Correctness of automated distribution of sequential programs

Cyrille Bareau, Benoit Caillaud, Claude Jard, René Thoraval

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1724

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

CORRECTNESS OF AUTOMATED DISTRIBUTION OF SEQUENTIAL PROGRAMS

Cyrille BAREAU
Benoît CAILLAUD
Claude JARD
René THORAVAL

Juillet 1992

RR-1724

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Correctness of Automated Distribution of Sequential Programs

Cyrille Bareau Benoît Caillaud Claude Jard René Thoraval*

IRISA Campus de Beaulieu
F-35042 RENNES Cedex FRANCE
<*name*>@irisa.fr

Publication Interne n°665 - Juin 1992 - 32 pages - Programme 1

Abstract

In this paper, we prove that the data-driven parallelization technique, which compiles sequential programs into parallel programs for distributed memory parallel computers, is correct. We define a model based on labeled transition systems, and we prove, from the chosen compilation rules, the confluence of all possible behaviours of the parallel programs we obtain, in spite of asynchronism due to the communications.

We also show that this model is powerful enough to prove the correctness of various optimizations of the basic compilation mechanism.

Preuve de la distribution de programmes séquentiels

Résumé

Le but de cet article est de prouver la correction de la technique de parallélisation dirigée par les données, qui permet de compiler des programmes séquentiels en programmes parallèles s'exécutant sur machines à mémoire distribuée. On définit un modèle, fondé sur les systèmes de transition étiquetés, qui permet de montrer, à partir des règles de compilation choisies et malgré l'asynchronisme dû aux envois de messages, que tous les comportements possibles des programmes parallèles obtenus sont confluents.

Nous montrons que ce modèle est suffisamment général pour prouver diverses optimisations apportées au mécanisme élémentaire de compilation.

*Université de Nantes, Section Informatique, 2 rue de la Houssinière F-44072 Nantes cedex 03

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (URA 227) UNIVERSITE DE RENNES I INSA DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

1 Introduction

1.1 Motivation

Large scale parallelism promises high performance computing using the power of distributed memory parallel computers. Unfortunately, it is now clear that difficulties in programming them efficiently limit their use.

Current programming techniques are often based on communicating sequential processes. This approach makes it difficult to program large parallel systems in a practical and efficient way. The definition of processes is application dependent and requires a strong participation by the user. To resolve this problem, one radical answer is to hide parallelism from the programmer.

Currently, the main parallelizing technique in this area is the "data-driven parallelization" approach, which consists of creating communicating processes automatically from sequential code plus data decomposition specifications [5, 12, 1, 10]. To be used as an automatic parallelization method, this promising technique needs to be extended and mastered: it has to be given a solid theoretical foundation.

1.2 Approach

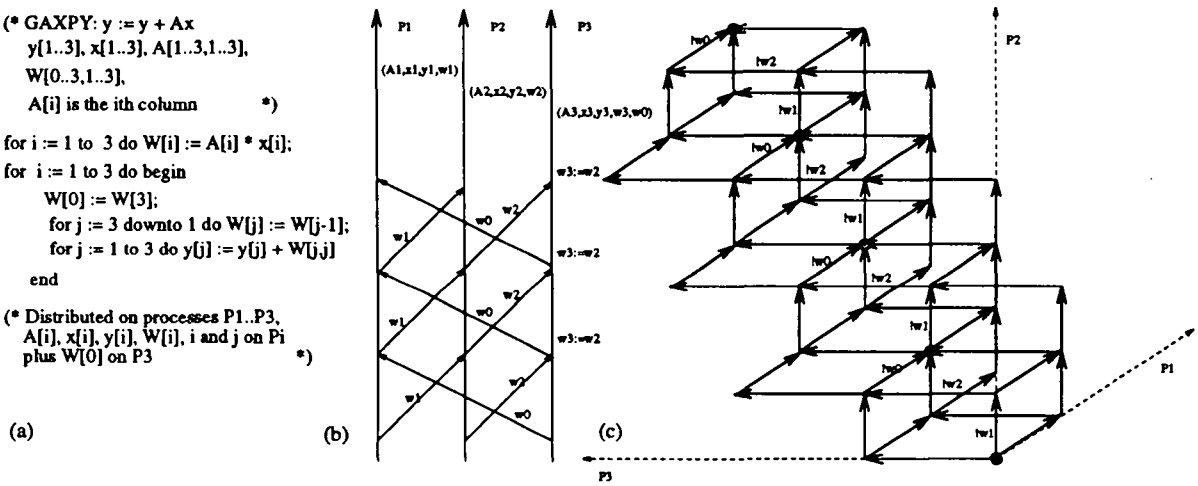


Figure 1: (a) source program, (b) its execution, (c) the resulting transition system

We will introduce informally this parallelization technique using the example of figure 1. We deal with a sequential source program (a), annotated with a data distribution which assigns program variables to processes (here on the three processes P1 to P3). The principle is that the owner of a variable is responsible for its assignment (local writes), and thus the distant variables belonging to the right part of the assignment must be exchanged. Figure (b) shows the message exchanges obtained by running the parallel generated code. Figure (c) shows the corresponding transition system which will serve as a model: we observe the classical explosion of the possible behaviours due to asynchrony and parallelism, but with a regular structure due to confluence. Some particular snapshots are circled (when communication channels are empty): these states will be connected with the source program states.

The data-driven asynchronous parallelization technique has been widely accepted, though it has not yet been formally proven. Whereas its correctness seems obvious, a proof is more difficult. Non-determinism due to asynchronous communications gives rise to an involved problem which is quite hard to solve directly. In order to overcome this obstacle (and for aesthetic reasons as well), it is suitable to split the problem into well-jointed parts. Moreover, as we intend to extract a reusable framework from the correctness proof of a particular parallelization technique, such a decomposition turns out to be central. The proof to be given has to capture the "essence" of the correctness of the technique under consideration.

A natural and fruitful decomposition is suggested by the following informal argument, which helps understanding the correctness of the technique. It is important to observe that this argument is based on the last claim, also the most questionable one :

- applied to a fragment of sequential code which may be executed in one step, the technique seems correct ;
- therefore, applied to an entire sequential program, it also seems correct, provided a "good angel" ensures a sufficiently synchronized behaviour of the processes (the parallel simulation of one step has to be completed before embarking on the simulation of another step) ;
- finally, although the real non-determinism forbids such a guaranty, it seems limited enough not to cause unacceptable distortions : the presence of the "good angel" is in fact unnecessary.

In this paper, we formalize the whole argument and prove that it does hold. First, we deal with the last and most problematic claim in a way that is independent of the parallelization technique under consideration. We define a model, close to the class of confluent systems described in [8] (chapter "Determinacy and Confluence"). In spite of limited non-determinism, our model allows one to import classical techniques for proving correctness of compilation functions (see for instance [9]) within a deterministic world. Thus, this model justifies the formalization of the first two claims, which are clearly related to such proof techniques. As desired, it provides the basis of a formal framework for designing other asynchronous parallelization or optimization techniques, and for proving their correctness.

1.3 Paper organization

The paper is organized as follows:

We first present the theoretical model, based on products of labeled and partially deterministic transition systems [3]. This allows us to compare the behaviours of the sequential source program with the corresponding parallel ones. One key semantic property of the resulting transition systems is the "diamond" property that, thanks to Newman's theorem, reduces the combinatorics of all the parallel behaviours to a single significant behaviour easily connected to the source program. We prove also that some properties of the communication medium are required, and that they are provided, for example, by the usual point-to-point Fifo channels.

Then we propose a simplified sequential language, given by a syntax and an operational semantics. We define the notations for the distribution of data, basis of the program

generation in a SPMD model. We formally describe the compilation rules, relying on the standard notion of "refresh" of remote variables. Finally we prove the correctness of these rules (based on the same approach as in [4]).

The last part of the paper shows that our formal model and proof methodology can be used to design other transformations; we use as examples an optimization of the refresh mechanism, the formal treatment of arrays in the language and an anticipation of message send.

2 Models and theoretical basis

2.1 Transition systems

A transition system P is denoted by $\langle q_P, Q_P, A_P, \rightarrow_P \rangle$, where q_P is the initial state, Q_P is the set of states, A_P is the action alphabet and $\rightarrow_P \subseteq Q_P \times A_P \times Q_P$ is the transition relation. A transition $(q, \alpha, q') \in \rightarrow_P$ is denoted by $q \xrightarrow{\alpha}_P q'$. The usual product on transition systems is denoted by $\prod$ or $\times$.

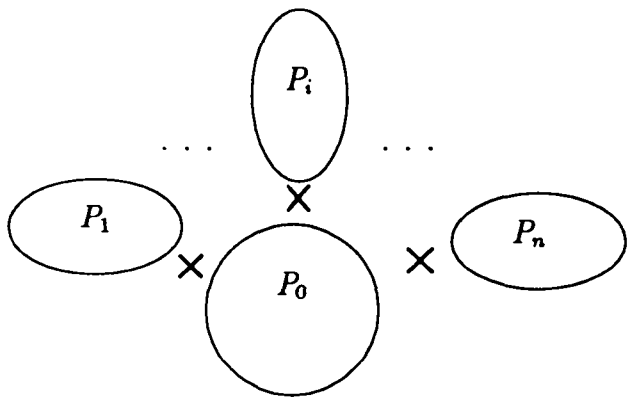


Figure 2: Architecture of a parallel program

A distributed program over processors $I = 1 \ldots n$ is hereafter modeled by a vector $(P_i)_{i=0\ldots n}$ of transition systems (figure 2), where P_0 models the communication medium, and $\forall i \in I$, P_i models the i^{th} process of the program. Communications between processes take place through P_0 which captures the communication asynchronism.

The set E denotes the collection of all data values that can bet sent or received. The alphabet of P_0 (A_{P_0}) is the set of communication actions and is the disjoint union ($\oplus$) of send (!) and receive (?) actions of a message $e \in E$ from $i \in I$ to $j \in I$:

$$
\begin{array}{lrcl}
 & A_{P_0} & = & \Sigma \\
 & \Sigma & = & \oplus_{i \in I} \Sigma_i \\
\forall i \in I & \Sigma_i & = & \Sigma_i^! \oplus \Sigma_i^? \\
\forall i \in I, \forall \gamma \in \{!, ?\} & \Sigma_i^\gamma & = & \oplus_{j \in I} \Sigma_{i,j}^\gamma \\
\forall i, j \in I & \Sigma_{i,j}^! & = & \{!e_{i \rightarrow j}\}_{e \in E} \\
\forall i, j \in I & \Sigma_{i,j}^? & = & \{?e_{j \rightarrow i}\}_{e \in E}
\end{array}
$$

Any process that performs a receive is non-deterministic since it accepts several data values — i.e. several receive events. However this non-determinism is particular since it can be discarded by an abstraction of data values. The equivalence relation ρ (definition 1) and the quotient of a transition system by an equivalence relation over actions (definition 2) formalize this data abstraction.

Definition 1 *Let ρ be the equivalence relation defined by:*

$$\rho = \{(?e_{i\to j}, ?f_{i\to j})\}_{i,j\in I\, e,f\in E}$$

The equivalence relation ρ enables an abstraction of data values on receive actions.

Definition 2 (Quotient Transition System) *Let P be a transition system and ξ an equivalence relation over A_P. The quotient transition system P/ξ is defined by:*

$$P/\xi = \langle q_P, Q_P, A_P/\xi, (1_{Q_P} \times \varphi_\xi \times 1_{Q_P})(\to_P)\rangle$$

where $\varphi_\xi : A_P \longrightarrow A_P/\xi$ is the mapping which associates to a letter $\alpha \in A_P$ the class of α modulo ξ, and $1_{Q_P} \times \varphi_\xi \times 1_{Q_P} : (q,\alpha,q') \longmapsto (q,\varphi_\xi(\alpha),q')$.

We now define two predicates:

Definition 3 *Let P be a transition system, and $B \subseteq A_P$.*

$$\begin{cases} det^{ext}(P,B) & \Longleftrightarrow & \forall q \in Q_P \ |\{\alpha \mid \exists q' \in Q_P,\ (q,\alpha,q') \in \to_P\} \cap B| \leq 1 \\ det^{int}(P) & \Longleftrightarrow & \forall \alpha \in A_P,\ \forall q \in Q_P,\ |\{q' \mid (q,\alpha,q') \in \to_P\}| \leq 1 \\ & & (i.e.\ \to_P \text{ is a function} : Q_P \times A_P \longrightarrow Q_P) \end{cases}$$

The predicate $\det^{ext}(P,B)$ means that the transition system is externally deterministic over B: for each state of P, at most one action of B can be performed. And $\det^{int}(P)$ means that the transition system is internally deterministic, which corresponds to the usual meaning of determinism.

In the sequel, distributed programs under hypotheses 1 and 2 are considered.

Hypothesis 1 *The distributed program matches the following predicates:*

$$\begin{cases} \forall i \in I & det^{ext}(P_i/\rho, A_{P_i}/\rho) \\ \forall i,j \in I & det^{ext}\left(P_0, \Sigma^?_{i,j}\right) \\ \forall i \in I \cup \{0\} & det^{int}(P_i) \end{cases}$$

Hypothesis 1 ensures that nondeterminism in P_i arises only when P_i performs a receive action from a given process, but with an unknown value. Since P_0 is deterministic over the receive actions of messages from j to i, the product $P_i \times P_0$ is deterministic over the actions of P_i.

Definition 4 (Diamond Property) *Let P be a transition system and $B \subseteq A_P^2$.*

- *P has the diamond property over B iff* $\forall q \in Q_P, \forall(\alpha,\beta) \in B, \forall q_1, q_2 \in Q_P,\ q_1 \neq q_2$:

$$\left\{ \begin{array}{l} q \xrightarrow{\alpha}_P q_1 \\ q \xrightarrow{\beta}_P q_2 \end{array} \right. \Longrightarrow \exists q' \in Q_P \left\{ \begin{array}{l} q_1 \xrightarrow{\beta}_P q' \\ q_2 \xrightarrow{\alpha}_P q' \end{array} \right.$$

In such a case, it is convenient to say that the following diagram holds:

$$\begin{array}{ccc} q & \xrightarrow{\alpha}_P & q_1 \\ \downarrow_P \beta & & \downarrow_P \beta \\ q_2 & \xrightarrow{\alpha}_P & q' \end{array}$$

- *P has the diamond property iff it has the diamond over A_P^2.*

Hypothesis 2 *P_0 has the diamond property over $\Sigma^2 \setminus \bigcup_{i,j \in I} \left(\Sigma^!_{i,j}\right)^2$*

Theorem 1 *Let $(P_i)_{i=0\ldots n}$ be a distributed program verifying hypotheses 1 and 2. The product transition system $P = \prod_{i=0}^{n} P_i$ has the diamond property and is internally deterministic, i.e. $det^{int}(P)$.*

The sketch of the proof of theorem 1 is as follows: every pair of distinct actions that can occur from any state of the distributed program should be considered. Furthermore for each pair it should be shown that the diamond property is verified : hypothesis 1 ensures the diamond property for any pair of actions that are not both communications, and hypothesis 2 ensures the diamond property for any pair of communication actions.

The reflexive and transitive closure of $\rightarrow_P$ is denoted $\rightarrow_P^\star$.

Definition 5 (Confluence) *A transition system P is said to be confluent when the transition system $(q_P, Q_P, A_P^\star, \rightarrow_P^\star)$ has the diamond property — i.e. the following diagram holds:*

$$\begin{array}{ccc} q & \longrightarrow_P^\star & q_1 \\ {}_P\downarrow_\star & & {}_P\downarrow_\star \\ q_2 & \longrightarrow_P^\star & q' \end{array}$$

Theorem 2 (Newman, 1942) *A transition system which has the diamond property is confluent.*

Theorem 3 *An internally deterministic confluent transition system has the following properties:*

1. *There is at most one maximal state (sink state).*
2. *The transition system has a maximal state if and only if it has no infinite computation.*

A proof of both theorems is detailed in [2].

2.2 Implementation of P_0

We hereafter use unbounded point-to-point FIFO communication channels. We give in this section a model of FIFO channels in terms of a transition system and show that it verifies hypotheses 1 and 2.

Definition 6 *Let* $P_0^F = \left\langle (\emptyset, \ldots, \emptyset), (E^\star)^{n^2}, \Sigma, \rightarrow_F \right\rangle$ *be a transition system, where* $\rightarrow_F$ *is defined by the following axiomatic:*

$\dfrac{q_{i,j} \xrightarrow{\alpha} q'_{i,j}}{(\ldots, q_{i,j}, \ldots) \xrightarrow{\alpha} (\ldots, q'_{i,j}, \ldots)}$	$e \cdot q_{i,j} \xrightarrow{?e_{i\rightarrow j}} q_{i,j}$	$q_{i,j} \xrightarrow{!e_{i\rightarrow j}} q_{i,j} \cdot e$

The transition system P_0^F is a model of a complete network of point-to-point FIFO channels over I.

Theorem 4 *P_0^F verifies hypotheses 1 and 2*

The proof follows the lines of theorem 1. However it is worth considering a particular case: one message send and receive on the same queue.

$$\begin{array}{ccc} q_{0,0}, \ldots, e \cdot q_{i,j}, \ldots, q_{n,n} & \xrightarrow{?e_{i\rightarrow j}} & q_{0,0}, \ldots, q_{i,j}, \ldots, q_{n,n} \\ \downarrow !f_{i\rightarrow j} & & \downarrow !f_{i\rightarrow j} \\ q_{0,0}, \ldots, e \cdot q_{i,j} \cdot f, \ldots, q_{n,n} & \xrightarrow{?e_{i\rightarrow j}} & q_{0,0}, \ldots, q_{i,j} \cdot f, \ldots, q_{n,n} \end{array}$$

It should be noted that the FIFO assumption is crucial, since desequencing communication channels do not fit the requirements of hypotheses 1 and 2.

2.3 Distribution proof bases

2.3.1 Correspondence between sequential and distributed program states

Let us now consider programs in a language $\mathcal{L}$ computing over variables. The set of variables is denoted $\mathcal{V}$ and each variable ranges over a single domain $\mathcal{D}$.

The state of a process $P_i, i \in I$ is represented by $q_i =< S_i, \sigma_i >$ where $S_i \in \mathcal{L}$ is the continuation program and $\sigma_i : \mathcal{V} \longrightarrow \mathcal{D}$ is an environment.

The communication medium of a parallel program is assumed to be a complete network of point-to-point FIFO channels, modeled by P_0^F.

Given a distribution mapping and a compilation function, the state correspondence connection $\rightsquigarrow$ is defined as follows:

Definition 7 *Let* $\pi : \mathcal{V} \longrightarrow 2^I \setminus \{\emptyset\}$, $[\![\cdot]\!] : \mathcal{L} \longrightarrow \mathcal{L}^I$ *be respectively a distribution mapping and a compilation function.*
The connection $\rightsquigarrow \subseteq Q_P \times Q_{P_0^F} \times \prod_{i\in I} Q_{P_i}$ *associates to any sequential state some particular parallel states (see figure 1). It is defined as follows:*

$$< S, \sigma > \rightsquigarrow \left(q_0, (< S_i, \sigma_i >)_{i\in I}\right) \iff \begin{cases} q_0 = (\emptyset, \ldots, \emptyset) \\ (S_i)_{i\in I} = [\![S]\!] \\ \forall v \in \mathcal{V}, \forall i \in \pi(v),\ \sigma(v) = \sigma_i(v) \end{cases}$$

2.3.2 Principle of the correctness proof

The aim of the correctness proof is to show that the semantics of programs is preserved by compilation. The semantics we consider is an "input-output" semantics in which only initial and maximal states are meaningful. Definition 8 defines this semantics, and definition 9 defines the correctness property.

Definition 8 *The meaning of a transition system P is a mapping which associates to an initial state q, a set of maximal states.*

$$\mathcal{M}_P(q) = \{q' | q \longrightarrow_P^\star q',\ q' maximal\} \cup \{\bot | q \longrightarrow_P^\omega\}$$

The symbol $\bot$ denotes the divergence of the transition system.

Assuming that $\bot \rightsquigarrow \bot$, the correctness of the compilation function is defined as follows:

Definition 9 *The compilation function $[\![\cdot]\!] : \mathcal{L} \longrightarrow \mathcal{L}^I$ is correct if and only if for any sequential program P:*

$$\forall q_1 \in Q_P,\ \forall q_2 \in Q_Q,\ \forall q_1' \in \mathcal{M}_P(q_1),\ \forall q_2' \in \mathcal{M}_Q(q_2),\ q_1 \rightsquigarrow q_2 \Longrightarrow q_1' \rightsquigarrow q_2'$$

Where $Q = P_0^F \times \prod_{i \in I} [\![P]\!]_i$.

It should be noted that the meaning of an internally deterministic confluent program is reduced to one element: either $\bot$ or a unique final state (theorem 3).

So, the correctness of a compilation function becomes:

Lemma 1 *Let assume that the language $\mathcal{L}$ verifies the hypotheses 1 and 2. The compilation function $[\![\cdot]\!] : \mathcal{L} \longrightarrow \mathcal{L}^I$ is correct if and only if for any program $S \in \mathcal{L}$ and any initial environment $\sigma : \mathcal{V} \longrightarrow \mathcal{D}$.*

- *If $\mathcal{M}(\langle S, \sigma \rangle) \neq \{\bot\}$, the following diagram holds:*

$$\begin{array}{ccc} \langle S, \sigma \rangle & \rightsquigarrow & \left\langle q_0, (\langle S_i, \sigma_i \rangle)_{i \in I} \right\rangle \\ \downarrow_\star & & \downarrow_\star \\ \langle S', \sigma' \rangle & \rightsquigarrow & \left\langle q_0', (\langle S_i', \sigma_i' \rangle)_{i \in I} \right\rangle \end{array}$$

  With $\langle S', \sigma' \rangle$ and $\langle S_i', \sigma_i' \rangle$ maximal.

- *If $\mathcal{M}(\langle S, \sigma \rangle) = \{\bot\}$, then :*

$$\begin{array}{ccc} \langle S, \sigma \rangle & \rightsquigarrow & \left\langle q_0, (\langle S_i, \sigma_i \rangle)_{i \in I} \right\rangle \\ \downarrow_\omega & & \downarrow_\omega \end{array}$$

Since $\mathcal{L}$ verifies hypotheses 1 and 2, theorem 1 applies and the meaning of a program, sequential or parallel, is reduced to a single element

Therefore lemma 1 is straightforward from definition 9 and theorem 3.

We will inductively show either correspondence from an elementary diagram which associates a finite, non empty parallel computation to a single step of the source sequential program. That is, for one step of the sequential program, there exists a particular non empty finite computation of the parallel program that preserves the state correspondence. This can be formalized by the following diagram:

$$\begin{array}{ccc} \langle S, \sigma \rangle & \rightsquigarrow & \left\langle q_0, (\langle S_i, \sigma_i \rangle)_{i \in I} \right\rangle \\ \downarrow & & \downarrow_+ \\ \langle S', \sigma' \rangle & \rightsquigarrow & \left\langle q'_0, (\langle S'_i, \sigma'_i \rangle)_{i \in I} \right\rangle \end{array}$$

This property relies on the semantics of the language $\mathcal{L}$ and on the compilation function $[\![\cdot]\!]$. It will be proved by inference on the rules of the operational semantics of $\mathcal{L}$ and on the axiomatic definition of $[\![\cdot]\!]$.

3 Parallelization

3.1 Language syntax and semantics

We present the process language $\mathcal{L}$ (the sequential language being the same one, but restricted to one processor). It looks very simple but contains all the features of an usual sequential language with respect to the control structure.

3.1.1 Syntax

A parallel program is defined as a collection $(P_i)_{i \in I}$ of processes generated by the following grammar in BNF notation:

$\mathcal{L}$ ::= *statement**
statement ::= $v := expr$
| **if** *expr* **then** $\mathcal{L}$ **else** $\mathcal{L}$ **fi**
| **while** *expr* **do** $\mathcal{L}$ **od**
| $!v_{i \to j}$
| $?v_{j \to i}$

We assume that the domain of values $\mathcal{D}$ contains an undefined value $\perp$. For a matter of readability, we ignore the question of types, which presents no difficulties.

In the sequel we will use ε and b to range over *expr*, S over $\mathcal{L}$ and s over *statement*.

3.1.2 Operational semantics

The operational semantics of the process language defines for each process i a transition relation $\longrightarrow_i \subseteq Q_{P_i} \times A_i \times Q_{P_i}$, where $A_i = \Sigma_i \oplus \{v := \varepsilon \ , \ b\}$, $v := \varepsilon$ and b denoting internal transitions.

Inference rules are as follows:

assignment rule

$$< v := \epsilon\ ,\ \sigma_i > \xrightarrow{v := \epsilon}_i < nil,\ \sigma_i[\sigma_i(\epsilon)/v] >$$

conditional rules

$$\frac{\sigma_i(b)}{< \textbf{if } b \textbf{ then } S_i^t \textbf{ else } S_i^f \textbf{ fi}\ ,\ \sigma_i > \xrightarrow{b}_i < S_i^t\ ,\ \sigma_i >}$$

$$\frac{\neg\sigma_i(b)}{< \textbf{if } b \textbf{ then } S_i^t \textbf{ else } S_i^f \textbf{ fi}\ ,\ \sigma_i > \xrightarrow{b}_i < S_i^f\ ,\ \sigma_i >}$$

iteration rules

$$\frac{\sigma_i(b)}{< \textbf{while } b \textbf{ do } S_i \textbf{ od}\ ,\ \sigma_i > \xrightarrow{b}_i < S_i\ ;\ \textbf{while } b \textbf{ do } S_i \textbf{ od}\ ,\ \sigma_i >}$$

$$\frac{\neg\sigma_i(b)}{< \textbf{while } b \textbf{ do } S_i \textbf{ od}\ ,\ \sigma_i > \xrightarrow{b}_i < nil\ ,\ \sigma_i >}$$

concatenation rule

$$\frac{< s_i\ ,\ \sigma_i > \xrightarrow{a}_i < S_i'\ ,\ \sigma_i' >}{< s_i\ ;\ S_i\ ,\ \sigma_i > \xrightarrow{a}_i < S_i'\ ;\ S_i\ ,\ \sigma_i' >}$$

message send rule

$$< !v_{i\to j}\ ,\ \sigma_i > \xrightarrow{!\sigma_i(v)_{i\to j}}_i < nil\ ,\ \sigma_i >$$

message receive rule

$$< ?v_{j\to i}\ ,\ \sigma_i > \xrightarrow{?e_{j\to i}}_i < nil\ ,\ \sigma_i[e/v] >$$

3.1.3 Verification of the deterministic hypotheses

The transition system defined by the operational semantics of the process language verifies the hypothesis 1:
$\forall i \in I$

$$\begin{array}{c} \det^{ext}(P_i/\rho, A_i/\rho) \\ \det^{int}(P_i) \end{array}$$

The proof is straightforward. Moreover, lemma 1 can be used since P_0^F verifies hypotheses 1 and 2.

3.2 Compilation of the basic refresh mechanism

This section presents the compilation of a sequential program into a parallel one, both in $\mathcal{L}$. We first define the notations for the distribution of data, basis of the program generation in a SPMD model. Then we describe the compilation rules, relying on the notion of *refresh* of remote variables, as it was implemented in the PANDORE compiler [1] of our team.

3.2.1 Data distribution

A distribution is a tuple $< I, \pi, \varphi >$ where:

- I is the set of processors.
- $\pi : \mathcal{V} \longrightarrow 2^I \setminus \{\emptyset\}$ is a mapping that associates to each variable v a set of processors such that $\forall i \in \pi(v)$, i owns v
- $\varphi : \begin{array}{l} \mathcal{V} \times I \longrightarrow I \\ \forall i \notin \pi(v), \varphi(v,i) \in \pi(v) \end{array}$
  φ is a mapping which, to each couple made of a variable v and a processor i that does not own v, associates a processor $\varphi(v,i)$ owning v, which is responsible for the refresh of v on i (updated values of v are sent from $\varphi(v,i)$ to i).

From now on, we will implicitly consider a given data distribution (the implementation of the mappings π and φ being left to the compiler writer).

We also introduce a notion of context: a context is any mapping from I to $2^{\mathcal{V}}$. For instance, μ is the context which associates to each processor i the set of variables owned by i:

$$\mu(i) = \{v \in \mathcal{V} \mid i \in \pi(v)\}$$

The operations on sets are extended to contexts. Let d, d' be two contexts, $V \subseteq \mathcal{V}$ and $\bullet \in \{\cap, \cup, \setminus\}$.

$$\begin{cases} d \bullet d' &= (d(i) \bullet d'(i))_{i \in I} \\ d \bullet V &= (d(i) \bullet V)_{i \in I} \end{cases}$$

3.2.2 Notations for the compilation

Let Δ : *expr* $\longrightarrow 2^{\mathcal{V}}$ and ξ_i : *statement* $\longrightarrow 2^{\mathcal{V}}$ be two mappings giving respectively the set of variables occurring in an expression, and the variables to be refreshed on the processor i for any statement:

$\forall i \in I$

$$\xi_i(v := \varepsilon) = \begin{cases} \Delta(\varepsilon) \setminus \mu(i) & \text{if } i \in \pi(v) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\xi_i(\textbf{if } b \textbf{ then } S^t \textbf{ else } S^f \textbf{ fi}) = \Delta(b) \setminus \mu(i)$$

$$\xi_i(\textbf{while } b \textbf{ do } S \textbf{ od}) = \Delta(b) \setminus \mu(i)$$

and finally ξ is the mapping: *statement* $\longrightarrow (I \longrightarrow 2^{\mathcal{V}})$ such that

$$\xi(s)(i) = \xi_i(s)$$

In order to properly define the *refresh* construction, we now need a mapping which associates to a context a sequence of communication statements. The execution of this sequence achieves the transmission of the values of every variable in the context. Any transmission takes place from one owner of the variable to a processor which needs it.

Let $F = \{!v_{j \to k} \ , \ ?v_{j \to k}\}_{j,k \in I, v \in \mathcal{V}}$ and $\Gamma : (I \longrightarrow 2^{\mathcal{V}}) \longrightarrow F^*$.

Γ is any mapping[1] such that for any context c, $\Gamma(c)$ is a word in which no letter appears twice; thus Γ defines a totally ordered set $(\Gamma(c), <_{\Gamma(c)})$. Moreover, we assume that it checks the following conditions:

$$\Gamma(c) = \{!v_{\varphi(v,j)\to j}\ ,\ ?v_{\varphi(v,j)\to j}\}_{j\in I, v\in c(j)}$$

and $\forall\ !v_{i\to j}, ?v_{i\to j}, !u_{i\to j}, ?u_{i\to j} \in \Gamma(c)$

$$\begin{array}{c} !v_{i\to j} <_{\Gamma(c)} ?v_{i\to j} \\ !u_{i\to j} <_{\Gamma(c)} !v_{i\to j} \iff ?u_{i\to j} <_{\Gamma(c)} ?v_{i\to j} \end{array}$$

The order $<_{\Gamma(c)}$ is such that each emission statement of a variable v occurs "before" the corresponding reception statement, and the order between the emissions of two variables is coherent with the order of their reception. Therefore, deadlocks are avoided.

The pr_i construct is the projection of a communication sequence $\Gamma(c)$ on each processor i, defined as follows:

$$\left\{\begin{array}{rcll} pr_i(nil) & = & nil & \\ pr_i(!v_{i\to j}.r) & = & !v_{i\to j}; pr_i(r) & \\ pr_i(?v_{j\to i}.r) & = & ?v_{j\to i}; pr_i(r) & \\ pr_i(!v_{k\to l}.r) & = & pr_i(r) & \text{if } k \neq i \\ pr_i(?v_{k\to l}.r) & = & pr_i(r) & \text{if } l \neq i \end{array}\right.$$

We can now define the refresh function:

$$refresh_i\ =\ pr_i \circ \Gamma$$

And finally, $exec_i$ is a projection which ensures the execution of an assignment only on the processors owning its left-hand side variable:

$$exec_i\,(v := \varepsilon) = \left\{\begin{array}{ll} v := \varepsilon & \text{if } i \in \pi(v) \\ nil & \text{otherwise} \end{array}\right.$$

3.2.3 Compilation rules

The compilation of a sequential program S results in a parallel program $(\llbracket S \rrbracket_i)_{i\in I}$. The function $\llbracket . \rrbracket_i : \mathcal{L} \longrightarrow \mathcal{L}$ is defined by the following rules:

nil rule

$$\llbracket nil \rrbracket_i = nil$$

concatenation rule

$$\llbracket s; S \rrbracket_i = \llbracket s \rrbracket_i ; \llbracket S \rrbracket_i$$

assignment rule

$$\llbracket v := \varepsilon \rrbracket_i = \left[\begin{array}{l} refresh_i\,(\xi\,(v := \varepsilon))\,; \\ exec_i\,(v := \varepsilon) \end{array}\right.$$

[1]The choice of a particular Γ is a question of implementation.

conditional rule

$$[\![\textbf{if } b \textbf{ then } S^t \textbf{ else } S^f \textbf{ fi}]\!]_i = \left[\begin{array}{l} \mathit{refresh}_i\left(\xi\,(\textbf{if } b \textbf{ then } S^t \textbf{ else } S^f \textbf{ fi})\right); \\ \textbf{if } b \textbf{ then } [\![S^t]\!]_i \textbf{ else } [\![S^f]\!]_i \textbf{ fi} \end{array}\right.$$

iteration rule

$$[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!]_i = \left[\begin{array}{l} \mathit{refresh}_i\,(\xi\,(\textbf{while } b \textbf{ do } S \textbf{ od})); \\ \textbf{while } b \textbf{ do} \\ \quad [\![S]\!]_i;\ \mathit{refresh}_i\,(\xi\,(\textbf{while } b \textbf{ do } S \textbf{ od})) \\ \textbf{od} \end{array}\right.$$

3.3 Correctness proof of the compilation

Theorem 5 *Let* $q =< s; S, \sigma >$, $q' =< S'; S, \sigma' >$ *and* $\tilde{q}$ *be such that*

$$\begin{array}{l} q \rightsquigarrow \tilde{q} \\ \downarrow \\ q' \end{array}$$

Then there exists a non-empty sequence of $\longrightarrow$*-transitions* $\tilde{q} \longrightarrow^+ \tilde{q}'$ *such that* $q' \rightsquigarrow \tilde{q}'$

Proof
Let $\tilde{q} = (q_0, (< [\![s; S]\!]_i, \sigma_i >)_{i \in I})$.
For all $i \in I$, let us note χ_i such that:

$$[\![s; S]\!]_i = \mathit{refresh}_i\,(\xi(s))\,; \chi_i(s); [\![S]\!]_i$$

The activation of the refreshment sequence $\Gamma\,(\xi(s))$ corresponds to a first (possibly empty) sequence of $\longrightarrow$-transitions, such that,

$$\tilde{q} \longrightarrow^* \tilde{q}'' = (q_0, (< \chi_i(s); [\![S]\!]_i, \sigma_i'' >)_{i \in I})$$

The definition of Γ ensures that this computation can be achieved.
From $q \rightsquigarrow \tilde{q}$ and from the definition of *refresh*, we obtain

$$\forall v \in \mathcal{V}, \forall i \in \pi(v),\ \sigma_i''(v) = \sigma(v) \tag{1}$$

$$\forall i \in I, \forall v \in \xi_i(s),\ \sigma_i''(v) = \sigma''_{\varphi(v,i)}(v) \tag{2}$$

and from 1 and 2 we deduce
if $s = v := \epsilon$

$$\forall i \in \pi(v),\ \sigma_i''(\epsilon) = \sigma(\epsilon) \tag{3}$$

and if s is a conditional or an iteration,

$$\forall i \in I,\ \sigma_i''(b) = \sigma(b) \tag{4}$$

There is at least one $\chi_i(s)$ which is different from nil, therefore the execution of one step of each $\chi_i(s) \neq nil$, in any order, produces a non-empty sequence $\tilde{q}'' \longrightarrow^+ \tilde{q}'$ where $\tilde{q}' = (q_0', (< S_i'; [\![S]\!]_i, \sigma_i' >)_{i \in I})$. Let us show that $q' \rightsquigarrow \tilde{q}'$.

a) Queues are empty in state $\tilde{q}''$, because each message has been received, and so they remain empty, therefore $q'_0 = q_0$.

b)

$$\forall v \in \mathcal{V}, \forall i \in \pi(v),\ \sigma'_i(v) = \sigma'(v) \tag{5}$$

The proof, by inference on the compilation rules, is straightforward. The only case in which the environment is modified is the assignment: in this case we have immediately 5 from 3.

c)

$$\forall i \in I, S'_i; [\![S]\!]_i = [\![S'; S]\!]_i \tag{6}$$

By inference on the compilation rules and thanks to the trivial property of the compilation function: $[\![S']\!]_i ; [\![S]\!]_i = [\![S'; S]\!]_i$, we have:

- $s = v := \varepsilon$.
  Then $S' = nil$ and $\chi_i(s) = exec_i(s)$ is rewritten into $S'_i = nil$.
- $s =$ **if** b **then** S^t **else** S^f **fi**.
  Then if $\sigma(b)$ holds, we have $S' = S^t$. From 4 we deduce that $\chi_i(s) =$ **if** b **then** $[\![S^t]\!]_i$ **else** $[\![S^f]\!]_i$ **fi** is transformed into $[\![S^t]\!]_i$, i.e. $S'_i = [\![S^t]\!]_i = [\![S']\!]_i$. The case $\neg\sigma(b)$ is similar.
- $s =$ **while** b **do** S'' **od**.
  Then if $\sigma(b)$ holds, we have s rewritten into $S' = S''; s$. From 4 we deduce that
  $\chi_i(s) =$ **while** b **do** $[\![S'']\!]_i ; refresh_i(\xi(s))$ **od** is transformed into
  $S'_i = [\![S'']\!]_i ; refresh_i(\xi(s)) ; \chi_i(s) = [\![S'']\!]_i ; [\![s]\!]_i = [\![S''; s]\!]_i = [\![S']\!]_i$.
  If $\neg\sigma(b)$ then s becomes nil and so, from 4, $\chi_i(s)$ also becomes nil, i.e. $S'_i = nil$.

□

As an immediate consequence of lemma 1 and theorem 5, we have the main result of this paper:

Theorem 6 *The compilation function* $[\![.]\!] : \mathcal{L} \longrightarrow \mathcal{L}^I$ *is correct.*

4 Illustrations of the generality of the approach

This section is devoted to a few optimizations and extensions of the compilation rules given in section 3.2.3, in order to show that our model can also be used as a tool for exploring other more complex parallelization techniques.

Section 4.1 deals with an optimization which aims to reduce the number of message exchanges. Section 4.2 raises the problem of array data structures in the language $\mathcal{L}$. Lastly an optimization consisting in an anticipation of message send is given in section 4.3.

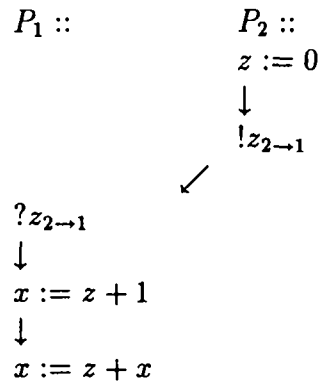
4.1 Optimization of the refresh mechanism

The basic refresh mechanism is not optimal in terms of communications. For instance the following example contains a useless send and receive pair:

$$\left|\begin{array}{ll} P :: & z := 0 \\ & x := z + 1 \\ & x := z + x \end{array}\right. , \left\{\begin{array}{l} I = \{1,2\} \\ \pi(x) = \{1\} \\ \pi(z) = \{2\} \end{array}\right. \Rightarrow \left|\begin{array}{ll} P_1 :: & ?z_{2\to 1} \\ & x := z + 1 \\ & ?z_{2\to 1} \\ & x := z + x \end{array}\right. , \left|\begin{array}{ll} P_2 :: & z := 0 \\ & !z_{2\to 1} \\ & !z_{2\to 1} \end{array}\right.$$

Indeed the second communication pair ($!z_{2\to 1}$ and $?z_{2\to 1}$) between the assignment of P_2 and the second assignment of P_1 is useless since the variable z is not modified in the meanwhile.

In other words, the refresh mechanism ensures the causality between assignments that cannot be swapped — for instance the assignments $z := 0$ and $x := z+1$ do not commute, therefore they must be causally ordered in P_1 and P_2. However the second communication pair is useless since $z := 0$ and $x := z + x$ are already causally related by transitivity:



The principle of the optimization is to delete refresh statements that are known at compile time to be useless. An optimal refresh mechanism can be achieved in this way, at the expense of an exponential code inflation. The optimization presented in this section is a compromise between the raw refresh and an optimal one: it does not produce a great amount of code and is far better than the raw refresh, yet not optimal.

The compilation function is defined with respect to a context $d : I \longrightarrow 2^{\mathcal{V}}$, which associates to a process $i \in I$ the set of variables $d_i \subseteq \mathcal{V}$ that are assumed to be refreshed on i.

The approximation lies in the conditional and iteration rules: for instance, the context assumed after the execution of a **if** ... **then** ... **else** ... **fi** statement is the intersection of contexts obtained by execution of the **then** and **else** parts.

4.1.1 Compilation rules

The compilation function is defined by $[\![\cdot]\!]_i = [\![\cdot]\!]_i^{\emptyset}$, where:

$$[\![\cdot]\!]_i^{\cdot} : \begin{array}{ccc} \mathcal{L} \times \left(I \to 2^{\mathcal{V}}\right) & \longrightarrow & \mathcal{L} \times \left(I \to 2^{\mathcal{V}}\right) \\ (S, d) & \longmapsto & [\![S]\!]_i^d \end{array}$$

Nil rule

$$[\![nil]\!]_i^d = (nil, d)$$

Concatenation rule

$$\frac{(S_1, \delta) = [\![s]\!]_i^d \ , \ (S_2, d') = [\![S]\!]_i^\delta}{[\![s; S]\!]_i^d = (S1;S2, d')}$$

Assignment rule

$$[\![v := \varepsilon]\!]_i^d = (refresh_i\,(\xi\,(v{:=}\varepsilon) \setminus d)\,; exec_i\,(v := \varepsilon) \ , \ (d \cup \xi(v{:=}\varepsilon)) \setminus \{v\})$$

Conditional rule

$$\frac{(S'_t, \delta_t) = [\![S_t]\!]_i^{d \cup \xi(b)} \ , \ \left(S'_f, \delta_f\right) = [\![S_f]\!]_i^{d \cup \xi(b)}}{[\![\textbf{if } b \textbf{ then } S_t \textbf{ else } S_f \textbf{ fi}]\!]_i^d = \left(\left[\begin{array}{l} refresh_i\,(\xi(b) \setminus d)\,; \\ \textbf{if } b \textbf{ then } S'_t \textbf{ else } S'_f \textbf{ fi} \end{array}\right], \delta_t \cap \delta_f\right)}$$

Iteration rule

$$\frac{(S', d') = [\![S]\!]_i^{d \cup \xi(b)} \ , \ (S'', d'') = [\![S]\!]_i^{(d \cap d') \cup \xi(b)}}{[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!]_i^d = \left(\left[\begin{array}{l} refresh_i\,(\xi(b) \setminus d)\,; \\ \textbf{while } b \textbf{ do } S''\,; \\ refresh_i\,(\xi(b) \setminus d'')\ \textbf{od} \end{array}\right], (d \cap d'') \cup \xi(b)\right)}$$

4.1.2 Extended operational semantics

For the purpose of proving the correctness of these compilation rules, we need to extend the operational semantics of the language. This extension consists in adding a context to the state of a sequential or parallel program: this context represents the set $d(i)$ of refreshed variables on each process $i \in I$. However the behavior of a program is not changed, since we show below that, given an initial context, there exists a one to one mapping between extended computations and computations given by the operational semantics of the section 3.1.2.

The following three rules define the changes in the context according to the events that occur in the program:

$$\frac{q \xrightarrow{v := \varepsilon} q'}{\langle q, d\rangle \xrightarrow{v := \varepsilon} \langle q', (d \cup \xi\,(v{:=}\varepsilon)) \setminus \{v\}\rangle}$$

$$\frac{q \xrightarrow{b} q'}{\langle q, d\rangle \xrightarrow{b} \langle q', d \cup \xi(b)\rangle}$$

$$\frac{q \xrightarrow{\alpha} q' \ , \ \alpha \in \Sigma}{\langle q, d\rangle \xrightarrow{\alpha} \langle q', d\rangle}$$

It should be noted that communications do not modify the context.

Any extended transition relation is in one to one correspondence with a transition of the operational semantics of the language. This one to one mapping is the projection $\langle\langle S, \sigma\rangle, d\rangle \longmapsto \langle S, \sigma\rangle$. Therefore the basic computations set and the extended computations set are isomorphic.

The following rule enables us to forget that such and such a variable has been refreshed, in order to establish a correspondence between the context assumed at compile time and the context obtained by simulation for a given statement:

$$\langle q, (d(0), \ldots, d(i) \oplus \{v\}, \ldots, d(n))\rangle \rhd \langle q, d\rangle$$

We hereafter consider computations consisting in an interleaving of $\longrightarrow\!\!\!\!\!\rightarrow$ and $\rhd$ transitions.

4.1.3 Extended state correspondence

The state correspondence should be modified in order to take into account the extension of the state of a parallel or sequential program. It is defined as follows:

Definition 10

- *The vector of programs $(S_i)_{i\in I}$ is said to split when there exists $K \geq 0$, a sequence of programs $\left(S^k\right)_{k=1\ldots K}$ and a sequence of contexts $\left(d^k\right)_{k=1\ldots K}$ such that:*

$$\left\{\begin{array}{l} \forall k = 1\ldots K,\ S^k \neq nil \\ \forall i \in I,\ S_i = [\![S^1]\!]_i^{d^1}; \ldots; [\![S^K]\!]_i^{d^K} \\ \forall k = 1\ldots K-1, \\ \forall \nu : \mathcal{V} \to \mathcal{D} \end{array}\right. \left| \begin{array}{l} \left(\exists \nu',\ \exists \delta^k,\ \left\langle\left\langle S^k, \nu\right\rangle, d^k\right\rangle \longrightarrow\!\!\!\!\!\rightarrow^* \left\langle\langle nil, \nu'\rangle, \delta^k\right\rangle\right) \\ \qquad \Longrightarrow \delta^k \supseteq d^{k+1} \end{array}\right.$$

  Such a couple of sequences is said to be a decomposition of $(S_i)_{i\in I}$.

- *The connection $\overset{\rightsquigarrow}{\rightsquigarrow}$ is defined between a state of a sequential program and a state of a parallel program.*
  $\langle\langle S, \sigma\rangle, d\rangle \overset{\rightsquigarrow}{\rightsquigarrow} \left\langle\left\langle q_0, \langle S_i', \sigma_i'\rangle_{i\in I}\right\rangle, d'\right\rangle$ if and only if:

$$\left\{\begin{array}{l} q_0 = (\emptyset, \ldots, \emptyset) \\ \forall v \in \mathcal{V},\ \forall i \in \pi(v),\ \sigma_i'(v) = \sigma(v) \\ \forall i \in I,\ \forall v \in d'(i),\ \sigma_i'(v) = \sigma(v) \\ d' \subseteq d \\ (S_i')_{i\in I} \text{ splits into } \left(S^k\right)_{k=1\ldots K} \text{ with contexts } \left(d^k\right)_{i=1\ldots K} \\ \text{such that } S = S^1; \ldots; S^K \text{ and } d^1 = d' \end{array}\right.$$

Lemma 2 *The compilation function $[\![\cdot]\!]_i$ is correct if and only if for any program $S \in \mathcal{L}$ and any initial environments σ and $(\sigma_i)_{i\in I}$, if*

$$\langle\langle S, \sigma\rangle, \emptyset\rangle \overset{\rightsquigarrow}{\rightsquigarrow} \left\langle\left\langle q_0, \left\langle [\![S]\!]_i^{\emptyset}, \sigma_i\right\rangle_{i\in I}\right\rangle, \emptyset\right\rangle$$

then either diagrams holds:

•

$$\begin{array}{ccc} \langle\langle S,\sigma\rangle,\emptyset\rangle & \underset{\sim}{\overset{\sim}{\sim}} & \left\langle\left\langle q_0,\left\langle [\![S]\!]_i^{\emptyset},\sigma_i\right\rangle_{i\in I}\right\rangle,\emptyset\right\rangle \\ \downarrow & & \left(\downarrow \cup \bigtriangledown\right)_{\star} \\ \left\langle\langle nil,\hat{\sigma}\rangle,\hat{d}\right\rangle & \underset{\sim}{\overset{\sim}{\sim}} & \left\langle\left\langle q_0,\langle nil,\hat{\sigma}_i\rangle_{i\in I}\right\rangle,\hat{d}'\right\rangle \end{array}$$

•

$$\begin{array}{ccc} \langle\langle S,\sigma\rangle,\emptyset\rangle & \underset{\sim}{\overset{\sim}{\sim}} & \left\langle\left\langle q_0,\left\langle [\![S]\!]_i^{\emptyset},\sigma_i\right\rangle_{i\in I}\right\rangle,\emptyset\right\rangle \\ \downarrow_{\omega} & & \left(\downarrow \cup \bigtriangledown\right)_{\omega} \end{array}$$

Proof From the definition of $\underset{\sim}{\overset{\sim}{\sim}}$ we state:

$$\begin{array}{c} \langle\langle S,\sigma\rangle,\emptyset\rangle \underset{\sim}{\overset{\sim}{\sim}} \left\langle\left\langle q_0,\left\langle [\![S]\!]_i^{\emptyset},\sigma_i\right\rangle_{i\in I}\right\rangle,\emptyset\right\rangle \\ \Longleftrightarrow \\ \langle S,\sigma\rangle \rightsquigarrow \left\langle q_0,\left\langle [\![S]\!]_i^{\emptyset},\sigma_i\right\rangle_{i\in I}\right\rangle \end{array}$$

and:

$$\begin{array}{c} \exists\, \hat{d},\hat{d}',\ \left\langle\langle nil,\hat{\sigma}\rangle,\hat{d}\right\rangle \underset{\sim}{\overset{\sim}{\sim}} \left\langle\left\langle q_0,\langle nil,\hat{\sigma}_i\rangle_{i\in I}\right\rangle,\hat{d}'\right\rangle \\ \Longleftrightarrow \\ \langle nil,\hat{\sigma}\rangle \rightsquigarrow \left\langle q_0,\langle nil,\hat{\sigma}_i\rangle_{i\in I}\right\rangle \end{array}$$

From the extended operational semantics we deduce:

$$\begin{cases} \exists d',\ \langle q,d\rangle\,(\longrightarrow\!\!\!\!\!\rightarrow \cup\ \triangleright)^{\star}\,\langle q',d'\rangle & \Longleftrightarrow \quad q \longrightarrow^{\star} q' \\ \langle q,d\rangle\,(\longrightarrow\!\!\!\!\!\rightarrow \cup\ \triangleright)^{\omega} & \Longleftrightarrow \quad q \longrightarrow^{\omega} \end{cases}$$

Then we conclude by applying lemma 1.

4.1.4 Correctness proof of the compilation rules

The correctness proof is quite similar to the proof of theorem 5. However it is far more tricky since the state correspondence $\underset{\sim}{\overset{\sim}{\sim}}$ is more complex. In this section we will only give parts of the proofs that are different from the proofs of section 3.3.

Let us prove the following lemma:

Lemma 3 *For any pair of corresponding states:*

$$\langle\langle s;S,\sigma\rangle,d\rangle \underset{\rightsquigarrow}{\overset{\rightsquigarrow}{\sim}} \left\langle\left\langle q_0,\langle\Phi_i,\sigma_i\rangle_{i\in I}\right\rangle,d_0\right\rangle$$

The following diagram holds:

$$\begin{array}{ccc} \langle\langle s;S,\sigma\rangle,d\rangle & \underset{\sim}{\overset{\sim}{\sim}} & \left\langle\left\langle q_0,\langle\Phi_i,\sigma_i\rangle_{i\in I}\right\rangle,d_0\right\rangle \\ \downarrow & & \downarrow \\ & & \downarrow_{+} \\ & & \bigtriangledown_{\star} \\ \left\langle\left\langle \hat{S},\hat{\sigma}\right\rangle,\hat{d}\right\rangle & \underset{\sim}{\overset{\sim}{\sim}} & \left\langle\left\langle q_0,\left\langle\hat{\Phi}_i,\hat{\sigma}_i\right\rangle_{i\in I}\right\rangle,\hat{d}_1\right\rangle \end{array}$$

Proof of lemma 3 From the definition of $\tilde{\rightsquigarrow}$ we have:

$$\Phi_i = [\![T]\!]_i^{d_0} ; [\![S_2]\!]_i^{d_2} ; \ldots ; [\![S_K]\!]_i^{d_K}$$

From the concatenation compilation rule, we deduce:

$$[\![T]\!]_i^{d_0} = [\![s]\!]_i^{d_0} ; [\![S_1]\!]_i^{d_1}$$

Since $T;S_2;\ldots;S_K = s;S$, we have:

$$\left\{ \begin{array}{rcl} \Phi_i & = & [\![s]\!]_i^{d_0} ; \Theta_i \\ \Theta_i & = & [\![S_1]\!]_i^{d_1} ; \ldots ; [\![S_k]\!]_i^{d_K} \\ [\![s]\!]_i^{d_0} & = & \underbrace{refresh_i\,(\xi(s) \setminus d_0)}_{R_i} ; \chi_i^{d_0}(s) \end{array} \right.$$

Therefore, by the same reason as in section 3.3:

$$\left\langle \left\langle q_0, \left\langle [\![s]\!]_i^{d_0} ; \Theta_i, \sigma_i \right\rangle_{i \in I} \right\rangle, d_0 \right\rangle \longrightarrow\!\!\!\!\!\rightarrow^{\star} \left\langle \left\langle q_0, \left\langle \chi_i^{d_0}(s) ; \Theta_i, \sigma_i' \right\rangle_{i \in I} \right\rangle, d_0 \right\rangle$$

The *refresh* sequences $(R_i)_{i \in I}$ have been executed, then one step of each non-empty $\chi_i^{d_0}(s)$ should be executed. Let S_i' be their remainder:

$$\forall i \in I \left\{ \begin{array}{ll} S_i' = nil & \text{if } \chi_i^{d_0}(s) = nil \\ \left\langle \chi_i^{d_0}(s), \sigma_i' \right\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle S_i', \hat{\sigma}_i \rangle & \text{if } \chi_i^{d_0}(s) \neq nil \end{array} \right.$$

Since at least one $\chi_i^{d_0}(s)$ is not empty, we have:

$$\left\langle \left\langle q_0, \left\langle \chi_i^{d_0}(s) ; \Theta_i, \sigma_i' \right\rangle_{i \in I} \right\rangle, d_0 \right\rangle \longrightarrow\!\!\!\!\!\rightarrow^{+} \left\langle \left\langle q_0, \langle S_i'; \Theta_i, \hat{\sigma}_i \rangle_{i \in I} \right\rangle, d' \right\rangle$$

By nearly the same reason as in section 3.3, it can be shown that:

$$\left\{ \begin{array}{lrcl} \forall v \in \mathcal{V},\ \forall i \in \pi(v), & \hat{\sigma}(v) & = & \hat{\sigma}_i(v) \\ \forall i \in I,\ \forall v \in d'(i) & \hat{\sigma}(v), & = & \hat{\sigma}_i(v) \\ d' \subseteq \hat{d} & & & \end{array} \right. \tag{7}$$

However, in order to prove the state correspondence, one should prove that $\hat{\Phi}_i = S_i';\Theta_i$ splits in:

$$\left\{ \begin{array}{rcl} \hat{S} & = & \hat{S}_1;\ldots;\hat{S}_L \\ \hat{\Phi}_i & = & [\![\hat{S}_1]\!]_i^{\hat{d}_1} ; \ldots ; [\![\hat{S}_L]\!]_i^{\hat{d}_L} \end{array} \right. \tag{8}$$

It should be also shown that:

$$\hat{d}_1 \subseteq d' \tag{9}$$

so that $\left\langle \left\langle q_0, \langle S_i'; \Theta_i, \hat{\sigma}_i \rangle_{i \in I} \right\rangle, d' \right\rangle \ \triangleright^{\star} \ \left\langle \left\langle q_0, \left\langle \hat{\Phi}_i, \hat{\sigma}_i \right\rangle_{i \in I} \right\rangle, \hat{d}_1 \right\rangle$.

Proof of (8) and (9) The proof of point (8) is a bit tricky: either $\forall i \in I,\ S'_i = nil$ then $\hat{\Phi}_i = \Theta_i$, or $\exists i \in I,\ S'_i \neq nil$, in which case it splits in $S'_i = [\![R_1]\!]_i^{f_1}; \ldots; [\![R_M]\!]_i^{f_M}$.

It should be noticed that in every compilation rule, the context obtained by compilation is smaller than the context obtained by any simulation using the operational semantics. This ensures that the following is a decomposition of $\hat{\Phi}_i = S'_i;\Theta_i$:

$$\hat{\Phi}_i = [\![R_1]\!]_i^{f_1}; \ldots; [\![R_M]\!]_i^{f_M}; [\![S_1]\!]_i^{d_1}; \ldots; [\![S_k]\!]_i^{d_K}$$

Let us denote this decomposition:

$$\hat{\Phi}_i = [\![\hat{S}_1]\!]_i^{\hat{d}_1}; \ldots; [\![\hat{S}_L]\!]_i^{\hat{d}_L}$$

Point (9) is proved by exactly the same argument on contexts obtained by compilation and simulation.

End of proof of lemma 3 From (8), (9) and (7), we can show the state correspondence which closes the following diagram:

$$\begin{array}{ccc}
\langle\langle s;S,\sigma\rangle, d\rangle & \simeq & \left\langle\left\langle q_0, \left\langle [\![s]\!]_i^{d_0};\Theta_i, \sigma_i\right\rangle_{i\in I}\right\rangle, d_0\right\rangle \\
 & & \downarrow \\
 & & \downarrow_\star \\
 & & \left\langle\left\langle q_0, \left\langle \chi_i^{d_0}(s);\Theta_i, \sigma'_i\right\rangle_{i\in I}\right\rangle, d_0\right\rangle \\
\downarrow & & \downarrow \\
\downarrow & & \downarrow_+ \\
 & & \left\langle\left\langle q_0, \langle S'_i;\Theta_i, \hat{\sigma}_i\rangle_{i\in I}\right\rangle, d'\right\rangle \\
 & & \nabla_\star \\
\left\langle\left\langle \hat{S}, \hat{\sigma}\right\rangle, \hat{d}\right\rangle & \simeq & \left\langle\left\langle q_0, \left\langle \hat{\Phi}_i, \hat{\sigma}_i\right\rangle_{i\in I}\right\rangle, \hat{d}_1\right\rangle
\end{array}$$

The diagram of lemma 3 is embedded in this diagram.

End of correctness proof By induction on the diagram of lemma 3, one of the two diagrams of lemma 2 holds, which concludes the correctness proof.

4.2 Extension to array data structures

This section is devoted to the manipulation of arrays, in order to generalize our model. From a theoretical point of view, dealing with arrays does not involve much problems, the main difference being that most of compilation functions produce now some guarded code (the predicates of the basic case being no more evaluable at compile-time).

We first present some slight modifications in the notations used so far, then we describe a new "dynamic" version of the *refresh* function, we give a sketchy proof of correctness of this new model and we conclude by an example.

4.2.1 Notations and extensions

We modify $\mathcal{V}$ by introducing a syntactical distinction between scalar variables and array elements : $\mathcal{V} = \mathcal{V}_s \cup \mathcal{V}_a$.

$\mathcal{V}_s$: scalar variables

$$\mathcal{V}_a = \{A[e_1, \ldots, e_n] \mid A : \text{array}, e_k \in expr\} : \text{array elements}$$

Extension of σ : let $v = A[e_1, \ldots, e_n] \in \mathcal{V}_a$, then $\sigma(v)$ is in fact $\sigma(A[\sigma(e_1), \ldots, \sigma(e_n)])$.

Furthermore, π and φ are now compilation functions (i.e. they produce code), and are extended in this way :

$\pi(A[e_1, \ldots, e_n]) = \pi(A[\sigma(e_1), \ldots, \sigma(e_n)])$

$\varphi(A[e_1, \ldots, e_n], p) = \varphi(A[\sigma(e_1), \ldots, \sigma(e_n)], p)$

and we will use $\psi : \mathcal{V} \times I \longrightarrow 2^I$ such that $p \in \psi(v, q) \Longleftrightarrow \varphi(v, p) = q$.

Lastly we need a new syntactic function, similar to Δ, which gives the variables (scalars and/or array elements) occurring as index of arrays in an expression:

$$\Delta_{ind} : expr \longrightarrow 2^{\mathcal{V}}$$

For instance, if $e = A[i + j] * B[C[j], 3 * k] + l - 2$ then $\Delta_{ind}(e) = \{i, j, C[j], k\}$.

4.2.2 Generalization of the refresh

The principle of the generalized *refresh* is to evaluate the value of every index of an array element before the evaluation of the value of the element itself. As we consider a general framework, without restrictions to the case where such indices are known at compile-time, we also have to deal with the case of an array index being itself an array element. Therefore, the *refreshgen* is a recursive compilation function that calls a slightly different version of *refresh* : the new refresh is a function creating code with guards.

$$refreshgen_i\,(V, P) \equiv \left| \begin{array}{ll} \left\{ \begin{array}{l} refreshgen_i\,(V' \cap \mathcal{V}_a, I)\,; \\ refresh_i^r\,(V' \cap \mathcal{V}_s, I)\,; \\ refresh_i^r\,(V, P) \\ \quad \text{with } V' = \bigcup_{v \in V} \Delta_{ind}(v) \end{array} \right\} & \text{if } V \neq \emptyset \\ \text{nil} & \text{if } V = \emptyset \end{array} \right.$$

where $refresh_i^r = pr_i^r \circ \Gamma^r$, pr_i^r and Γ^r being modified versions of pr_i and Γ respectively.

The function Γ^r produces a sequence of guarded communication routines[2]:

$$\Gamma^r(V, P) = \left\{ \begin{array}{l} \mathcal{G}(k, P, !v_{me \to k}), \\ \mathcal{G}(me, P, ?v_{\varphi(v,me) \to me}) \end{array} \right\}_{me \in I, v \in V, k \in \psi(v,me)}$$

where $\mathcal{G} : I \times 2^I \times statement \longrightarrow statement$ is a compilation function that produces guarded code in a dynamic case and unguarded code otherwise:

- if the value of P is known at compile-time (i.e. $P = I$ or $P = \pi(v)$ with $v \in \mathcal{V}_s$),

$$\mathcal{G}(i, P, s) = \left| \begin{array}{ll} s & \text{if } i \in P \\ nil & \text{otherwise} \end{array} \right.$$

- if the value of P is known only at runtime,

$$\mathcal{G}(i, P, s) = \textbf{if } i \in P \textbf{ then } s \textbf{ fi}$$

[2]for the sake of brevity, we have chosen to present a "naive" Γ^r. Indeed, it is possible to implement one that will produce a far more efficient code.

The function Γ^r checks the same conditions as in the basic case, and $pr_i^r(\Gamma^r(V,P))$ keeps all constructions of Γ^r with $me{=}\ i$.

We also need a trivial runtime version of *exec* :

$$exec_i^r(v := \varepsilon) = \mathcal{G}(i, \pi(v), v := \varepsilon)$$

4.2.3 Compilation rules

In the conditional and iterative rules, the only modification to deal with is to replace *refresh* by *refreshgen*. For the assignment rule, we have now to compute the refresh of the left-side variable : if it is an array element, then its index has to be evaluated.

assignment rule

$$[\![v := \varepsilon]\!]_i = \left[\begin{array}{l} refreshgen_i\,(\Delta_{ind}(v), I)\,; \\ refreshgen_i\,(\Delta(\varepsilon), \pi(v))\,; \\ exec_i^r(v := \varepsilon) \end{array}\right.$$

condition rule

$$[\![\textbf{if } b \textbf{ then } S^t \textbf{ else } S^f \textbf{ fi}]\!]_i = \left[\begin{array}{l} refreshgen_i\,(\Delta(b), I)\,; \\ \textbf{if } b \textbf{ then } [\![S^t]\!]_i \textbf{ else } [\![S^f]\!]_i \textbf{ fi} \end{array}\right.$$

iterative rule

$$[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!]_i = \left[\begin{array}{l} refreshgen_i\,(\Delta(b), I)\,; \\ \textbf{while } b \textbf{ do } [\![S]\!]_i\,;\, refreshgen_i\,(\Delta(b), I) \textbf{ od} \end{array}\right.$$

4.2.4 Correctness proof outline

Case of the scalar variables : for a set $V \subseteq \mathcal{V}_s$, we have the immediate result:

$$refreshgen_i\,(V,P) \equiv refresh_i^r\,(V,P)$$

Moreover, if we assume that the restriction of Γ^r to $\mathcal{V}_s$ checks the same conditions as Γ (cf. paragraph 3.2.2), then $refresh_i^r$ and $refresh_i$ produce the same communication sequences, and therefore the processes have the same behaviour.

Case of the array elements : according to the definition of the *refreshgen* function, each access to an array element $v = A[e_1, \ldots, e_m]$ with $\Delta_{ind}(v) = \{v_1, \ldots, v_n\}$ (where each v_i is a scalar) is translated into a refresh of all needed variables $v_1, \ldots, v_n$ on every processor and therefore on the owner of v, followed by the refresh of v. Henceforth the correctness of the latter is ensured. Moreover, the recursive style of *refreshgen* maintains this correctness when the v_i are array elements.

4.2.5 Example

To illustrate our model, we present a non-trivial example: the following program is a part of the well-known Cholesky solver algorithm.

```
(* Declarations *)
A : array [1..n,1..n] of real;
x,p :  array [1..n] of real;
i,j,k :  integer;

i:=1;
```

```
while i<=n do
    x[i]:=A[i,i];                                          (1)
    k:=1;
    while k<i do
        x[i]:=x[i]-A[i,k]^2;                               (2)
        k:=k+1
    od;
    p[i]:=1/sqrt(x[i]);                                    (3)
    j:=i+1;
    while j<=n do
        x[i]:=A[i,j];                                      (4)
        k:=1;
        while k<i do
            x[i]:=x[i]-A[i,j]*A[i,k];                      (5)
            k:=k+1
        od;
        A[j,i]:=x[i]*p[i];                                 (6)
        j:=j+1
    od;
    i:=i+1
od
```

As usual in distributed programming, efficiency mostly depends on the mapping of data on the processors; here the distribution is quite obvious: A, x and p are distributed by row, i.e. $\forall i,j \in [1,n], \pi(A[i,j]) = \pi(x[i]) = \pi(p[i]) = \{i\}$ and the scalars i, j, k are distributed on all processors: $\pi(i) = \pi(j) = \pi(k) = \{1, \ldots, n\}$.

Let us consider the assignment (2):

$$refreshgen_l(\{x[i], A[i,k]\}, \pi(x[i])) \equiv pr_l^r(\Gamma^r(\{i,k\}, I)); pr_l^r(\Gamma^r(\{x[i], A[i,k]\}, \pi(x[i])))$$

Notice that $\Gamma^r(\{i,k\}, I)$ contains no message construct because for all l in I, $\psi(i,l) = \psi(k,l) = \emptyset$ can be statically computed. And we have:

$$\Gamma^r(\{x[i], A[i,k]\}, \pi(x[i])) = \left\{ \begin{array}{l} \mathcal{G}(h, \pi(x[i]), !v_{l \to h}), \\ \mathcal{G}(l, \pi(x[i]), ?v_{\varphi(v,l) \to l}) \end{array} \right\}_{v \in \{x[i], A[i,k]\}, h \in \psi(v,l)}$$

The principle is the same for the other assignments; in practice, the compilation produces a lot of guarded message statements, but at run-time, only (6) will produce effective refresh.

Considering that $\mathtt{send}_l(\mathtt{V},\mathtt{J})$ is a function computing the send sequence of the variables of V from l to each processor $p \in \psi(v,l) \cap \mathtt{J}$, and that $\mathtt{receive}_l(\mathtt{V})$ computes the receive sequence on l of each variable v of V from $\varphi(v,l)$, then the code on a processor l will be for instance (it depends in fact on the chosen implementation of Γ^r) the following one:

```
i:=1;
while i<=n do
    send_l({A[i,i]},π(x[i]));
```

```
    G(l,π(x[i]),receive_l({A[i,i]}));
    G(l,π(x[i]),x[i]:=A[i,i]);                                    (1)
    k:=1;
    while k<i do
        send_l({x[i],A[i,k]},π(x[i]));
        G(l,π(x[i]),receive_l({x[i],A[i,k]}));
        G(l,π(x[i]),x[i]:=x[i]-A[i,k]^2);                         (2)
        k:=k+1
    od;
    send_l({p[i]},π(p[i]));
    G(l,π(p[i]),receive_l(x[i]));
    G(l,π(p[i]),p[i]:=1/sqrt(x[i]));                              (3)
    j:=j+1;
    while j<=n do
        send_l({A[i,j]},π(x[i]));
        G(l,π(x[i]),receive_l(A[i,j]));
        G(l,π(x[i]),x[i]:=A[i,j]);                                (4)
        k:=1;
        while k<i do
            send_l({x[i],A[i,j],A[i,k]},π(x[i]));
            G(l,π(x[i]),receive_l({x[i],A[i,j],A[i,k]}));
            G(l,π(x[i]),x[i]:=x[i]-A[i,j]*A[i,k]);                (5)
            k:=k+1
        od;
        send_l({x[i],p[i]},π(A[j,i]));
        G(l,π(A[j,i]),receive_l({x[i],p[i]}));
        G(l,π(A[j,i]),A[j,i]:=x[i]*p[i]);                         (6)
        j:=j+1
    od;
    i:=i+1
od
```

Note that an extension of the optimized refresh mechanism (cf. section 4.1) to arrays would allow the suppression of two useless refresh: those of A[i,j] in assignment (5) and of p[i] in assignment (6).

Beside this, there are of course possible techniques for avoiding at compile-time the computation of useless refresh such as, for instance, the refresh of variable A[i,k] of the processor $\pi(A[i,k])=\{i\}$ towards the processor $\pi(x[i])=\{i\}$. These techniques, based upon domain analysis, are currently under study in our team, and we aim to integrate them in our compiler.

4.3 Anticipation of message send

This section deals with a post-compilation optimization: one should notice that message exchanges are placed as late as possible, i.e. right before the assignment or predicate, the evaluation of which requires some variables to be refreshed. However, it is valuable to

anticipate as much as possible message sending, in order to minimize idle times during the evaluation of receive statements.

This optimization is local to each process, since it takes each process independently and produces an "equivalent" optimized process, with anticipated message send statements.

4.3.1 Optimization rules

Let $\sim$ be the asymmetrical binary relation over programs:

$$\frac{u \neq v}{v{:=}\varepsilon \sim\ !u_{i\to j}} \qquad nil \sim\ !u_{i\to j}$$

$$\frac{u \neq v}{?v_{k\to i} \sim\ !u_{i\to j}} \qquad \frac{l \neq j}{!v_{i\to l} \sim\ !u_{i\to j}}$$

$$\frac{S \sim\ !u_{i\to j}\ ,\ s \sim\ !u_{i\to j}}{S;s \sim\ !u_{i\to j}}$$

$$\frac{S^t \sim\ !u_{i\to j}\ ,\ S^f \sim\ !u_{i\to j}}{\textbf{if } b \textbf{ then } S^t \textbf{ else } S^f \textbf{ fi} \sim\ !u_{i\to j}}$$

$$\frac{S \sim\ !u_{i\to j}}{\textbf{while } b \textbf{ do } S \textbf{ od} \sim\ !u_{i\to j}}$$

This relation defines whether a message send can be swapped with a sequence of code or not. The $\sim$ relation is used to define the optimization function $\{\!|\cdot|\!\} : \mathcal{L} \longrightarrow \mathcal{L}$ which is defined as follows:

$$\{\!|v{:=}\varepsilon|\!\} = v{:=}\varepsilon \qquad \{\!|nil|\!\} = nil$$
$$\{\!|!u_{i\to j}|\!\} = !u_{i\to j} \qquad \{\!|?u_{i\to j}|\!\} = ?u_{i\to j}$$

$$\{\!|\textbf{if } b \textbf{ then } S^t \textbf{ else } S^f \textbf{ fi}|\!\} = \textbf{if } b \textbf{ then } \{\!|S^t|\!\} \textbf{ else } \{\!|S^f|\!\} \textbf{ fi}$$

$$\{\!|\textbf{while } b \textbf{ do } S \textbf{ od}|\!\} = \textbf{while } b \textbf{ do } \{\!|S|\!\} \textbf{ od}$$

$$\frac{\{\!|S|\!\} = S';s'\ ,\ s' \sim s}{\{\!|S;s|\!\} = \{\!|S';s|\!\};s'} \qquad \frac{\{\!|S|\!\} = S';s'\ ,\ s' \not\sim s}{\{\!|S;s|\!\} = S';s';\{\!|s|\!\}}$$

4.3.2 Proof principle

The principle of the correctness proof is to use the confluence property of both parallel programs and to establish a correspondence between any execution of the source parallel program and a particular computation of the optimized program. Definition 12 defines this correspondence. Lemma 4 gives the elementary diagram that should be proved in order to show the correctness (definition 13) of the optimization function.

In first place, a projection over sequences of message send actions is defined:

Definition 11 *Let* $\Phi_{i\to j} : \{!e_{k\to l}\}^{\star}_{k,l\in I,\, e\in E} \longrightarrow \{!e_{k\to l}\}^{\star}_{k,l\in I,\, e\in E}$ *be the function defined below:*

$$\begin{cases} \Phi_{i\to j}(nil) = nil \\ \Phi_{i\to j}(!e_{i\to j}\cdot\lambda) = !e_{i\to j}\cdot\Phi_{i\to j}(\lambda) \\ \dfrac{(i,j)\neq(k,l)}{\Phi_{i\to j}(!e_{k\to l}\cdot\lambda) = \Phi_{i\to j}(\lambda)} \end{cases}$$

Definition 12 *The relation* $\approx$ *is defined as follows:*

$$\left\langle q, \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle \overset{\Delta}{\approx} \left\langle q', \langle S'_i,\sigma'_i\rangle_{i\in I}\right\rangle$$
$$\Longleftrightarrow$$
$$\begin{cases} \forall i\in I,\ \sigma_i = \sigma'_i \\ \exists (\lambda_i)_{i\in I} \in \left(\{!e_{i\to j}\}^{\star}_{e\in E, i,j\in I}\right)^I, \begin{cases} \forall i\in I,\ \lambda_i; S'_i = \{\!\!\{S_i\}\!\!\} \\ \forall i,j\in I,\ \Phi_{i\to j}(\lambda_i) = \Phi_{i\to j}(\Delta) \end{cases} \\ \forall i,j\in I,\ q'_{i,j} = q_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\Delta)\right) \end{cases}$$

Definition 13 *The anticipation function is correct if and only if either diagram holds:*

1. *If* $\mathcal{M}\left(\left\langle(\emptyset,\ldots,\emptyset), \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle\right) \neq \{\bot\}$*:*

$$\begin{array}{ccc} \left\langle(\emptyset,\ldots,\emptyset), \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle & \overset{(\emptyset,\ldots,\emptyset)}{\approx} & \left\langle(\emptyset,\ldots,\emptyset), \langle \{\!\!\{S_i\}\!\!\},\sigma_i\rangle_{i\in I}\right\rangle \\ \downarrow_{\star} & & \downarrow_{\star} \\ \left\langle(\emptyset,\ldots,\emptyset), \langle nil,\hat{\sigma}_i\rangle_{i\in I}\right\rangle & \overset{(\emptyset,\ldots,\emptyset)}{\approx} & \left\langle(\emptyset,\ldots,\emptyset), \langle nil,\hat{\sigma}_i\rangle_{i\in I}\right\rangle \end{array}$$

2. *If* $\mathcal{M}\left(\left\langle(\emptyset,\ldots,\emptyset), \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle\right) = \{\bot\}$*:*

$$\begin{array}{ccc} \left\langle(\emptyset,\ldots,\emptyset), \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle & \overset{(\emptyset,\ldots,\emptyset)}{\approx} & \left\langle(\emptyset,\ldots,\emptyset), \langle \{\!\!\{S_i\}\!\!\},\sigma_i\rangle_{i\in I}\right\rangle \\ \downarrow_{\omega} & & \downarrow_{\omega} \end{array}$$

Lemma 4 *The anticipation is correct if the following diagram holds:*

$$\begin{array}{ccc} \left\langle q, \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle & \overset{\Delta}{\approx} & \left\langle q', \langle S'_i,\sigma'_i\rangle_{i\in I}\right\rangle \\ \downarrow_{\star}\,\mu & & \downarrow_{\star}\,\mu' \\ \downarrow\,\alpha & & \downarrow\,\alpha \\ \left\langle \hat{q}, \left\langle \hat{S}_i,\hat{\sigma}_i\right\rangle_{i\in I}\right\rangle & \overset{\mu^{-1}\cdot\Delta\cdot\mu'}{\approx} & \left\langle \hat{q}', \left\langle \hat{S}'_i,\hat{\sigma}'_i\right\rangle_{i\in I}\right\rangle \end{array}$$

with $\alpha \in \{v{:=}\varepsilon, b, ?e_{i\to j}\}_{i,j\in I,\, v\in\mathcal{V},\, e\in expr,\, e\in E}$ *and* $\mu,\mu' \in \{!e_{i\to j}\}^{\star}_{i,j\in I,\, e\in E}$.

Proof of lemma 4 By induction on the diagram of lemma 4, either diagram holds:

1. If $\mathcal{M}\left(\left\langle(\emptyset,\ldots,\emptyset), \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle\right) \neq \{\bot\}$:

$$\begin{array}{ccc} \left\langle(\emptyset,\ldots,\emptyset), \langle S_i,\sigma_i\rangle_{i\in I}\right\rangle & \overset{(\emptyset,\ldots,\emptyset)}{\approx} & \left\langle(\emptyset,\ldots,\emptyset), \langle S'_i,\sigma_i\rangle_{i\in I}\right\rangle \\ \downarrow_{\star} & & \downarrow_{\star} \\ \left\langle(\emptyset,\ldots,\emptyset), \langle nil,\hat{\sigma}_i\rangle_{i\in I}\right\rangle & \overset{\Delta}{\approx} & \left\langle q, \langle nil,\hat{\sigma}_i\rangle_{i\in I}\right\rangle \end{array}$$

However, from definition 12 we deduce that $\Delta = (\emptyset,\ldots,\emptyset)$, hence $q = (\emptyset,\ldots,\emptyset)$. Therefore the first diagram of definition 13 is matched.

2. If $\mathcal{M}\left(\left\langle(\emptyset,\ldots,\emptyset),\langle S_i,\sigma_i\rangle_{i\in I}\right\rangle\right)=\{\bot\}$:
   The second diagram of definition 13 is matched.

Theorem 7 *The optimization function* $[\![\cdot]\!]:\mathcal{L}\longrightarrow\mathcal{L}$ *is correct.*

Proof of theorem 7 By using lemma 4, it is sufficient to show the following diagram:

$$\begin{array}{ccc}
\left\langle q,\langle S_i,\sigma_i\rangle_{i\in I}\right\rangle & \overset{\Delta}{\approx} & \left\langle q',\langle S'_i,\sigma'_i\rangle_{i\in I}\right\rangle \\
\downarrow_\star\,\mu & & \downarrow_\star\,\mu' \\
\left\langle \tilde{q},\left\langle \tilde{S}_i,\tilde{\sigma}_i\right\rangle_{i\in I}\right\rangle & & \left\langle \tilde{q}',\left\langle \tilde{S}'_i,\tilde{\sigma}'_i\right\rangle_{i\in I}\right\rangle \\
\downarrow\,\alpha & & \downarrow\,\alpha \\
\left\langle \hat{q},\left\langle \hat{S}_i,\hat{\sigma}_i\right\rangle_{i\in I}\right\rangle & \overset{\mu^{-1}\cdot\Delta\cdot\mu'}{\approx} & \left\langle \hat{q}',\left\langle \hat{S}'_i,\hat{\sigma}'_i\right\rangle_{i\in I}\right\rangle
\end{array}$$

With $\mu,\mu'\in\{!e_{i\to j}\}^\star_{i,j\in I,\,e\in E}$

- Clearly, we have:

$$\forall i\in I,\ \hat{\sigma}_i=\hat{\sigma}'_i \tag{10}$$

- From definition 12 and the diagram above, we have:

$$q'_{i,j}=q_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\Delta)\right) \tag{11}$$

$$\tilde{q}_{i,j}=q_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\mu)\right) \tag{12}$$

$$\tilde{q}'_{i,j}=q'_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\mu')\right) \tag{13}$$

From (11) and (13), we deduce:

$$\tilde{q}'_{i,j}=q_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\Delta\cdot\mu')\right) \tag{14}$$

From (12) we have:

$$q_{i,j}=\tilde{q}_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\mu^{-1})\right) \tag{15}$$

Therefore:

$$\tilde{q}'_{i,j}=\tilde{q}_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\mu^{-1}\cdot\Delta\cdot\mu')\right) \tag{16}$$

The execution of α changes equally $\tilde{q}$ and $\tilde{q}'$, therefore:

$$\hat{q}'_{i,j}=\hat{q}_{i,j}\cdot\sigma_i\left(\Phi_{i\to j}(\mu^{-1}\cdot\Delta\cdot\mu')\right) \tag{17}$$

- The sequence $\mu^{-1} \cdot \mu'$ is the sequence of outputs performed in the right hand parallel program and not in the left hand one. From definition 12 we have :

$$\exists (\lambda_i) \in \left(\{!v_{i\to j}\}^{\star}_{v\in\mathcal{V},i,j\in I}\right)^I \text{ such that } \begin{cases} \forall i \in I,\ \{\!|S_i|\!\} = \lambda_i ; S_i' \\ \forall i,j \in I,\ \Phi_{i\to j}(\lambda_i) = \Phi_{i\to j}(\Delta) \end{cases}$$

Let $\left(\hat{\lambda}_i\right) \in \left(\{!e_{i\to j}\}^{\star}_{e\in E,\, i,j\in I}\right)^I$ be such that $\forall i \in I,\ \{\!|\hat{S}_i|\!\} = \hat{\lambda}_i ; \hat{S}_i'$. From (12) and (13) we deduce :

$$\forall i,j \in I\ \ \Phi_{i\to j}(\mu^{-1}) \cdot \Phi_{i\to j}(\lambda_i) \cdot \Phi_{i\to j}(\mu') = \Phi_{i\to j}(\hat{\lambda}_i)$$

Therefore we have:

$$\forall i,j \in I,\ \ \Phi_{i\to j}\left(\hat{\lambda}_i\right) = \Phi_{i\to j}\left(\mu^{-1} \cdot \Delta \cdot \mu'\right) \tag{18}$$

From equations (10), (17) and (18), we have shown that the last line of the diagram matches definition 12. Which concludes the proof of theorem 7.

5 Conclusion

We have presented a formal explanation of automated distribution of sequential programs. Though the technique of data-driven parallelization is widely accepted, it appears that the formal description of its associated compilation rules, the semantic properties to be preserved and the correctness proofs are quite difficult to establish. We think however such a theoretical foundation is necessary to make the discipline in progress.

The main technical contributions of the paper are:

- a formal model based on products of labeled and partially deterministic transition systems, allowing the comparison of the sequential source behaviour with the corresponding parallel ones.
- a proof technique based on confluence, which avoids the combinatorics of considering all the parallel behaviours.
- a complete example of treatment of a simplified sequential language. Compilation rules were detailed enough to permit a straightforward prototype implementation in ML (CAML [11]), generating a distributed code written in ESTELLE [6] (an ISO language to describe communicating processes) and experimented with our favorite distributed environment [7].

By dealing with an optimization of the refresh mechanism and the formal treatment of arrays in the language, we have also tried to show that our contribution may serve as a basis for designing and proving other (and new) parallelizing rules.

6 Acknowledgments

The paper has benefitted by informal discussions with the members of the *PAMPA* team. In particular, we would like to thank Françoise André and Jean-Louis Pazat for proof reading of the paper and criticisms on its practical aspects. Special thanks also to Roderick McConnell.

Many thanks to Paul Caspi and Alain Girault (*IMAG*, Grenoble, France) for the fruitful talks we had with them on the similar problem of distributing reactive programs.

References

[1] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore: a system to manage data distribution. In *ACM International Conference on Supercomputing*, June 11-15 1990.

[2] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

[3] A. Arnold. Transition systems and concurrent processes. *Mathematical Problems in Computation Theory, Banach Center Publications*, 21, 1988.

[4] Luc Bougé. *On the semantics of languages for massively parallel Simd architectures.* Research report 91-14, LIP/ENS Lyon, April 1991.

[5] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, (2):151–169, 1988.

[6] ISO 9074. *Estelle: a Formal Description Technique based on an Extented State Transition Model.* ISO TC97/SC21/WG6.1, 1989.

[7] C. Jard and J.-M. Jézéquel. ECHIDNA, an Estelle-compiler to prototype protocols on distributed computers. *Concurrency Practice and Experience*, June 1992.

[8] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[9] H. R. Nielson and F. Nielson. *Semantics with Applications: a Formal Introduction.* Wiley, 1992.

[10] Edwin M. Paalvast, Henk J. Sips, and A.J. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *International Conference on Parallel Processing*, August 1991.

[11] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. *The CAML reference manual.* Rapport Technique 121, INRIA, septembre 1990.

[12] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt. Superb: a tool for semi-automatic mimd/simd parallelization. *Parallel Computing*, (6):1–18, 1988.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 658 MULTISCALE SIGNAL PROCESSING : ISOTROPIC RANDOM FIELDS ON HOMOGENEOUS TREES
Bernhard CLAUS, Ghislaine CHARTIER
Mai 1992, 28 pages.

PI 659 ON THE COVARIANCE-SEQUENCE OF AR-PROCESSES. AN INTERPOLATION PROBLEM AND ITS EXTENSION TO MULTISCALE AR-PROCESSES
Bernhard CLAUS, Albert BENVENISTE
Mai 1992, 36 pages.

PI 660 EXCEPTION HANDLING IN COMMUNICATING SEQUENTIAL PROCESSES DESIGN, VERIFICATION AND IMPLEMENTATION
Jean-Pierre BANATRE, Valérie ISSARNY
Mai 1992, 38 pages.

PI 661 REACHABILITY ANALYSIS ON DISTRIBUTED EXECUTIONS
Claide DIEHL, Claude JARD, Jean-Xavier RAMPON
Juin 1992, 18 pages.

PI 662 RECONSTRUCTION 3D DE PRIMITIVES GEOMETRIQUES PAR VISION ACTIVE
Samia BOUKIR, François CHAUMETTE
Juin 1992, 40 pages.

PI 663 FILTRES SEMANTIQUES EN CALCUL PROPOSITIONNEL
Raymond ROLLAND
Juin 1992, 22 pages.

PI 664 REGION-BASED TRACKING IN AN IMAGE SEQUENCE
François MEYER, Patrick BOUTHEMY
Juin 1992, 50 pages.

PI 665 CORRECTNESS OF AUTOMATED DISTRIBUTION OF SEQUENTIAL PROGRAMS
Cyrille BAREAU, Benoît CAILLAUD, Claude JARD, René THORAVAL
Juin 1992, 32 pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399