*⟩ ::= ⟨*vertex_name*⟩
⟨*transductor*⟩ ::= ⟨*vertex_name*⟩
⟨*reductor*⟩ ::= ⟨*vertex_name*⟩

The declaration assigns inductor-reductor pairs to a single transductor. Once the transductor is used in the wiring expression, the corresponding morphism will be constructed by the compiler.

An important special case of loxomorphisms is the pipeline of transductors. If two consecutive stages of the pipeline are fragmented, the messages go through the adjacent reductor and inductor which causes global synchronisation after the first stage. In the general case this step is essential since the fragments of the original problem may need to be combined with others to obtain the intended result. However, for some problems this process can be performed with some degree of locality: each message may need to be

combined with only a few others. For these cases the morphism provides special input and output ports that act as a "shortcut" for messages. Given two serially connected morphisms, the programmer can place some vertex (expectedly – a synch table) called *override*, which combines messages in the way that they may go through without the explicit join-split (Figure 2.9).

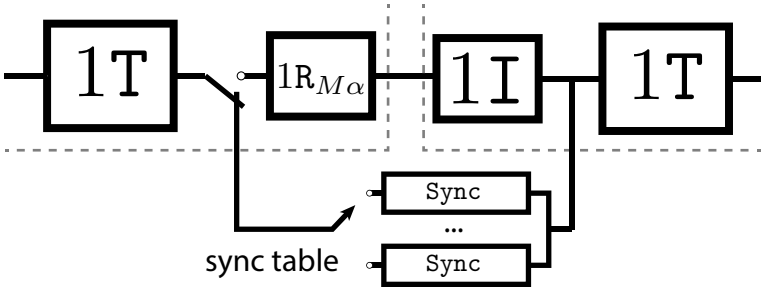


Figure 2.9: Pipeline of morphisms

2.4.5 Serial Replication

An algorithmic loop can be implemented in an AstraKahn network as a serially connected sequence:

$$\underbrace{A\,.\,.\,A\,.\,.\,\cdots\,.\,.\,A}_{n}$$

where A is a net representing one iteration of the loop, and n is a desired number of iterations. In what follows we will refer to an individual A as a *stage*.

Such a construction does not support loops iterating a variable number of times during the execution. Furthermore, the static sequence of networks takes computational resources even if there are no messages to process.

The serial replication wiring pattern in AstraKahn solves both problems. The pattern creates a "lazy", potentially infinite pipeline: the next stage is created only at the moment that the message is emitted by the last stage. Messages leave the pipeline once they satisfy the exit-condition programmed in each stage.

Let A be a net with the same number of input and output channels, and there is a bijection between their names. Assume also that the stage includes one or more synchronisers. We will refer to the aggregate state of these synchronisers as the *stage state*.

Definition (forward fixed point)**.** The stage A is said to have a *forward fixed point* (FFP) if it is possible to prove at runtime that an input message from some input channel will pass through the stage unchanged to the identically named output channel.

Since the definition is very broad, let us provide some particular cases of the forward fixed point. First, an FFP cannot pass through any boxes since they are treated as black boxes, i.e. their behaviour cannot be analysed. Hence, for an FFP it is only possible to pass through synchronisers, whose behaviour can be analysed, and mergers. For example, the FFP can be detected by some predicate that is tested by stage's synchronisers such that the message passed unchanged if the predicate is true. Alternatively, a stage can have an input channel that is directly connected to the identically named output channel: in this case once a message is know to be an FFP once it goes to this input channel.

Definition (reversed fixed point)**.** The stage A is said to have a *reversed fixed point* (RFP) if it has a subset of reachable states (called the *RFP states*) such that in each of these states the stage passes all messages unchanged to the identically named output channels, remaining within the subset of states.

An application of these stage properties to the serial replication pattern is the following. If the described stage A has both FFP and RFP and happened to occur in a wiring expression as `A*`, the "lazy" pipeline of identical replicas of A is created: a new stage is initialised once a message is processed by the current last stage.

If a message is an FFP, it leaves the pipeline through one of the separate output channels connected to the rest network (otherwise the message would infinitely create new stages bypassing through them unchanged. In a sense, the FFP "warps through" the infinite chain of the stage replicas).

On the other hand, RFP serves for reducing the length of the pipeline: once a message triggers one of the RFP states at some stage, the stage is removed by the coordinator since now it is simply passed the input messages unchanged.

Therefore, with the FFP and RFP specified appropriately, serial replication allows to create dynamic pipeline structure corresponding to loops with variable number of iterations.

Chapter 3

Implementation

As a proof-of-concept and a framework for further research and experiments, a prototype AstraKahn compiler and runtime system have been implemented. This chapter provides some details on design of the implementation.

The architecture of the prototype is given in Figure 3.1. The compiler compiles an AstraKahn program down to a network of *runtime components*. Each component is an object (in terms of object oriented programming) that corresponds to some AstraKahn vertex by implementing unified communication and execution interfaces as appropriate. Then the runtime system runs the network, managing communication and parallel execution of the components.

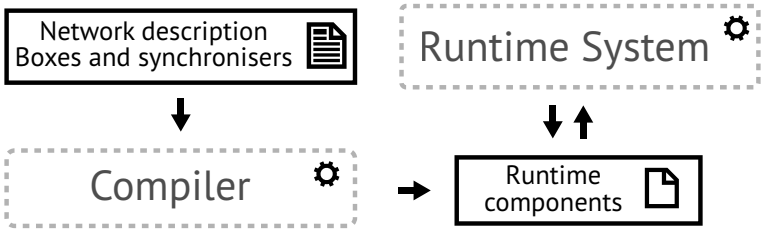


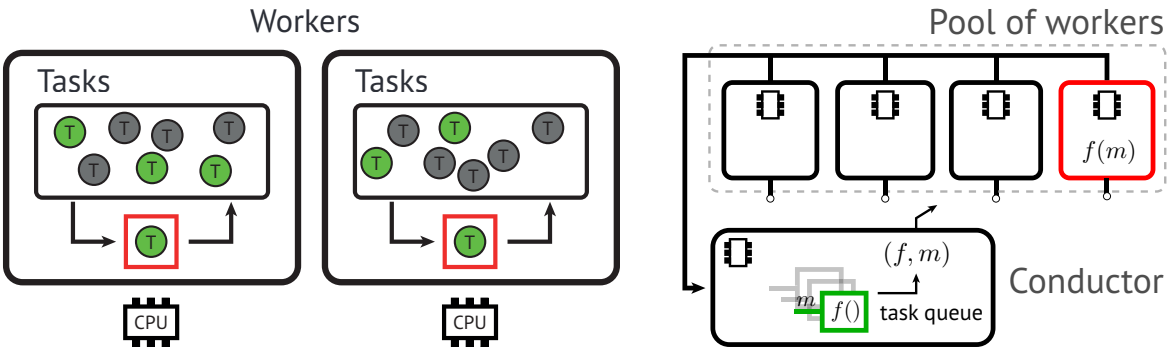
Figure 3.1: Architecture of the AstraKahn implementation.

3.1 Runtime System

3.1.1 Overview

A network of concurrent communicating vertices can be implemented in various ways. A straightforward one is to represent the vertices as kernel-level threads or processes communicating via shared memory or other inter-process communication primitives. This approach was employed in the original S-Net runtime system [10]. The drawbacks of the strategy are the lack of scheduling flexibility (since threads are governed by the OS) and the high overhead caused by expensive context switching and network reconfiguration.

The drawbacks can be avoided by using lightweight user-level threads. The LPEL, a runtime layer for S-Net [11], uses this strategy as follows. Vertices of an S-Net network are represented as *tasks* distributed among a fixed number of *workers*, which are persistent kernel-level threads. The tasks can communicate within and across workers. Once a task is ready, it is executed in the context of its worker (Figure 3.2a).



(a) The LPEL architecture (b) LPEL with a dedicated conductor

Figure 3.2: Runtime systems based in lightweight user-level threads.

However, in order to run efficiently, the LPEL has to maintain the load-balance of the workers by task migration. A strategy of dynamic task migration for the LPEL is discussed in [12]. That said, the problem is rather challenging, and it seems to be an overkill to incorporate it into a research prototype. Another approach suggested in [13] is to assign one worker called *conductor* to maintain a *central task queue* (CTQ) and dispatch ready tasks to the *pool* of workers for execution (Figure 3.2b).

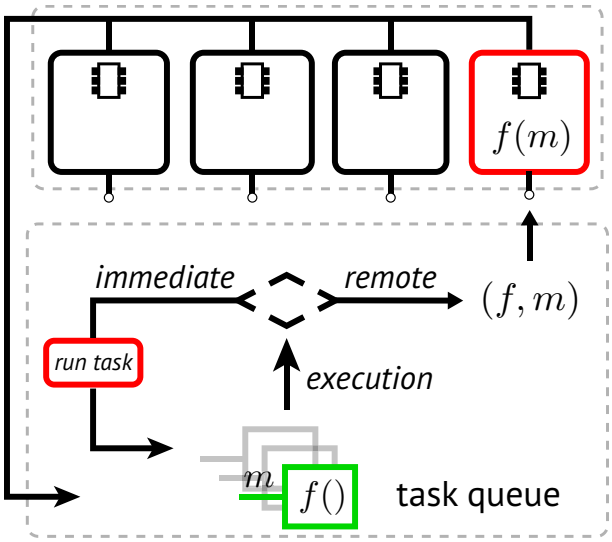


Figure 3.3: The runtime system of AstraKahn.

An immediate drawback of this approach is the communication load between the conductor and the workers on each execution of a vertex. However, we assume the task size to be small: messages usually contain lightweight data such as integers and references to heavier data objects rather than the objects themselves. Hence, to the first approximation we can neglect the inefficiencies associated with communication between the conductor and the workers. Another potential drawback is bad cache behaviour: closely connected vertices can be assigned to different workers (and, possibly, to different hardware cores) while taking turns to process the same message. We also neglect the drawback in the current implementation; however, the runtime system may account for it by employing a prioritised policy of scheduling vertices among workers.

We refer to the aforementioned method of execution as *remote*. The method is used for boxes: their functions and input messages are combined into tasks and asynchronously sent to the workers.

However, in some cases the remote execution is redundant and may cause a significant overhead:

- A segmentation mark being received by a box bypasses its functions and is forwarded to its output channels, possibly with amended depth;
- A synchroniser changes its state on receiving a message, which also requires a constant time. Furthermore, while choosing a state transition, the synchroniser needs to know the status of its input and output channels, which may change while the synchroniser is executing in the pool.

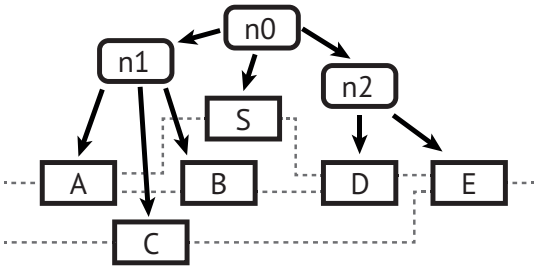
In these cases the conductor executes the components *immediately*, within the context of the conductor's process (Figure 3.3).

3.1.2 Runtime Components

Runtime components are objects that correspond to vertices and nets. The objects are structured in a tree that reflects the hierarchy of an AstraKahn program: its internal nodes are nets, and leaves are vertices (Figure 3.4). The tree is constructed by the AstraKahn compiler (see Section 3.3).

```
net n0
  net n1
    (A || C) .. B
  net n2
    D .. E
(n1 || S) .. n2
```

(a) Network description (simplified).



(b) Tree of runtime components

Figure 3.4: Correspondence between AstraKahn program and runtime components

The components have common interfaces for communication and execution.

Communication

Each component has a number of named input and output ports that denote connection endpoints (see Section 2.2.4). However, the implementation does not consider a channel as a separate entity. Instead, a bounded buffer is embedded into each input port. From the standpoint of an output port a connection is represented as a reference pointing to the buffer of some input port.

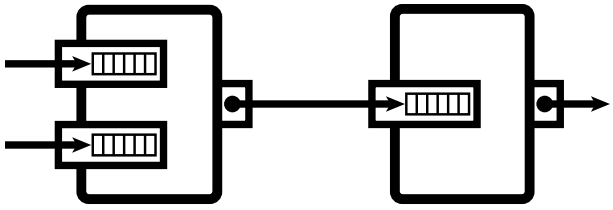


Figure 3.5: Implementation of AstraKahn channels.

The buffer is implemented as a double-ended queue, i.e. such that allows one to add or remove elements from either the head or tail. The resulting capability of components to insert messages into their input channel is used by reductors and inductors for continuation messages (see Section 3.2.1).

In AstraKahn a net is a shorthand for some subnetwork, it neither receives messages nor computes anything *on its own*. That said, runtime components representing nets contain explicit placeholders for inputs and output ports. Normally, these ports are simply references to (or, in fact, "proxies" for) corresponding "real" ports of net's constituent components. (Figure 3.6).

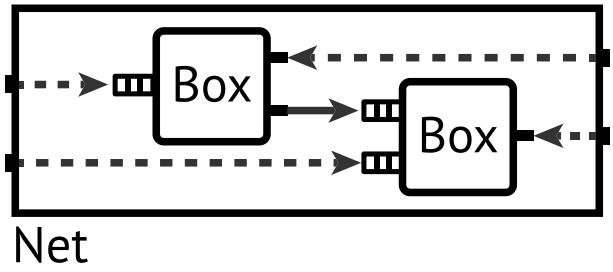


Figure 3.6: Ports of a net that is not involved in execution. Dashed grey lines denote references to ports, solid ones – references to input buffers.

However, the implementation allows some special built-in types of nets to receive and process messages on their own. Components of this type have a predefined *handler* that is called by the conductor once a messages is received by one of the net's "real" input ports. The handler has access to the net's internals, for example, it can add, remove or rewire any vertices inside the net. It also can forward the received messages to any input ports of constituent vertices (Figure 3.7).

Note that these non-standard nets are preprogrammed and can only be used for internal purposes. For example, such a net is used for serial replication wiring (Section 3.3.2).

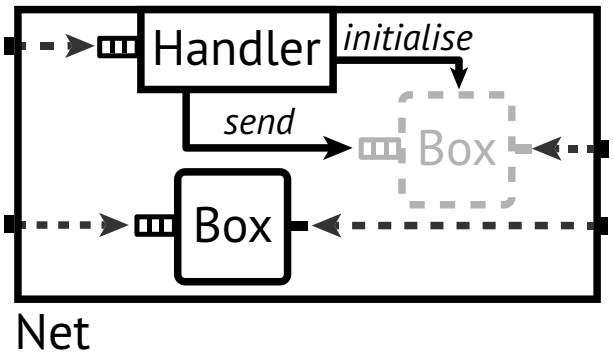


Figure 3.7: Ports of a net involved in execution.

Each node also has a private interface containing methods for fetching and sending messages, and checking the statuses of input and output channels. These methods are used by the execution interface.

Execution

An execution interface of runtime components consists of the following methods:

- `is_ready()` – returns a tuple of two Boolean values (R_i, R_o) where R_i indicates availability of input message(s) required to run, and R_o indicates whether the output channels are unblocked. The values (`True`, `True`) are returned when the component is ready for execution.
- `fetch()` – fetches and returns input message(s) required for execution.

- `run()` – performs execution of the component. The method returns either a task for remote execution or `True` if immediate execution is performed.
- `commit()` – accepts a result computed in the pool and sends it to the specified output port(s).

The interface is abstract and implemented by each vertex individually. Concrete interface implementations for all types of vertices are given in Section 3.2.

3.1.3 Scheduling

In order to run a network of components, the runtime system performs *dynamic scheduling*, i.e. computes a set of *ready* components at runtime. More specifically, the runtime system implements *data-driven scheduling*, where a component is considered *ready* if it has input message(s) to process. This approach is easy to implement, suitable for exploiting parallelism of asynchronous components and, provided that all channels are bounded, results in fair network execution [14]. The same approach was also used in the LPEL [11].

Therefore, the problem of scheduling boils down to effective computation of the set of ready components at runtime. Before executing a network, the conductor computes an initial set its ready components, and then updates it as follows. After each component's execution the conductor gets a set of components from which the messages were processed, and components to which output messages were sent. These components are candidates to become ready due to unblocked output channels or newly available input messages respectively. As soon as the set of ready components becomes empty, new candidates that happen to be ready are placed in it.

3.1.4 Network Execution

After computing an initial set of ready components, the conductor starts a loop that performs the following high-level steps:

1. pop ready component and run it immediately, or dispatch it to pool;
2. update set of ready components;
3. read results of remote execution from pool and commit them;
4. update set of ready components; if no component is ready, wait for further results from pool and repeat step 3; otherwise go to step 1.

Note that the pool is communicating concurrently with the conductor: dispatching a task to the pool is always non-blocking, and an operation that reads results from the pool is blocked only if there is no ready component.

While message communication and immediate execution are performed non-concurrently in the above loop, the overhead is assumed to be small compared to the running time of boxes executing in the pool. Nevertheless, when implemented in the runtime system, these operations are top candidates for profiling and optimisation.

3.2 Runtime Components

This section describes how AstraKahn vertices are implemented in terms of the runtime components. Elementary vertices are represented by single components with an execution interface declared appropriately, whereas built-in extensions are represented by nets composed of boxes and synchroniser defining the desired behaviour.

3.2.1 Boxes

Transductor

A transductor has the simplest execution interface (Algorithm 1):

Method `is_ready()`
/ Test if there is an input message and if all output channels are unblocked. */*
Properties: input */* input port */*
outputs */* array of output ports */*

R_i = `bool(`*is* input *not empty*`)`
R_o = `bool(`*does* outputs *have no blocked channels*`)`
return (R_i, R_o)

Method `commit(result)`
/ Return a task combined of the box function and the input message. If the input message is a segmentation mark, it is immediately sent to all output channels instead. */*
foreach (msg, port) **in** result **do**
send msg to port
end

Method `run(msg)`
/ Send output messages received from the pool to corresponding channels. */*
if msg *is* σ_k **then**
send σ_k to each port in outputs
return None */* Immediate execution */*
else
return (**self**.function, msg) */* Remote execution */*
end

Algorithm 1: Execution interface of transductor.

Inductor

In the execution interface of an inductor (Algorithm 2) `is_ready()` checks for two message slots available on each output channel since `run()` sends an additional σ_1 between output sequences if there is no other segmentation mark between them. Incoming segmentation marks are forwarded with the depth increased by one. When a data message is received, new elements of the output sequence (no more than one message per output) are computed in the pool and, if the elements are not final, a continuation message is inserted in the result. When accepted by `commit()`, the continuation, if any, is put back to the input channel, and the computed messages are sent to the output channels.

Method `is_ready()`
Properties: input */* input port */*
outputs */* array of output ports */*

R_i = `bool(`*is* input *not empty*`)`
R_o = `bool(`*can each port in* outputs *accept two messages*`)`
return (R_i, R_o)

Method `commit(result)`
Properties: segflag */* flag indicating the end of previous sequence generation */*

if result *contains a continuation* **then**
push result.continuation back to input
else
segflag = **True**
end
foreach (msg, port) **in** result **do**
send msg to port
end

Method `run(msg)`
Properties: inputs */* array of input ports */*,
segflag */* flag indicating the end of previous sequence generation */*
outputs */* array of output ports */*, function */* box function */*

if msg **is** σ_k **then**
$$\sigma_s = \begin{cases} \sigma_{k+1}, & k > 0 \\ \sigma_0, & k = 0 \end{cases}$$
send σ_s to each port in outputs
segflag = **False**
return None
else
if self.segflag **then**
send σ_1 to each port in outputs
segflag = **False**
end
return (function, msg)
end

Algorithm 2: Execution interface of inductor.

Dyadic Reductor

Consider an execution interface for a dyadic reductor (Algorithm 3). As stated in Section 2.2.2, we refer to all but the first output channels of the reductor as *auxiliary*.

- `is_ready()` requires unblocked output channels with the first one being able to accept two messages: when the reductor yields the result, it sends it to the first channel followed by a segmentation mark. Messages in both input channels are also required for the box to start;

- `fetch()` gets messages from both input channels, for reduction terms a and b_i in (2.2) respectively;
- `run()` tests the messages for being segmentation marks first. A σ_k in place of m_a results in reduction of the empty sequence and, adjusted appropriately, the segmentation mark is sent to the auxiliary channels. On the other hand, a σ_k in place of m_b indicates the end of the reduction sequence and results in sending the m_a (i.e. the final result of the reduction) to the first output channel followed by the appropriately adjusted segmentation mark;
- `commit()` implements the continuation mechanism: a partial result (or *accumulator*) of reduction plays the role of a continuation message. The method pushes it back to the first input channel and sends out auxiliary messages, if any.

The execution interface of a **monadic reductors** is defined in a similar way.

3.2.2 Synchronisers

Overview

First, the runtime component for a synchroniser maintains a number of *state* objects, one of which is marked as current. Each state object in turn consists of *transition* objects, each of which imposes a condition on input messages. The condition consists of three parts:

- the **input channel** from which a message has to be received;
- the **pattern** of labels that has to match the message. Keyword `.else` can be set instead of the pattern to indicate a transition that is taken if no other transitions' pattern matched in the current transition scope;
- a **predicate**: a Boolean expression that can include labels from the pattern and state variables.

Each transition can also have the following augmented actions:

- to `set` internal state of store variables with some integer values or messages;
- to `send` messages (or their combinations) to output channels;
- to `goto` the new state(s).

Within each state the transitions are also grouped in *blocks*, each having a priority denoted by $P = \overline{1..N_B}$: the smaller the number, the higher priority of the transitions.

The execution interface of the synchroniser is presented in the sequel. Pseudo-code listings for the discussed methods can be found in Appendix A.

`is_ready()`

A synchroniser is ready to run when the following requirements are satisfied:

- there is at least one input message on a channel for which there is at least one transition in the current state;
- destination channels in `send` statements associated with the potential transitions are unblocked. In other words, a transition being taken must not cause message dispatch to a blocked channel.

The method `is_ready()` tests these requirements. First, it computes a set of input channels that can trigger a transition from the current state. If these channels do not have messages, the negative decision is returned. Otherwise for each of the channels the

Method `is_ready()`
Properties: inputs */* array of input ports */*
outputs */* array of output ports */*

R_i = `bool(`*does* inputs *have non-empty ports*`)`
R_o = `bool(`*is each port in* outputs $[1..n_o]$ *unblocked*`)`
&& `bool(`*can* outputs $[0]$ *accept two messages*`)`
return (R_i, R_o)

Method `fetch()`
Properties: inputs */* array of input ports */*

m_a = inputs$[0]$.`get()`
m_b = inputs$[1]$.`get()`
return (m_a, m_b)

Method `commit(result)`
Properties: inputs */* array of input ports */*
outputs */* array of output ports */*

push result.accumulator back to inputs$[0]$
foreach (msg, outputs$[k]$, $k > 0$) **in** result **do**
send msg to outputs$[k]$
end

Method `run(`m_a, m_b`)`
Properties: inputs */* array of input ports */*
outputs */* array of output ports */*
function */* box function */*

if m_a *is* σ_k **then**
$$\sigma_s = \begin{cases} \sigma_{k+1}, & k > 0 \\ \sigma_0, & k = 0 \end{cases}$$
send σ_s to outputs$[1..n_o]$
return None

if m_b **is** σ_k **then**
send m_a to outputs$[0]$
if $k > 1$ **then**
$$\sigma_s = \begin{cases} \sigma_{k-1}, & k > 1 \\ \sigma_0, & k = 0 \end{cases}$$
send σ_s to outputs$[0]$
end
return None
end
return (function, m_a, m_b)

Algorithm 3: Execution interface of reductor.

method collects potential transitions that have `send` statements. If at least one `send` is targeted at a blocked channel, the input channel is removed from the set. At the end, if the set of ready input channels is not empty, it is stored for further use and the synchroniser is marked as ready.

`fetch()`

The method operates on the set of ready input channels computed previously. When there are more then one channel in this set, the choice has to be made non-deterministically by some fairness policy. In the implementation the method counts how many times each input channel was taken, and the choice is made in favour of the least frequently taken one. Then, message is fetched from the chosen channel.

`run()`

Once a message is fetched from some input channel, there may be more then one transition declared for it. In this case the transition is chosen by the following procedure:

- The transitions are tested by blocks starting from the one of highest priority;
- For each transition in the block the pattern and predicate are applied to the message. If the message passes the test, the transition is added to a *valid* set with a `.else` transition marked separately, if any.
- Finally, if there are no valid transitions except for the `.else` one, then it is taken. If there are more than one such transitions, the least frequently taken one is chosen. If there are no valid transitions at all, the next block is tested.
- If no transition has been chosen after testing all the blocks, the synchroniser drops the message and terminates.

Once a transition is taken, its associated actions are performed. There can be a situation when several new states are specified in the `goto` statement. The preferable ones are those in which the synchroniser is immediately ready to start. In any case, if there are several states to choose, the least frequently taken one is chosen.

3.2.3 Mergers and Copiers

As stated in Section 2.2.4, connections between a single output port and several input ones (and vice versa) are supported by vertices of special types: copiers and mergers.

A **merger** is a vertex that reads messages from several input channels and then sends them to a single output channel in no particular order, i.e. non-deterministically. This behaviour is implemented by a synchroniser, which operates non-deterministically when more than one transition is available:

```
synch merger (in1, in2, ... | out) {
  start {
    on:
      in1 { send this => out; }
      in2 { send this => out; }
      ...
  }
}
```

The code can easily be extended and generated by the runtime system for any given

number of input channels.

A **copier**, by contrast, reads messages from a single channel and copies them to each output ports. The vertex is implemented as follows:

```
synch copier (in | out1, out2, ...) {
  start {
    on:
      in {
        send this => out1,
             this => out2, ...;
      }
  }
}
```

3.2.4 Extensions

Built-in AstraKahn extensions are constructed from a number of boxes, synchroniser, and, possibly, preprogrammed executable nets (Section 3.1.2, Figure 3.7). In this section some examples of such constructions are presented.

Pure Nets

Recall that pure nets are subnetworks that behave exactly as AstraKahn boxes. In a sense, this concept generalises a box function by permitting it to be any subnetwork that meets certain requirements (Section 2.4.1). In terms of runtime components this generalisation can easily be implemented by extending a net component with the execution interface of the box category it supposed to emulate. An example of such a construction is given in Figure 3.9. The only difference between the approach and the conventional execution interface is that `run()` sends a message to the pure net instead of the pool, and `commit()` is a net handler that is called when the message is processed by the pure net.

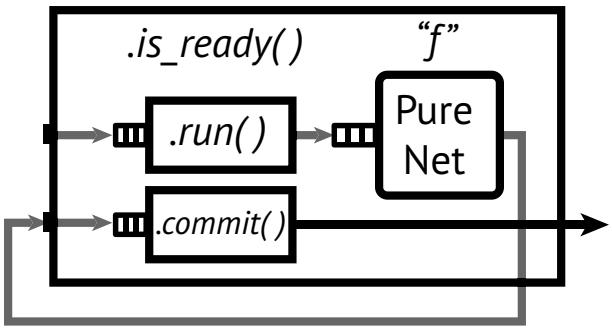


Figure 3.8: A generalised box with a pure network.

Parallel Boxes

Once proliferation is applied to a transductor, it is replaced by a net in Figure 3.9. It is a non-standard net since it has two control methods (denoted by $+/-$ in the figure) that allow the runtime system to change the proliferation factor. Apart from n parallel copies of the transductor, the net contains three synchronisers:

- **Splitter** distributes the incoming messages uniformly among the parallel transductors;

- **Switch** bypasses messages from the net's input to the Splitter until a segmentation mark is received. Then it stops receiving messages and sends the segmentation mark and a length of message sequence preceding it to the Merger.
- **Merger** forwards the messages from parallel transductors to the net's output until their number reaches the sequence length received from the Switch. Then the Merger sends a "releases" message to the Switch for it to continue receiving messages.

This protocol serves to keep the messages within their sequences: the net blocks on receiving a segmentation mark until the preceded sequence is fully processed.

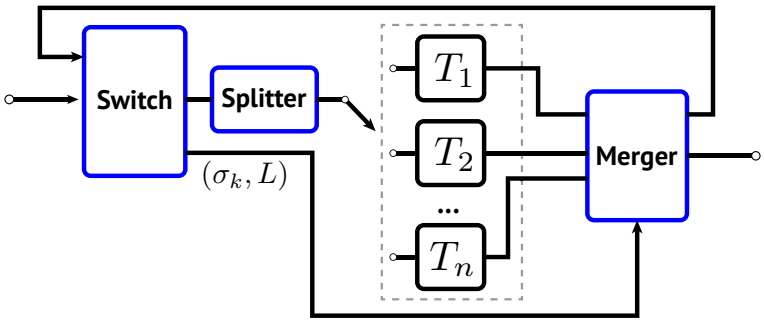


Figure 3.9: Parallel transductor.

Synch-Tables

A synch-table is constructed from an executable net with a single handler and a synchroniser generated by the compiler (Figure 3.10). The synchroniser receives an input message, combines values from it according to the declared pattern, and sends these combinations together with the input message to the handler. The handler, in turn, dynamically creates synchronisers corresponding to the received combinations, and sends the message to them. Once created, synchronisers stay in the net for further possible messages.

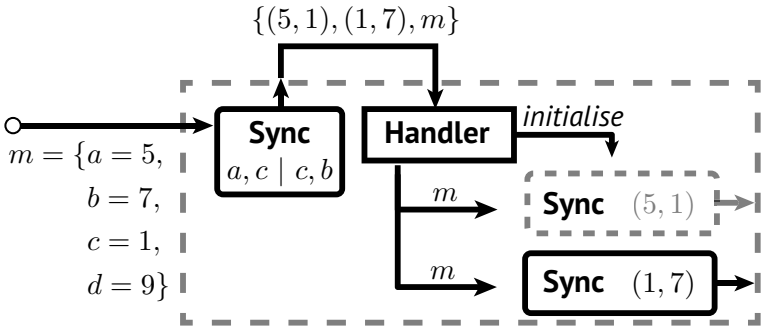


Figure 3.10: Synch-table.

3.3 Compiler

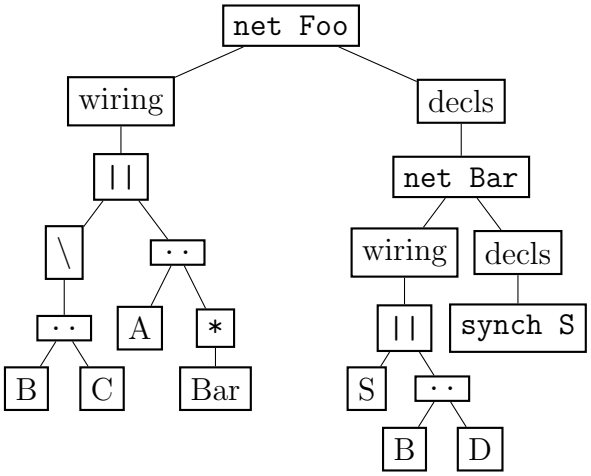
In this section we consider a transformation of textual network description into a network of runtime components.

3.3.1 Network Construction

An AstraKahn program consist of a network description, and declaration of synchronisers and boxes. The languages in which a network and synchronisers are written are presented in Section 2.3. The compiler takes the program and constructs a network of runtime components organised in a tree reflecting the hierarchy of the network (Figure 3.4b). Consider this process in details.

First, the compiler transforms the network description into an abstract syntax tree (AST):

```
net Foo (in | out)
  net Bar (in | out)
  synch S
  connect
    S || B .. D
  end
connect
  (B .. C)\ || A .. Bar*
end
```



(a) Network description. (b) AST.

Figure 3.11: An AstraKahn program transformed into an abstract syntax tree.

Then the following recursive procedure is performed:

- Visit root net. Construct scope of vertices that includes synchronisers and nested nets from declaration node, and all boxes;
- Perform post-order traversal of wiring expression:
  - for operand-nodes: construct corresponding runtime component (Section 3.2);
  - for operator-nodes: apply intended wiring pattern to operands (Section 3.3.2);

  Once a nested net is occurred, recursively apply procedure to it and, when its runtime component is build, proceed traversal.
- Create net component, include constructed runtime components in it.

With regard to the net hierarchy, the procedure visits the nets pre-orderly and in a 'lazy' manner, since nested nets are constructed only if they are used in wiring expressions.

3.3.2 Wiring

Once the compiler visits an operator-node of a wiring expression AST, it applies the wiring pattern to the operand(s). Since the connection between runtime components is expressed inside the components themselves (Section 3.1.2), from the compiler standpoint

an operand of the wiring operator is a set of runtime components from the corresponding sub-tree. For example, in the wiring expression of the net Bar (Figure 3.11b) operands of `||` are sets $\{S\}$ and $\{B, D\}$.

First, for each operand the compiler extracts input and output ports that are not connected to any components; we refer to the ports as *external*. Then, if there are identically named external input or output ports, they are plugged up with corresponding copiers or mergers respectively in order to insure one-to-one connections between ports.

Finally, the intended connection is applied. For *serial*, *parallel*, or *wrap-around* connections the ports are wired straightforwardly as described in Section 2.2.4. The case of the *serial replication* connection is more interesting since the connection is dynamic: it creates a "lazy" pipeline in which stages are constructed and wired on demand. In what follows a model implementation of the connection based on executable nets is discussed.

Consider a serial replication connection applied to a stage A with a single input and output channels. Assume that, regardless of actual implementation of forward and reversed fixed points, it is possible for the compiler to generate a net A' that wraps A and has the following properties:

- it has a single input port directly connected to A, and two output ports: "out", on which the net forwards messages that are FFP, and "next", on which the rest of the messages are sent;
- a runtime component of A' has a method `is_rfp()` that tests if the stage is in an RFP state.

In order to organise a dynamic pipeline of A', the compiler constructs an executable network (Figure 3.12) where the "out" ports of all stages are connected to the global output through a merger, and the "next" port of the last stage is connected to the net's handler. Once the handler receives a message, it creates a new stage of the pipeline, sends the message to it, and appends the new stage to the head of the pipeline. Furthermore, on each execution the handler removes RFP stages from the pipeline are rewire the remaining vertices.

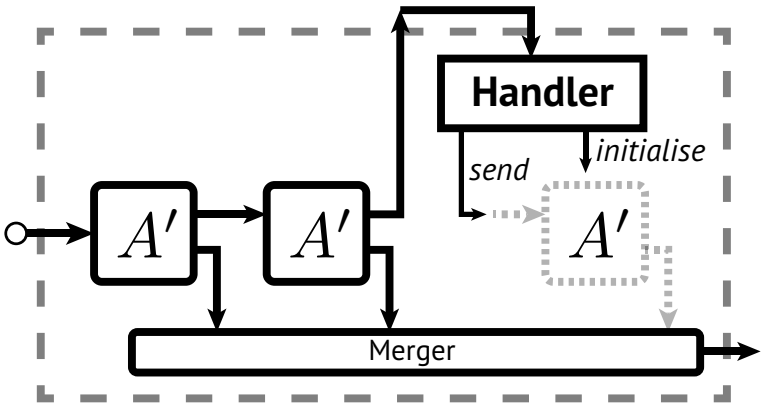


Figure 3.12: Serial replication connection.

Chapter 4

Self-Tuning Heuristics

4.1 Overview

In Section 2.4 we defined two regulating actions in AstraKahn that are managed by the coordinator:

- *Proliferation* replaces a primitive box by its parallel counterpart consisting of several copies of the original box. The number parallel copies is called the *proliferation factor*.
- *Fragmentation* breaks a single input message into a number of smaller ones by a morphism assigned to some transductor.

The coordinator applies these actions to an AstraKahn network at runtime in an attempt to improve its performance characteristics. The actions are coupled: fragmentation generates a large supply of messages, which can be processed in parallel by applying proliferation to boxes. This allows one to increases the throughput of individual boxes in the network and, possibly, decrease the overall processing time.

There are also two complementary actions: *deproliferation*, which decrease the proliferation factor for some boxes in favour of the ones with a higher computational demand, and *refragmentation*, which joins the generated messages into one and splits again with a different number of fragments.

However, under limited computational resources the positive effect of these actions has a certain upper bound. Hence, the coordinator must have sufficient intelligence to regulate the network in accordance with its intended behaviour and the computational demand. In order to do this the AstraKahn coordinator continually performs the optimisation loop:

- monitor runtime characteristics of each vertex;
- use built-in regulation strategies together with currently and previously observed properties to produce a new set of regulating actions;
- save regulating parameters for further use.

We refer to this form of runtime adaptivity as *self-tuning*. In the following sections we outline a number of possible regulating strategies.

4.2 Framework

Let an AstraKahn network implement some algorithm that admits divide-and-conquer strategy. The network receives a message, performs computations, and yields the result as a single message. Our objective is to minimise the overall running time, or, in other words, to minimise the *latency* of this network.

We focus on the networks that are primarily serially connected and consist mainly of transductors. This particular choice was made since a series of independent computational steps is a common pattern in algorithm design, and since AstraKahn transductors are designed to express primary computations in a program. Hereinafter we refer to the network as a *pipeline*, and to its transductors as *stages*.

At the beginning the coordinator is unaware of the network's runtime properties. In order to collect a substantial amount of profile information on the network, assume that either

- a program consists of a single pipeline and can be run several times;
- or the pipeline is serially replicated.

In either case the network's runtime profile is accumulated and specified in order to obtain more precise behavioural and performance model.

The proposed scheme is contingent on the availability of a large number of messages floating between pipeline stages. This could be a feature of the application defined in its natural form or the messages can be the result of applying a morphism, i.e. fragmentation.

For the fragmentation to be applied over the whole length of the pipeline, as opposed to a single transductor, each transductor is augmented with a morphism. Assume also that appropriate overrides are specified for each adjacent pair of morphisms. This requirement is essential in order to avoid global synchronisation caused by an explicit join-split in the middle of the pipeline.

Initially fragmentation is applied to the first stage of the pipeline, and an input message is split into a sequence of fragments that continue travelling along the pipeline. For a given point in time there is a number of "active" stages, i.e. the ones in which messages are being processed. We refer to this number as *effective pipeline depth*.

If there is no data dependency between the messages, the depth can potentially be equal to the whole length of the pipeline. However, it is often the case that a message depends on several other messages, since they are combined in overrides. Accordingly, the effective depth of the pipeline can be limited as well.

The main open issue with proliferation and fragmentation is one of predictive control. In a situation when distributed resources are limited, the decision to proliferate is based on an expectation of parallel speedup, future availability of computing cores, and communication constraints. Any predictive rule will necessarily have to take into account the resource footprint of each box including its dependencies on the content of messages circulating in the system. Additionally, since messages can be fragmented, a proliferation rule is dependent on the behaviour of the fragmentation mechanisms. It has to determine that proliferation is profitable despite the fragmentation overheads, as well as ascertaining the degree of the fragmentation.

4.3 Fragmentation

First, consider the initial message fragmentation. The number of fragments m has to be sufficient to use the available parallel resources. On the other hand, if m becomes too large, the processing time of an individual message becomes very short, while the communication overhead goes up as well. The communication overhead could eventually predominate over the parallel speedup. Additionally, the multitude of messages has the potential to overwhelm the buffering capacity of channels if the boxes receiving data from them prove too slow.

For a reasonable initial guess of m we can assume that the maximum effective depth is

equal to the full length of the pipeline, and that the processing rate of all the stages is the same; as a result, all the stages will be active the whole time. Under such circumstances, the fragmentation will produce $m = \sum_{i=1}^{n} c_i/2$ messages with the intention to load at least half of the channel's capacity at each stage. Later on, when the pipeline depth and other monitoring parameters are collected, m will be adjusted accordingly.

During the execution, if the communication overhead starts to be significant, or if the effective depth starts to gradually decrease due to a blocked channel, the number of messages will be reduced; and vice versa, if the parallel resources are not fully used during the execution, the initial fragmentation factor will be increased.

Here we assume that refragmentation is performed only between consecutive replicas of a serial replication, or not at all. It is simpler for the implementation since to apply refragmentation at an arbitrary point of the pipeline the runtime system has to make sure that the "front" of the message sequence is behind the vertex where the refragmentation is going to be made.

4.4 Proliferation

Proliferation replaces a single transductor by a number of copies. Such a replacement can potentially increase the performance provided that the copies operate in parallel. Parallel operation is not guaranteed: boxes are not assigned to the hardware cores, therefore if the number of boxes is greater than that of the available cores, the effect on the throughput depends on the quality of the scheduling.

However, assume that the effect from the parallel transductor is close to linear. This can be achieved for a relatively small network and sufficient number of cores. The resource limit is in terms of the maximum number of transductor copies that can be spawned. Assume also that each stage of the pipeline process messages with some throughput. Note that the running time of the pipeline is equal to its throughput multiplied by the number of messages. Hence, the throughput of the pipeline is equal to the minimum throughput of individual stages. Now the self-tuning of the pipeline boils down to the maximisation of the minimum throughput in the pipeline subject to the limited number of parallel boxes, which can be written formally as a max-min linear optimisation problem. The problem can easily be solved provided that the throughput of each stage is know [15]. However, the throughputs of the stages can only be determined at runtime, by observing the stages. In [16] a similar problem is solved for a pipeline in a steady state by monitoring the execution time of its stages and recalculating the proliferation factors after each new run.

The AstraKahn pipeline in question can not be assumed to be in a steady state since in the case of a small effective pipeline depth the stages work non- uniformly: only a small number of them are active at any given time. However, by choosing an appropriate time-scale the active stages can be considered to be in a steady state.

Let a pipeline of length L process all messages from a fragmented input sequence for time T. Assume the at the time t there are $d(t) \ll L$ stages are active for a period of time $T_s \leq T$. Within the period we can consider the sub-pipeline consisting of $d(t)$ stages to be in a steady state for T_s. Let T_p denote the time that it takes to process the observed properties of stages and reassign the proliferation factors. If $T_p \ll T_s$ (which can be tested at runtime as well), the runtime system can effectively solve the optimisation problem for the active pipeline stages achieving optimal parallel resource mapping from that time on.

The above approach is based on monitoring the properties of the stages and generating a short-term prediction that expectedly meets the constraints of the network. There is

another strategy in which the decision about proliferation is made individually by each of the stages based on local observations. The approach can be called *greedy* since each stage tries to optimise its local problem, in the hope that their collective choice will be optimal for the whole network.

For example, as before, assume that there is a global "pool" with M parallel transductors. Let a stage of the pipeline create a new parallel transductor on receiving a message provided that the pool is not empty. In other words, the stage always tries to process incoming messages in parallel subject to the available resources. In order to prevent monopolising parallel resources, the stage also "returns" parallel transductors back to the pool when the computational demand of the stage has decreased. For example, it can reduce its proliferation factor as soon as it is idle for an average inter-arrival time at the moment that the parallel transductor is taken from the pool.

Chapter 5

A Case for Morphisms: Particle-in-Cell

5.1 Overview

In order to demonstrate the adaptive facilities of AstraKahn, it is convenient to use a computational problem which, after straightforward domain decomposition, exhibits imbalances of the workload assigned to different processors. Such problems require problem-specific parallelisation patterns and load balancing techniques to achieve sufficient hardware utilisation.

The Particle-in-Cell (PIC) method [17], which is often used in plasma physics for modelling the motion of charged particles in electromagnetic fields, has the described property. The main challenge is that balancing computations on particles leads to a large communication overhead due to poor data locality, which may cancel out the benefits of load balancing. Consequently, modern parallelisation techniques for PIC inevitably suffer from a parametric trade-off between the computation imbalance and communication overhead. The choice of parameters that ensure optimal performance is usually the matter of heuristics or manual tuning.

5.2 The Problem

Consider a 1-dimensional plasma consisting of N_p electrons and N_p ions with no external fields applied. Since ions are much more massive than electrons, we can assume $m_i/m_e \to \infty$, neglect the motion of ions, and treat them as a uniform neutralising background. Our aim is to study the motion of the electrons in the electrostatic field.

The particles follow the Newton-Lorentz equations of motion:

$$\begin{aligned} m_e \frac{dv_i}{dt} &= \frac{q_e E(x_i)}{m_e} \\ \frac{dx_i}{dt} &= v_i \end{aligned} \tag{5.1}$$

where x_i and v_i are the coordinate and velocity of the ith particle respectively. From Maxwell's equations it follows that

$$E(x) = -\frac{\partial \varphi(x)}{\partial x} \tag{5.2}$$

$$\frac{\partial E(x)}{\partial x} = \frac{\rho(x)}{\varepsilon_0} \tag{5.3}$$

Combination of (5.2) and (5.3) results in Poisson's equation:

$$\frac{\partial^2 \varphi(x)}{\partial x^2} = -\frac{\rho(x)}{\varepsilon_0} \tag{5.4}$$

where ρ is the charge density. Hence, given a spatial charge distribution, one can compute electric field by solving (5.4) and (5.2), and then solve the equations of motion (5.1).

5.3 Particle-in-Cell

Particle-in-Cell (PIC) is a method for numerical solution of the equations from the previous section. Consider a 1D plasma within the domain $0 \leq x \leq L$ represented by a grid $\{x_j, j \in [0, N_g)\}$ of equidistant points:

$$x_j = j\Delta x, \quad \Delta x = L/N_g$$

N_p particles are described as a set of coordinate-velocity pairs $\{(r_i, v_i), i \in [0, N_p)\}$. Normally that $N_g \ll N_p$. The evolution of the system is computed in discrete time with a specified time-step Δt. In what follows we assume the charge and mass of electron, reference charge density, and electric constant to be equal to 1. This can be done by choosing an appropriate normalisation.

The following four stages are performed sequentially for each time-step:

- **Scatter.** Charge densities on grid points are computed from the particle positions by the following weighting:

$$\rho_j = \frac{N_p}{\Delta x} \sum_{i=1}^{N_p} W(r_i - x_j) \tag{5.5}$$

We assume $W(x)$ to be a linear function[1]

$$W(x) = \begin{cases} (1 - |x|)/\Delta x, & |x|/\Delta x < 1 \\ 0, & |x|/\Delta x \geq 1 \end{cases} \tag{5.6}$$

Therefore, a particle with the coordinate $r_i : x_j \leq r_i \leq x_{j+1}$ contributes to the charge density as follows:

$$\boxed{\begin{aligned} \rho_j &= \rho_j + \frac{1}{\Delta x}\left(\frac{x_{j+1} - r_i}{x_{j+1} - x_j}\right) \\ \rho_{j+1} &= \rho_{j+1} + \frac{1}{\Delta x}\left(\frac{r_i - x_j}{x_{j+1} - x_j}\right) \end{aligned}} \tag{5.7}$$

- **Field Solve.** The Poisson equation (5.4) can be written in finite-difference form:

$$\frac{\varphi_{j-1} - 2\varphi_j + \varphi_{j+1}}{\Delta x} = -\rho_j$$

[1] The choice of $W(x)$ correspond to the model called *cloud-in-cell* in which particles contribute to the charge density only at the nearest grid points. The contribution to the density value is proportional to the distance from the particle to the nearest grid point. This can be thought of as the density of finite uniformly charged cloud.

In order to solve it we use iterative Gauss-Seidel method:

$$\boxed{\varphi_j^n = \frac{1}{2}\left[\varphi_{j-1}^n + \varphi_{j+1}^{n-1} + \rho_j(\Delta x)^2\right]} \tag{5.8}$$

Then E is immediately obtained from (5.2):

$$\boxed{E_j = \frac{\varphi_{i-1} - \varphi_{i+1}}{\Delta x}} \tag{5.9}$$

- **Gather.** In order to compute electric field acting on particles from the values on grid points the same weighting as in **Scatter** is used:

$$E_i = \sum_{j=1}^{N_g} E_j W(x_j - r_i)$$

Using the weighting function W from (5.6) we obtain that only two neighbouring grid points (i.e. x_j and x_{j+1} such that $x_j \leq r_i \leq x_{j+1}$) contribute to the electric field acting on a particle. The resulting field is computed as follows:

$$\boxed{E_i = \left(\frac{x_{j+1} - r_i}{x_{j+1} - x_j}\right) E_j + \left(\frac{r_i - x_j}{x_{j+1} - x_j}\right) E_{j+1}} \tag{5.10}$$

- **Push.** Motion equations (5.1) can be numerically integrated with so-called *leap-frog* method:

$$\boxed{\begin{aligned} r_i^{t+\Delta t} &= r_i^t + v_i^{t+\Delta t/2}\Delta t \\ v_i^{t+3\Delta t/2} &= v_i^{t+\Delta t/2} + E\Delta t \end{aligned}} \tag{5.11}$$

In order to compute the values at $t = 0$, $v^{-\Delta t}$ must be computed first from electric field computed at the initial point in time.

The stages of a PIC iteration and dependencies between the grip and particles are outlined in Figure 5.1.

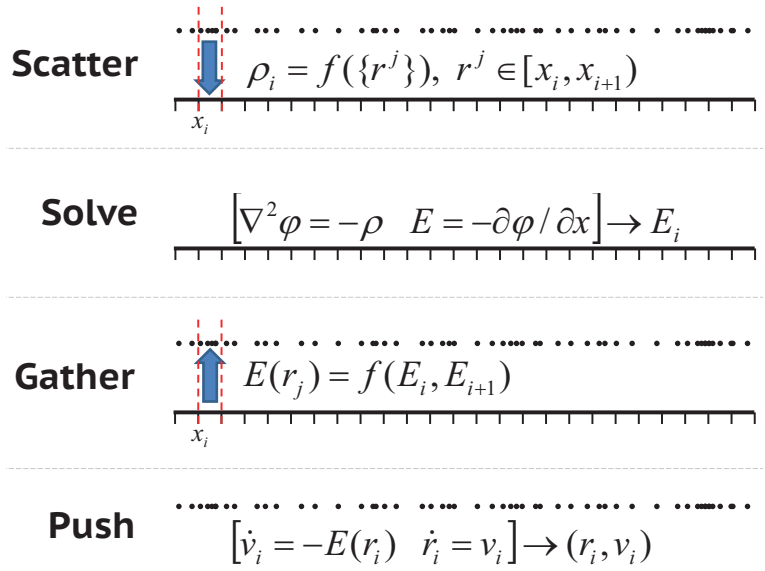


Figure 5.1: An iteration of the PIC method.

5.4 Parallelisation

5.4.1 Overview

The PIC uses two main data structures: a numerical grid and an array of particles' quantities. Since both of them are used on each time-step either severally or jointly, it is difficult to obtain a decomposition of the problem that allows balanced parallel computations with minimal overhead. An overview of existing parallel strategies for PIC is presented in [18]. In this section the main problems and trade-offs of PIC parallelisation are outlined.

First, in the Scatter and Gather the quantities from particles are used to compute values on corresponding grid points, and vice versa. In order to minimise communication between processors at each time-step, grid points should be placed locally with corresponding particles:

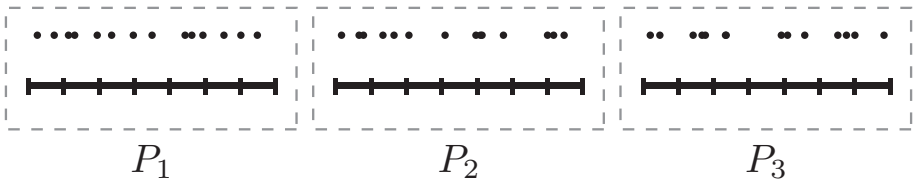


Figure 5.2: Grid-particle locality.

That said, communication is still required since boundary particles contribute to the adjacent grid points from different processors. Furthermore, since those particles are moving, some effort is required to preserve the grid-particle locality. For example, given a static grid partition, particles that are pushed beyond a grid boundaries must be immediately moved to adjacent processors.

Apart from the communication overhead, there is a load-balancing issue. Although Push is the most expensive stage of an iteration, the decomposition based purely on particles would violate the desirable grid-particle locality. On the other hand, an irregular grid partition with the grid-particle locality preserved and a nearly even particle distribution will unlikely result in balanced computations because of the motion of particles. In this case an effort has to be made either toward prediction of the plasma evolution or dynamic reconfiguration of the partition.

As a case study we chose the parallelisation strategy developed in [19], which is based on a static grid partition with the grid-particle locality preserved and runtime load-balancing. The following sections describe this strategy and discuss its implementation in AstraKahn with regard to the runtime adaptive mechanisms provided by the language.

5.4.2 Decomposition

Consider the 1D PIC problem defined in . Split the problem into k subproblems as follows:

- the grid is divided into k non-overlapping regions with nearly equal numbers of consecutive grid points;
- each region is augmented with the corresponding particles: for a region $G = (x_s, ..., x_e)$ the particles $P = \{(r_i, v_i) \mid x_s \leq r_i < x_e + \delta x\}$ are selected, i.e. particles located between two adjacent regions go to the left one:

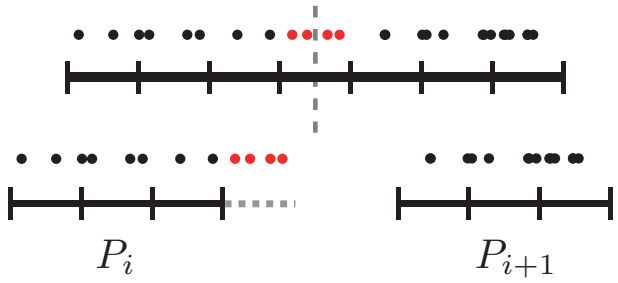


Figure 5.3: Particles partition.

5.4.3 Communication

Although the subproblems can be processed in parallel at each of the PIC stages, they are not fully independent. This requires the following communication steps to be carried out:

- **Scatter and Gather:** The subproblems possess the grid-particle locality, i.e. they only need to communicate the boundary grid points and particles:

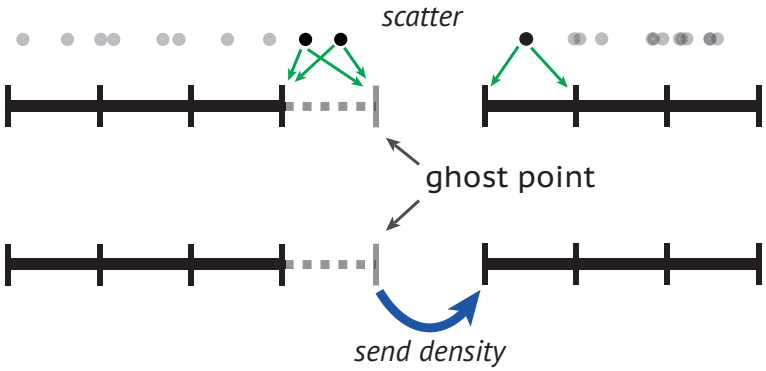


Figure 5.4: Communication after Scatter.

- **Push:** Communication is needed to preserve the grid-particle locality: the particles pushed outside a region are sent into its adjacent regions:

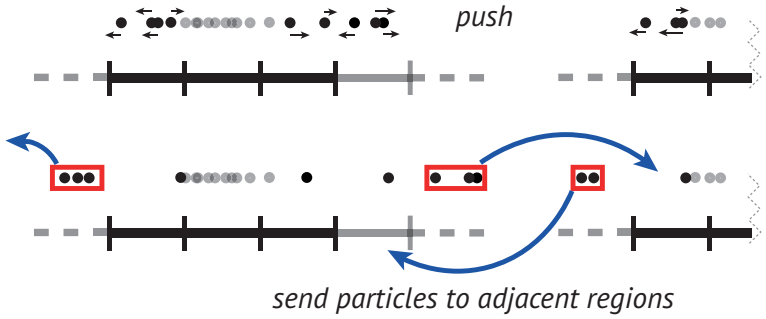


Figure 5.5: Communication after Push.

- **Field Solve:** The three point stencil used in the Gauss-Seidel method (see (5.8)) requires the regions to maintain ghost-points and send their values to the corresponding adjacent regions:

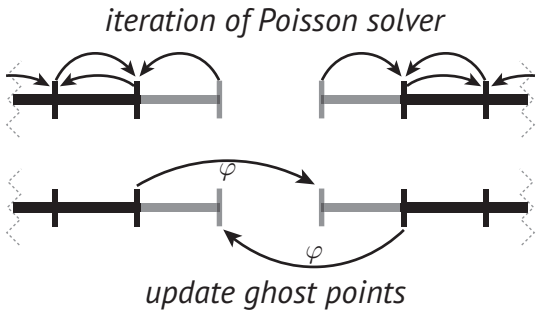


Figure 5.6: Communication after Field Solve.

5.4.4 Load-Balancing

Since the grid partition is regular, the computation at the Field Solve stage is always balanced. By contrast, the computational balance of the other three stages depends on the distribution of particles among the grid regions. The distribution is governed by particle motion and can not be guaranteed to be even.

In order to balance the computations there is a mechanism for particle migration: particles from "overpopulated" processors can by dynamically delegated to "underpopulated" ones via *windows*. A window is a contiguous block of grid points that is assigned to the other processor (Figure 5.7). The windows do not affect the Field Solve that is performed on the original regions, whereas Gather, Push, and Scatter stages are performed on the windows by the assigned processor.

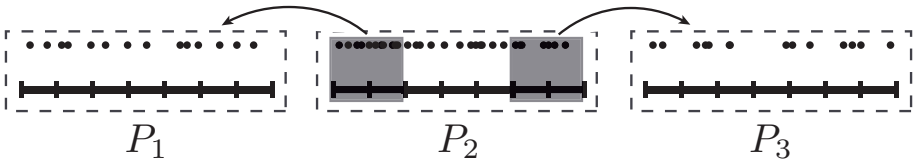


Figure 5.7: Particle windows assigned to other processors.

Given an appropriate delegation strategy, the mechanism effectively balances the particles among processors. However, it introduces a new trade-off between the migration overhead and particle imbalance. The mechanism provides a user-defined parameter representing a threshold of particle imbalance that triggers the balancing procedure.

5.5 Implementation in AstraKahn

5.5.1 Sequential PIC

A sequential PIC problem can naturally be represented in AstraKahn as a serial replication of the four described stages connected in a pipeline:

```
(scatter .. poisson* .. ef_solve .. gather .. push)*
```

where the Field Solve stage is split into two separate vertices standing for the Poisson and electric field solvers.

Provided that an input message contains a grid G, particle array P, and necessary scalar quantities, each vertex in the pipeline can be implemented as a transductor that performs computations on the grid or particles and which sends the resulting message.

Assume that the reverse and forward fixed points in the serial replications are such that the `poisson` vertex is replicated until the error becomes sufficiently small, and the whole

pipeline is replicated a specified number of times corresponding to the desired simulation time.

5.5.2 Parallel PIC

Fragmentation

In order to enable parallelism in the network we need to augment the vertices with corresponding morphisms that split the initial message into a number of smaller ones and which join them back to obtain the result. Furthermore, the transductors representing PIC stages must be programmed to be able to process "fragments" of an initial problem.

The morphism's inductor is the same for all stages and follows the decomposition strategy described in Section 5.4.2. In addition, since Gather, Scatter, and Field Solve use the adjacent grid points from the neighbouring regions, two *ghost points* are attached to each message[2] (Algorithm 4).

function `split`(M, k)
$G_i|_{i=1\ldots k} = M.G.\texttt{split()}$
$P_i|_{i=1\ldots k} = \{(x, v) \in M.P | \min_x G_i \leq x < (\max_x G_i + M.\delta x)\}$
$Ghost_i^L\big|_{i=1\ldots k} = G_{i-1 \pmod k}[-1]$
$Ghost_i^R\big|_{i=1\ldots k} = G_{i+1 \pmod k}[0]$
$R_i|_{i=1\ldots k} = M.\texttt{copy()}$
$R_i|_{i=1\ldots k} = R_i.\texttt{update}(G_i, P_i, Ghost_i^L, Ghost_i^R)$
`return`$([R_1 \ldots R_k])$

Algorithm 4: Morphism inductor.

By contrast, the morphism's reductor is stage-specific because it has to perform communication between subproblems. However, instead of writing five separate reductors, we compose each of them out of three boxes two of which are common for all stages:

- The first box is a reductor that simply combines the regions, corresponding particle arrays, and ghost points without performing any operations on them (Algorithm 5).
- The next box is the transductor that performs stage specific communications as described in Section 5.4.3. For example, for the Scatter stage it adds the partial density from the right ghost cells to their "actual" counterparts in the neighbouring regions. (Algorithm 6).
- Finally, the last box is also a transductor that joins the subproblems into a single grid and particle array (Algorithm 7).

Putting it all together, the five morphisms are constructed as follows:

[2]Hereinafter we omit the message data that is not involved in the algorithm in question (e.g. Δx, Δh, the number of grid points, etc.). We assume the data to be inherited from the input messages as appropriate.

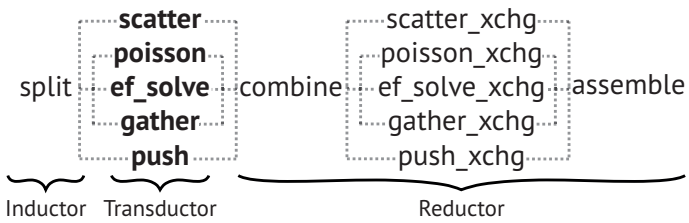


Figure 5.8: Parallel PIC morphisms.

function `combine`$(M_i,\ i = 1 \dots k)$
$R =$ `msg`$\left(\left(M.G_i,\ M.P_i,\ Ghost_i^L,\ Ghost_i^R\right)\big|_{i=1\dots k}\right)$
`return`(R)

Algorithm 5: Morphism reductor, part 1: combiner

function `scatter_xchg`(M)
$M.G_{i+1 \pmod k}[0]\big|_{i=1\dots k}$ += $M.Ghost_i^R$
`return`(M)

Algorithm 6: Morphism reductor, part 2: communication

function `assemble`(M)
$G =$ `join`$(M.G_i|_{i=1\dots k})$
$P =$ `join`$(M.P_i|_{i=1\dots k})$
`return(msg`$(G, P))$

Algorithm 7: Morphism reductor, part 3: assembler

Overrides

In order to avoid global synchronisation after each stage, appropriate overrides have to be specified for each connected pair of stages. Since each subproblem communicates only with its immediate neighbours, the data can be efficiently transferred by synch-tables. As an example, consider an override between `scatter` and `poisson`, a similar approach applies also to the rest of the morphism pairs.

As follows from Algorithm 4, the right ghost point of a region has to be sent to its right neighbour. This can be done by extending the scatter's transductor with a separate output channel, on which the right ghost point is sent. Then the messages go to the synch-table that routes them in such a way that the ghost point and the corresponding neighbouring region are sent to the same synchroniser. The synchroniser in turn combines the messages and sends the result to the next stage (Figure 5.9).

However, for the communication to be completed the density value from the attached ghost point has to be added to the leftmost point of the grid. This can be done by adding an auxiliary transductor before the one in the `poisson` morphism.

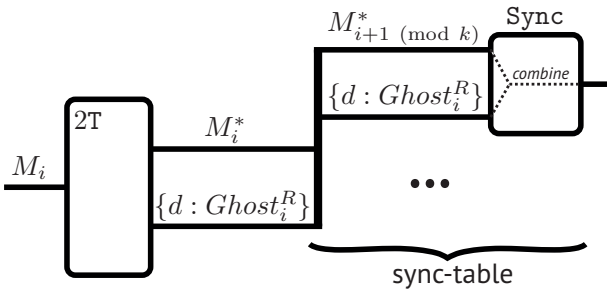


Figure 5.9: Override between `scatter` and `poisson` stages.

Load balancing

Consider the PIC load balancing method discussed in Section 5.4.4. In AstraKahn terms the window assignment corresponds to copying grid- and particle-data within the selected window from one message to another. Since the decision about window creation is made at runtime and collectively for all subproblems, it is convenient to embed load balancing into the fragmentation mechanism.

Let each morphism's inductor compute the particle distribution among the subproblems. If the particle imbalance exceeds a (sufficiently low) predefined threshold, a global set of windows is computed, and each generating message is augmented with sets of "incoming" (assigned *from* other messages) and "outcoming" (assigned *to* other messages) windows:

$$W_i^{in} = \{(G_j^i, P_j^i) \mid j \in S_{in} \subseteq \{1..k\} \backslash i\}$$
$$W_i^{out} = \{(G_i^j, P_i^j) \mid j \in S_{out} \subseteq \{1..k\} \backslash i\}$$

where (G_i^j, P_i^j) is a window of the jth message assigned to the ith one.

Transductors for `gather`, `push`, and `scatter` have to be programmed to work within the grid region $\big(G_i \backslash \{G_i^j \mid j \in S_{out}\}\big) \cup \{G_j^i \mid j \in S_{in}\}$. This creates additional communication points: as a normal region, a window contains particles that depend on the leftmost point of the adjacent region (see Figure 5.4). Furthermore, since the electric field is computed regardless of the windows, the `scatter`–`poisson` and `ef_solve`–`gather` overrides have exchange the charge density and electric field between messages and their "outcoming" windows.

Now the runtime system can perform the load balancing procedure by "refragmenting" the messages. It does so once an imbalance is detected in the execution times of parallel boxes at Gather, Push, or Scatter stages. The threshold of the imbalance is controlled by self-tuning mechanism.

Chapter 6

Conclusion and Future Work

6.1 Summary

This thesis presented the up-to-date definition of AstraKahn and the architecture of its prototype, which includes the language compiler and runtime system. On the runtime system level the core vertices are described as objects that implement unified interfaces, while the built-in extensions are composed of the core vertices hierarchically. This allows one to adjust vertex behaviour and implement new extensions without going deeply into the code; this is particularly useful at the current stage since the language is not stable yet.

The described prototype is being implemented in Python with a multicore environment in mind; at the moment the core and most extensions of AstraKahn are completed [8].

6.2 Future Work

Once the prototype is implemented, our primary goal is to conduct various case studies and to examine self-tuning heuristics in order to find a good adaptation strategy suitable for whole classes of AstraKahn applications.

Furthermore, we are interested in practical applications involving pipeline and data parallelism, as well as irregular computations, since the optimal level of concurrency in such cases is particularly difficult to ascertain. The PIC method in the present thesis is an example of such application.

Further work will include supporting the Message Definition Language and components' reconciliation [9], as well as studying the synchroniser model and developing its static analysis; a preliminary work on this topic is done in [20].

Appendix A

Execution Interface of Synchronisers

```
method is_ready()
    inputs_ready = ready inputs that cause transitions from the current state
    if inputs_ready is empty then
        return (False, False)
    end

    /* Remove channels that cause potential send to blocked channels */
    foreach channel in inputs_ready do
        transitions = transitions from the current state caused by channel
        outputs_to = outputs used in send statements of transitions

        if outputs_to has blocked channels then
            remove channel from inputs_ready
        end
    end

    if inputs_ready is empty then
        return (True, False)
    else
        store inputs_ready
        return (True, True)
    end
```

Algorithm 8: Synchroniser: readiness test.

```
method fetch(inputs_ready)
    channel = take(least frequently taken channel from inputs_ready)
    message = get(channel)

    return channel, message
```

Algorithm 9: Synchroniser: message fetch.

```
method run(channel, message)
    choose_transition()
    if no transition has been taken then
        /* Drop the message. */
        return True
    end

    /* Perform operations augmented to the taken transition. */
    assign()
    send()
    new_states = goto()
    return choose_state(new_states)
```

Algorithm 10: Synchroniser: execution.

```
procedure choose_transition()
    foreach priority of transitions in the current state do
        /* Iterate over transition priorities in the current state from highest to lowest. */
        allTransitions = transitions caused by channel | priority

        valid_transitions = allTransitions | ordinary ones, with satisfied conditions
        elseTransition = allTransitions | .else, with satisfied conditions

        if (valid_transitions is empty) and (elseTransition is None) then
            continue
        else if (valid_transitions is empty) and (elseTransition is not None) then
            take(elseTransition)
        else if valid_transitions is not empty then
            take(least frequently taken transition from valid_transitions)
    end
```

Algorithm 11: Synchroniser: choose transition according to priorities and conditions.

```
procedure choose_state(new_states)
    immediate_states = new_states
    foreach state in immediate_states do
        inputs_ready = ready inputs that cause transitions from state
        outputs_to = outputs used in send stmts in state

        if (inputs_from is empty) or (outputs_to has blocked channels) then
            remove state from immediate_states
        end
    end

    if immediate_states is not empty then
        return least frequently taken state from immediate_states
    else
        return least frequently taken state from new_states
    end
```

Algorithm 12: Synchroniser: choose preferable state to move.

Bibliography

[1] Farhad Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, 1122:1–18, 1998.

[2] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[3] Michael G. Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. The concurrent collections programming model. Technical Report TR 10-12, Rice University, 2010.

[4] Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *The 14th Workshop on Compilers for Parallel Computing*, 2009.

[5] F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S.B. Scholz, and A. Shafarenko. *S-Net Language Report 2.0*. Number 499 in Technical Report. University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[6] B. Gijsbers and C. Grelck. An efficient scalable runtime system for macro data flow processing using S-Net. *International Journal of Parallel Programming*, 42(6):988–1011, 2014.

[7] Alex Shafarenko. AstraKahn: A coordination language for streaming networks. 2014.

[8] Max Kuznetsov. AstraKahn compiler and runtime system. https://bitbucket.org/mkuznets/astrakahn-runtime, June 2015. (accessed June 3, 2015).

[9] Pavel Zaichenkov, Olga Tveretina, and Alex Shafarenko. Interface reconciliation in Kahn process networks using CSP and SAT. *CoRR*, abs/1503.00622, 2015.

[10] C. Grelck and F. Penczek. Implementation architecture and multithreaded runtime system of S-Net. In S.B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 60–79. Springer-Verlag, 2011.

[11] D. Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technical University of Vienna, Vienna, Austria, 2011.

[12] Merijn Verstraaten, Stefan Kok, Raphael Poss, and Clemens Grelck. Task migration for S-Net/LPEL. In C. Grelck, K. Hammond, and S.B. Scholz, editors, *2nd HiPEAC*

Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany. HiPEAC, 2013.

[13] Vu Thien Nga Nguyen and Raimund Kirner. Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms. In *ICA3PP (1)*, pages 357–369, 2013.

[14] Thomas M Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California. Berkeley, California, 1995.

[15] Seymour Kaplan. Application of programs with maximin objective functions to problems of optimal resource allocation. *Operations Research*, 22(4):pp. 802–807, 1974.

[16] Jaroslav Sykora and Sven-Bodo Scholz. Towards self-adaptive concurrent software guided by on-line performance modelling. In C. Grelck, K. Hammond, and S.B. Scholz, editors, *2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany*. HiPEAC, 2013.

[17] C. K. Birdsall and Langdon. *Plasma Physics via Computer Simulation (Series on Plasma Physics)*. Taylor and Francis, 1991.

[18] Edward A. Carmona and Leon J. Chandler. On parallel PIC versatility and the structure of parallel PIC approaches. *Concurrency: Practice and Experience*, 9(12):1377–1405, 1997.

[19] Steven J. Plimpton, David B. Seidel, Michael F. Pasik, Rebecca S. Coats, and Gary R. Montry. A load-balancing algorithm for a parallel electromagnetic particle-in-cell code. *Computer Physics Communications*, 152(3):227 – 241, 2003.

[20] Anna Tikhonova. A synchronisation facility for a stream processing coordination language. Master's thesis, University of Hertfordshire. Hatfield, UK, 2015.