Parallel Implementation Models for the λ-calculus Using the Geometry of Interaction (Extended Abstract)

Jorge Sousa Pinto*

Departamento de Informática
Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
jsp@di.uminho.pt

Abstract. An examination of Girard's execution formula suggests implementations of the Geometry of Interaction at the syntactic level. In this paper we limit our scope to ground-type terms and study the *parallel* aspects of such implementations, by introducing a family of abstract machines which can be directly implemented. These machines address all the important implementation issues such as the choice of an inter-thread communication model, and allow to incorporate specific strategies for dividing the computation of the execution path into smaller tasks.

1 Introduction

This paper proposes novel parallel implementation techniques for the λ-calculus based on the geometry of interaction (GoI) [6, 5, 7]. GoI-based implementation is quite different from other techniques: it uses a graph representation of each term, from which its value is derived by performing *path* computations, which can be done locally and asynchronously. This encompasses both β-reduction and the variable substitution mechanism.

Informally, for every ground-type term there is a path which leaves from the root of the respective graph, and traverses the term, finishing back at root. This path will survive reduction; in particular in the normal form of the term, it will simply go from the root to the constant which is the value of the term. The GoI treats this path algebraically, by assigning a weight to every edge in the initial graph. This allows, on one hand, to identify the unique path which survives reduction, and on the other hand, to calculate algebraically its weight, which is invariant throughout reduction, and equal, in fact, to the value of the term.

The geometry of interaction has been developed as a semantics for linear logic proof-nets [4]. Combined with a standard translation of the λ-calculus into these nets, the results may then be lifted to the scope of functional programs. The nodes in the graph of each term are logical symbols with premises and

* Research done whilst staying at Laboratoire d'Informatique (CNRS UMR 7650), École Polytechnique. Partially supported by PRAXIS XXI grant BD/11261/97.

conclusions, and each orientated edge links a conclusion of a node to a premise of another node. Paths are sequences of (direct $\cdot \longrightarrow \cdot$ or reverse $\cdot \longleftarrow \cdot$) edges. *Straight* paths are those that do not bounce (i.e., no edge is followed by the same edge in the opposite direction) and do not twist (a path arriving at a premise of a node is not followed by an edge leaving from the other premise).

Persistent paths are those that remain invariant with respect to reduction, and the geometry of interaction is a tool for calculating them: Girard's *execution formula* gives the interpretation of a term as a set of straight paths (called *regular* paths) which are proved [3] to be exactly the persistent paths.

Regular paths are calculated algebraically: edges in the graph are labelled with a *weight*, a term in the GoI *dynamic algebra* $\mathcal{L}^*$. The weight $w(\cdot)$ of a path is defined inductively: $w(\varepsilon) = 1$ for the empty path ε, and $w(t\gamma) = w(\gamma) \cdot w(t)$ where γ is a path and t an edge, and $\cdot$ denotes composition in $\mathcal{L}^*$. For the case of ground-type terms a single persistent path exists which starts and ends at the root of the graph, and the term can be evaluated by calculating its weight.

Implementation via GoI. The first work which proposed to use the GoI as an implementation mechanism [2] defined *virtual reduction* (VR), a local and confluent reduction on graphs, which already suggested the use of parallelism. Virtual reduction allows to add to a graph new edges representing composed paths. Since it preserves the execution of terms, VR provides a way of calculating regular paths. In order to avoid compositions corresponding to bouncing paths, as well as the repeated compositions of pairs of edges, virtual reduction *filters* the weights of the composed edges, for which an extension of the algebraic structure is required. This makes the calculations rather complex.

Directed Virtual Reduction [1], applied with the *combustion* strategy, eliminates this complexity and achieves strong local confluence, but at a cost: the introduction of many bureaucratic reduction steps. To the best of our knowledge, only the directed version has been implemented [11], with the introduction of the *half-combustion* strategy, which allows for a higher degree of parallelism.

The other way in which GoI has been used for implementation was by turning graphs into bideterministic automata, attaching an action to each edge. Actions act on *contexts* (which play the role of words), as given by the context semantics of [8]. The first GoI implementation [10] was in fact obtained in this way, for the **PCF** language: the Geometry of Interaction Machine compiles terms into assembly code of a generic register machine, which runs an automaton.

Our Approach. In this paper we apply the execution formula directly. Our approach resembles VR in that it is syntactic, however it uses $\mathcal{L}^*$ rather than the more complicated structure of VR, thus algebraic manipulation is kept simple. This simplicity is a result of the representation of terms using matrices of weights, following Girard's presentation of the GoI.

From a data-structures point of view, matrices are a convenient representation for graphs. We have studied elsewhere [12] sequential algorithms for calculating execution paths, derived from the execution formula and using the same matrix representation that will be used here.

We achieve concurrent execution by calculating segments of the path (assigned to different threads of computation) starting from different nodes of its graph, and allowing the threads to communicate so that the weight of a finished segment can be used to calculate the weight of another (longer) segment.

Each implementation in this paper will be presented as an abstract machine, an abstract rewriting system working on *machine configurations*. While strongly based on the theory, each machine addresses all the major implementation questions, including the choice of a model for inter-thread communication (shared-memory vs. message-passing) and synchronization issues. The machines are parameterized, allowing to impose different path reduction strategies.

Plan. In Sect. 2 we briefly review basic concepts of the geometry of interaction. Section 3 defines the computational tasks that will be distributed for parallel execution, and in Sect. 4 a shared-memory abstract machine is defined. This is optimized in Sect. 5 to eliminate the need for synchronization mechanisms. In Sect. 6 we study redundant path computations and propose an abstract machine that gets rid of redundancy. Section 7 defines a distributed-memory machine, and in Sect. 8 we conclude with some comments about implementing these ideas.

The long version of this paper contains proofs and examples of execution.

2 Background

Our treatment of the theory here is necessarily superficial; for a more thorough introduction to GoI (including VR and DVR), see [6, 5, 2, 3].

The Language. We will use a typed λ-calculus with a single base type. The syntax of our terms (ranged over by t, u, v) will be (with x, y, z variables, n an integer constant and S the successor function):

$$t ::= \mathsf{n} \mid \mathsf{S} \mid x \mid uv \mid \lambda x.u$$

The typing rules are the standard ones: if with $x : \sigma$ we have $M : \tau$ then $\lambda x.M : \sigma \to \tau$; if $M : \sigma \to \tau$ and $N : \sigma$ then $MN : \tau$. The constants $\mathsf{S} : \mathbf{nat} \to \mathbf{nat}$ and $\mathsf{n} : \mathbf{nat}$ have the expected types. The reduction rules we wish to implement are β-reduction and a δ-rule for the constants:

$$\begin{aligned} (\lambda x.t)u &\longrightarrow t[u/x] \\ \mathsf{S}\mathsf{n} &\longrightarrow \mathsf{n}+1 \end{aligned}$$

All the results in the paper can be extended to include conditionals and recursion; we choose to keep the language as simple as possible for the sake of clarity.

The Geometry of Interaction Dynamic Algebra $\mathcal{L}^*$. We define a single-sorted signature with constants $0, 1, p, q, r, s, t, d$; two unary operators $(\cdot)^*$ and $!(\cdot)$, and an infix (denoted by .) binary composition. The equational theory $\mathcal{L}^*$ is defined over this signature as follows (where variables x, y stand for arbitrary terms):

- The structure is monoidal, with identity 1 and composition as multiplicative operation, and 0 is an absorbing element for composition. Associativity allows to write $u.v$ as uv, and both $u.(v.w)$ and $(u.v).w$ as uvw.
- The *inversion operator* $(\cdot)^*$ is an involutive antimorphism for 0, 1, and composition:
$$0^* = 0 \qquad 1^* = 1$$
$$(x^*)^* = x \qquad (xy)^* = y^*x^*$$
- The *exponential operator* ! is a morphism for 0, 1, inversion, and composition:
$$!(0) = 0 \qquad !(1) = 1$$
$$!(x)^* = !(x^*) \qquad !(x)!(y) = !(xy)$$
- The constants verify the *annihilation* equations:
$$c^*c = 1 \quad \text{for } c = q, p, r, s, t, d$$
$$q^*p = p^*q = 0$$
$$r^*s = s^*r = 0$$
- The following *commutation equations* are verified:
$$!(x)r = r!(x) \qquad !(x)s = s!(x)$$
$$!(x)t = t!!(x) \qquad !(x)d = dx$$
- To accomodate our language, we extend this theory following [9] with constants n (for each natural number) and S, with equations:
$$\mathsf{n}\,\mathsf{S} = \mathsf{S}\,(\mathsf{n}+1) \qquad \mathsf{S}^*\mathsf{S} = 1$$

Each commutation equation has a dual form as a consequence of $(xy)^* = y^*x^*$, for instance, $d^*!(x) = xd^*$. A binary sum operator may also be included in this theory, which is commutative and associative, has 0 as identity, and composition distributes over it. We call $\mathcal{L}^*_+$ the theory $\mathcal{L}^*$ extended with this operator.

Execution Paths. The standard presentation consists in first defining the weight of paths in proof-nets. *Execution* paths are *regular* (i.e. their weight does not equal 0 in $\mathcal{L}^*$) and have as source and goal conclusions of the net. A translation of λ-terms into nets allows to lift the interpretation to the λ-calculus.

Decidability of Regularity. The term-rewriting system $\mathcal{R}_{\mathcal{L}^*}$ is obtained by orientating from left to right all the equations in $\mathcal{L}^*$ (including the dual commutation equations). $\mathcal{R}_{\mathcal{L}^*}$ is confluent [13]. A *stable form* is 1 or any term $a!(m)b^*$ in $\mathcal{L}^*$ where a and b are positive flat terms, i.e., they contain no applications of $(\cdot)^*$ or $!(\cdot)$, and m is stable. Stable forms are normal with respect to $\mathcal{R}_{\mathcal{L}^*}$. Every stable form is equal to some term AB^* with A and B positive but not necessarily flat.

Proposition 1 (AB^* property [13, 3]). *If γ is a straight path in some net then its weight $w(\gamma)$ can be rewritten either to a stable form or to 0.*

Let γ be a path with $w(\gamma)$ rewritable to a stable form ab^*. Since for a positive monomial x, $\mathcal{L}^* \vdash x^*x = 1$, then $\mathcal{L}^* \vdash a^*ab^*b = 1$, and $\mathcal{L}^* \vdash a^*w(\gamma)b = 1$. Thus $\mathcal{L}^* \nvdash w(\gamma) = 0$ (otherwise $\mathcal{L}^* \vdash 0 = 1$, contradicted by the existence of non-trivial models for $\mathcal{L}^*$). This gives a decidable process for checking regularity.

Matrix Presentation. The interpretation or *execution* of a net is the set of all tuples (s, d, φ) such that there is an execution path with source s, goal d, and weight φ, with s, d (the indexes of) two conclusions of the net. One way to represent this is as a matrix of weights indexed by the conclusions of the net.

Following Girard [6] we associate to a proof a pair of matrices $(\Pi^\bullet, \sigma)$, indexed by the *terminal ports* of the corresponding proof-net. These are either conclusions of symbols which are connected to cut links, or conclusions of the net. An *up-down* path is a path that starts upwards at a terminal port and ends downwards at a terminal port. An *elementary* path is an up-down path which doesn't cross any cut link. The matrix $\Pi^\bullet$ associated to a net contains all the (sums of the weights of) elementary paths in it, and σ contains information relative to the cuts in the net (it corresponds to an involutive permutation on the non-conclusion terminal ports). In the long version of the paper we show how to build these matrices directly from a λ-term.

Definition 1 (Execution Formula). *Let t be a term and $(\Pi^\bullet, \sigma)$ the matrices associated to it. The* execution *of t is defined as follows, where C is called the* central part *of the formula:*

$$\mathcal{E}x(t) = (1-\sigma^2)C(1-\sigma^2) \quad \textit{where} \quad C = \Pi^\bullet \sum_{k=0}^{\infty} (\sigma\Pi^\bullet)^k$$

In this paper we only interpret ground-type terms. In these conditions the execution formula expresses an invariant on computation: if $t \longrightarrow t'$, then $\mathcal{E}x(t) = \mathcal{E}x(t')$. The unique conclusion (or *root*) of the corresponding proof-net will by convention have the highest index in the matrices. $\mathcal{E}x(t)$ is a square matrix of dimension N containing 0 everywhere except $\mathcal{E}x(t)_{N,N}$, which is the weight of the execution path of t. The following result from [10] is the last ingredient required for using the execution formula as an evaluation device:

Proposition 2. *If t has ground type and reduces to a constant c, and ϕ is the weight of its execution path, then $\mathcal{L}^* \vdash \phi = c$.*

3 Basic Computation Tasks

As far as the design of an abstract machine is concerned, the first step is to choose a notion of basic task of computation. Then the operation of the machine simply manages the concurrent execution of these tasks by the available threads.

Let φ be a path ending at a terminal port connected to a cut link C. A basic task is the action of composing φ with all the paths consisting of the cut link C followed by (i) an elementary path or (ii) any other up-down path. An element $(\sigma\Pi^\bullet)_{i,j}$ is the weight of a path starting downwards at terminal port i, crossing a cut, and traversing an elementary path. If φ ends at port i, multiplying its weight by the row vector $(\sigma\Pi^\bullet)_i$ captures the first case above. By adding the weights of other up-down paths to a copy of $\sigma\Pi^\bullet$, the second case is also captured.

Auxiliary Functions. Let N be the number of terminal ports in a net $\mathcal{N}$. We consider defined (in the context of a configuration) a matrix of weights B of dimension N (initially containing a copy of $\sigma\Pi^\bullet$), and a predicate $\mathsf{storePred}$ on paths, represented as tuples (s, d, φ), with s and d the source and goal ports, and φ their weight. $\mathsf{storePred}$ identifies paths which will cease to be grown. Instead, their weight will be stored to be reused later. The path starting at root should never stop being grown, thus one imposes $\mathsf{storePred}(N, d, \varphi) = \mathsf{false}$.

The function $\mathcal{I}_{cs}$ takes as argument a path (s, d, φ) and returns a pair (l_1, l_2) of lists of paths. This pair is obtained by composing the weight φ with all the weights in the row indexed by d in matrix B, and including each of the resulting weights (s, m, τ) in l_1 or l_2 according to whether $\mathsf{storePred}(s, m, \tau)$ holds or not.

A related function $\mathcal{I}'_{cs}$ also takes a path (s, d, φ) and returns a pair (l_1, l_2) of lists, now obtained by composing every weight stored in the column indexed by s in matrix X (part of the current configuration) with φ, and splitting the resulting paths using $\mathsf{storePred}$. The use of $\mathcal{I}'_{cs}$ will become clear in section 6.

4 Shared-Everything Abstract Machines

In this section we define a first abstract machine (i.e., a notion of configurations and a reduction relation) corresponding to a shared-memory implementation.

Definition 2. *An* SE-configuration *is a tuple* $\langle B \mid S \mid C \mid [t_1, \ldots t_m] \rangle$ *where*

- B *is a matrix of weights of paths of dimension* N*, representing a net.*
- $S, C \in (\mathbb{N} \times \mathbb{N} \times \mathcal{L}^*)^*$ *are the* storage *and* composition task lists*, respectively.*
- *Each* thread t_k *is a* State*, a term built from the following signature:*

$$\begin{array}{ll} \mathsf{store} : \mathbb{N} \times \mathbb{N} \times \mathcal{L}^* \to \mathrm{State} & \mathsf{compose} : \mathbb{N} \times \mathbb{N} \times \mathcal{L}^* \to \mathrm{State} \\ \mathsf{delist} : \mathrm{State} & \mathsf{stop} : \mathcal{L}^* \to \mathrm{State} \\ \mathsf{enlist} : (\mathbb{N} \times \mathbb{N} \times \mathcal{L}^*)^* \times (\mathbb{N} \times \mathbb{N} \times \mathcal{L}^*)^* \to \mathrm{State} & \end{array}$$

We will use the following definitions in the context of a configuration: $N_D = \{n \in \mathbb{N} \mid n \leq N\}$ and $N_T = \{n \in \mathbb{N} \mid n \leq m\}$.

The abstract machine rules are given in Table 1, where we omit some unchanged components. Standard notation is used for lists. The auxiliary function $add(B, i, j, \alpha)$ gives the matrix obtained by adding the weight α to $B_{i,j}$. The rules define a reduction relation $\longrightarrow$ on single-threaded configurations.

Definition 3 (SE Reduction). *The* $\xrightarrow{se}$ *reduction relation on multi-threaded configurations is the smallest relation verifying:*

$$\frac{\langle B \mid S \mid C \mid [t_i] \rangle \longrightarrow \langle \widehat{B} \mid \widehat{S} \mid \widehat{C} \mid [\widehat{t_i}] \rangle}{\langle B \mid S \mid C \mid [t_1, \ldots t_i, \ldots t_m] \rangle \xrightarrow{se} \langle \widehat{B} \mid \widehat{S} \mid \widehat{C} \mid [t_1, \ldots \widehat{t_i}, \ldots t_m] \rangle}$$

Definition 4. *A tuple* (s, d, ϕ)*, where* $\phi \in \mathcal{L}^*_+$*, is said to* belong to the execution *of a term iff* ϕ *can be written as a sum* $\phi = \sum_n \phi_n$*, such that for each* ϕ_n *one has* $\phi_n + \alpha_n = (\Pi^\bullet(\sigma\Pi^\bullet)^{i_n})_{s,d}$ *for some* i_n *and some term* $\alpha_n \in \mathcal{L}^*_+$.

Table 1. Shared-everything (SE) abstract machine

0 $s = d = N$	Net Thread	B $\mathsf{compose}(s, d, \varphi)$	B $\mathsf{stop}(\varphi)$
I	Net Thread	B $\mathsf{store}(s, d, \varphi)$	$add(B, \sigma(s), d, \varphi)$ delist
II $s \neq N$ or $d \neq N$	Net Thread	B $\mathsf{compose}(s, d, \varphi)$	B $\mathsf{enlist}(\mathcal{I}_{cs}(s, d, \varphi))$
III	STasks Thread	S $\mathsf{enlist}((s, d, \varphi) : T_s, T_c)$	$(s, d, \varphi) : S$ $\mathsf{enlist}(T_s, T_c)$
IV	CTasks Thread	C $\mathsf{enlist}(\varepsilon, (s, d, \varphi) : T_c)$	$(s, d, \varphi) : C$ $\mathsf{enlist}(\varepsilon, T_c)$
V	Thread	$\mathsf{enlist}(\varepsilon, \varepsilon)$	delist
VI	STasks Thread	$(s, d, \varphi) : S$ delist	S $\mathsf{store}(s, d, \varphi)$
VII	CTasks STasks Thread	$(s, d, \varphi) : C$ ε delist	C ε $\mathsf{compose}(s, d, \varphi)$

The following propositions establish sufficient conditions for the correctness of the machine and for the execution path to be computed.

Proposition 3. *Let t be a term represented as $(\Pi^\bullet, \sigma)$, and $\Sigma_0 = \langle \sigma\Pi^\bullet \mid \varepsilon \mid C_0 \mid [\mathsf{delist} \ldots \mathsf{delist}] \rangle$ a configuration where C_0 contains only paths $(i, j, \Pi^\bullet_{i,j})$ taken from $\Pi^\bullet$, excluding repetitions. Let $\Sigma_0 \xrightarrow{se}_* \Sigma = \langle B \mid S \mid C \mid [t_1 \ldots t_m] \rangle$.*

1. *If t_k is some thread in Σ and $t_k = \mathsf{compose}(s, d, \varphi)$ or $t_k = \mathsf{store}(s, d, \varphi)$, then (s, d, φ) belongs to the execution of t.*
2. *If $t_k = \mathsf{enlist}(T_s, T_c)$ is some thread in Σ and $(s, d, \varphi) \in T_s$ or $(s, d, \varphi) \in T_c$, then (s, d, φ) belongs to the execution of t.*
3. *If $(s, d, \varphi) \in S$ or $(s, d, \varphi) \in C$, then (s, d, φ) belongs to the execution of t.*
4. *For all s, d, the tuple $(\sigma(s), d, B_{s,d})$ belongs to the execution of t.*

Proposition 4. *Consider an initial configuration in the conditions of Prop. 3, where additionally C_0 contains the paths $(N, j, \Pi^\bullet_{N,j})$ such that $\Pi^\bullet_{N,j} \neq 0$. Then the machine stops with a final configuration $\overline{\Sigma}$ containing a thread in the state $\mathsf{stop}(\varphi)$, where φ is the weight of the unique execution path of the term.*

Deterministic Execution. For initial configurations Σ_0 in the conditions of Prop. 4 with C_0 containing *only* the weight of the path in row N of $\Pi^\bullet$, execution of the abstract machine is deterministic and equivalent to the Geometry of Interaction Machine [10]. There is always a single path inside the machine, that results from growing the unique elementary path with source the root of the term, and given the ground-type of the term, the result of each step (and each invocation of $\mathcal{I}_{cs}$) must be a unique path, for which the storePred predicate always returns false, thus a new compose task will be generated with the new path as argument.

Concurrent Execution. The present machine is a formalization of a variant of the producer-consumers model for (shared-memory) parallel programming, where the consumer threads are also producers, running the following cycle: first dequeue a task from the shared queue, then process it (possibly enqueueing new tasks); restart. In the abstract machine there are two different task lists, with different priorities (store tasks have higher priority than compose tasks).

The concurrent behaviour of $\xrightarrow{se}_*$ comes from sequentiality with non-determinism. A single thread executes a machine rule at each step of the $\xrightarrow{se}$ reduction, allowing to capture synchronization when accessing shared data-structures. For instance, if two threads may execute at the same time rule I, there will be a (2 step) $\xrightarrow{se}_*$ reduction with the correct result (B is changed by the two threads).

If parallel computation is desired, one must add more paths to C_0 than those in row N of $\Pi^\bullet$. These paths will be concurrently grown into longer regular paths. When a path φ computed by thread t_1 reaches a port from which another path φ' has been grown by t_2, φ' will be used for extending φ. At this point, t_1 and t_2 communicate using matrix B: if s and d are the source and goal ports of φ', then the weight φ' will be added to the current weight in $B_{\sigma(s),d}$ and composition of φ with φ' will happen naturally since $\sigma(s)$ is the goal port of φ.

One could devise a strategy for virtual reduction mimicking the abstract machine – given a set of nodes, grow all the paths leaving from those nodes. In VR each composition results in a new edge immediately incorporated in the net, whereas the machine will only perform store operations (which will add paths to the net represented by matrix B) with selected paths (which then cease to be grown). To illustrate this point, consider the net in Fig. 1, where a path is to be grown starting from the source of the edge φ_0. Virtual reduction produces the net on the left, where φ_0 has been composed with γ_1 to give φ_1, which has then been composed with γ_2 and so on. The abstract machine grows the path φ_0 until storePred is verified, and only then does it store the result path φ.

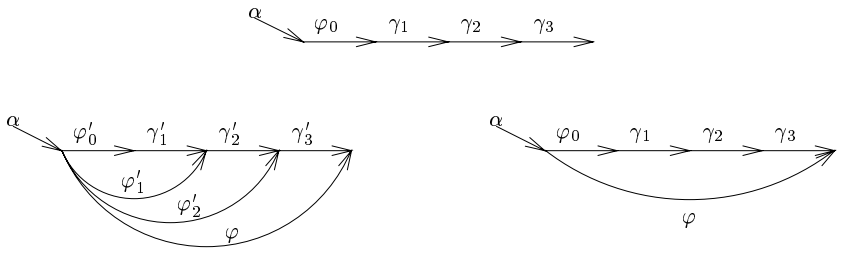

Fig. 1. Example path reductions

The machine is parameterized on which paths to start growing (included in C_0), and on the predicate storePred. One possible criterion is given in Sect. 6.

Implementation. The abstract machine may be implemented in any shared-memory architecture; threads in a configuration will be mapped into machine threads independently running the machine rules; threads will run in true parallelism, with synchronization introduced for accessing the shared data-structures. This ensures that the behaviour of the implementation corresponds to the interleaving reduction of the abstract machine. Any shared-memory library contains the appropriate synchronization devices, which we will here call *locks*.

Synchronization is required for accessing the shared task lists S and C, which may be read and written by any thread. This is done by associating a lock to each list: the C-lock must be acquired before and released after execution of rules IV and VII, and the same happens for the S-lock with rules III and VI.

Since the elements in a matrix are stored in independent memory positions, they can be protected individually, rather than treating B as a monolithic structure with a single lock. Synchronization is needed when two threads execute simultaneously rule I with the same s, d arguments: the individual lock associated to $B_{\sigma(s),d}$ must be acquired by a thread $\mathsf{store}(s, d, \varphi)$ executing rule I.

5 Distributed-Task-Lists Abstract Machines

Much synchronization is required for the parallel implementation of the SE-machine. We will eliminate this by including in threads private task lists.

Definition 5. *A* DTL-configuration *is a tuple* $\langle B \mid [t_1, \ldots t_m] \rangle$ *where*

- B *is a matrix of weights of paths of dimension* N*, representing a net.*
- *Each* t_k *is a* thread*,* $t_k = \langle S_k \mid C_k \mid st_k \rangle$*, where* $S_k, C_k \in (\mathbb{N} \times \mathbb{N} \times \mathcal{L}^*)^*$ *are the* storage *and* composition task lists *of the thread, respectively, and* st_k *its* state*, built from the same signature as before.*

Table 2 defines a reduction relation $\longrightarrow$ on single-threaded configurations (rule 0 will henceforth be considered implicit). We then define the following:

Definition 6 (DTL reduction). $\xrightarrow{dtl}$ *is the smallest relation verifying*

$$\frac{\langle B \mid [t_i] \rangle \longrightarrow \langle \widehat{B} \mid [\widehat{t_i}] \rangle}{\langle B \mid [t_1, \ldots t_i, \ldots t_m] \rangle \xrightarrow{dtl} \langle \widehat{B} \mid [t_1, \ldots \widehat{t_i}, \ldots t_m] \rangle}$$

Properties. Proposition 3 holds slightly modified, with initial configurations $\Sigma_0 = \langle \sigma\Pi^\bullet \mid [t_1^0, \ldots t_m^0] \rangle$ where each thread $t_k^0 = \langle \varepsilon \mid C_k^0 \mid \mathsf{delist} \rangle$, each C_k^0 contains only paths $(s, d, \Pi^\bullet_{s,d})$ from $\Pi^\bullet$, and the same path does not occur repeatedly in any two or in the same C_k^0. Also the third condition in the proposition is changed to "if t_k is some thread and $(s, d, \varphi) \in S_k$ or $(s, d, \varphi) \in C_k$, then (s, d, φ) belongs to the execution of t". Proposition 4 holds as well with small modifications: in the initial configurations, the paths $(N, j, \Pi^\bullet_{N,j})$ must be contained in some C_0^k. Finally another interesting property holds:

Table 2. Distributed-task-lists (DTL) abstract machine

I		Net	B	$add(B, \sigma(s), d, \varphi)$
	t_k	State	$\mathsf{store}(s, d, \varphi)$	delist
II		Net	B	B
$s \neq N$ or $d \neq N$	t_k	State	$\mathsf{compose}(s, d, \varphi)$	$\mathsf{enlist}(\mathcal{I}_{cs}(s, d, \varphi))$
III	t_k	STasks	S	$(s, d, \varphi) : S$
		State	$\mathsf{enlist}((s, d, \varphi) : T_s, T_c)$	$\mathsf{enlist}(T_s, T_c)$
IV	t_k	CTasks	C	$(s, d, \varphi) : C$
		State	$\mathsf{enlist}(\varepsilon, (s, d, \varphi) : T_c)$	$\mathsf{enlist}(\varepsilon, T_c)$
V	t_k	State	$\mathsf{enlist}(\varepsilon, \varepsilon)$	delist
VI	t_k	STasks	$(s, d, \varphi) : S$	S
		State	delist	$\mathsf{store}(s, d, \varphi)$
VII		CTasks	$(s, d, \varphi) : C$	C
	t_k	STasks	ε	ε
		State	delist	$\mathsf{compose}(s, d, \varphi)$

Proposition 5. *Let $i : N_D \to N_T$ be any map. Consider a configuration $\Sigma_0 = \langle B^0 \mid [t_1^0, \ldots t_m^0] \rangle$ with $t_k^0 = \langle \varepsilon \mid C_k^0 \mid \mathsf{delist} \rangle$, and C_k^0 containing only paths $(s, d, \Pi_{s,d}^{\bullet})$ such that $i(s) = k$. If $\Sigma_0 \xrightarrow{dtl} \Sigma$ and t_k is any thread in Σ with state $\mathsf{compose}(s, d, \varphi)$ or $\mathsf{store}(s, d, \varphi)$, then $i(s) = k$.*

Remarks. Suppose i is the identity function (and there are enough threads in the configuration). Then this proposition means that (with an appropriate initial configuration) thread t_k will handle all paths with source k, and only those. An immediate consequence is that when executing rule I, each thread writes to positions located in a unique row (indexed by $\sigma(k)$) in matrix B.

If not enough threads are available for all terminal ports, the function i will map terminal ports to threads. In this case each thread t_k will process paths with source ports from a distinct set, and will thus write to positions in different rows of B. t_k will however read from positions in any row of B.

Implementation. Each thread reads from and writes to its own task list only (so no synchronization is needed for accessing those lists). As to matrix B, no protection is needed, because of the previous remark. Thus this abstract machine can be implemented as a wait-free shared-memory program (no locks used).

6 Eliminating Redundancy

Consider again Fig. 1, and a path α ending where φ_0 starts. Two paths (φ_0 and φ) are available for composition with α. If α is composed with φ_0, the resulting path may continue being extended by composing with γ_1 and so on. These are redundant computations, since φ has already been computed. In terms of the

abstract machine, after the path $\varphi : s \to d$ has its weight stored in $B_{\sigma(s),d}$, the path α with goal the port $\sigma(s)$ may be composed not only with φ, but also with the elementary φ_0 from which φ was grown, whose weight still stands in the row indexed by $\sigma(s)$ in B. The current machine either follows *both* paths, performing many redundant computations, or, if some thread extends α before φ has been stored in B, it does not follow φ at all, which sequentializes execution.

To eliminate these redundancies, it is sufficient to remove from the net the path φ_0. In the abstract machine it can be removed from matrix B:

Definition 7. *A redundancy-free initial DTLW-configuration is any* $\Sigma_0 = \langle B^0 \mid [0]_N \mid [t_1^0, \ldots t_m^0] \rangle$, *all* $t_k^0 = \langle \varepsilon \mid C_k^0 \mid \mathsf{delist} \rangle$, *and*

$$B^0_{\sigma(s),d} = \begin{cases} 0 & \text{if } \Pi^\bullet_{s,d} \in C_k^0 \text{ for some } k, \\ (\sigma\Pi^\bullet)_{\sigma(s),d} & \text{otherwise} \end{cases}$$

There is a problem with this if φ has not yet been computed: whereas before the path α could continue being extended by composing with φ_0, now it will die, preventing computation of the execution path. This is solved by keeping account of all the paths candidates for composition with paths in B, and performing the corresponding compositions when new weights are stored in B. These "waiting paths" (such as α in the example) will be kept in a matrix X in configurations (DTL-configurations with this X component are called DTLW), and the function $\mathcal{I}'_{cs}$ will be used to perform the necessary compositions.

Proposition 5 cannot hold, since thread t_k will handle paths with arbitrary sources, generated by composing elements of X (of arbitrary source) with a path (s, d, φ) to be stored. Thus this machine cannot be implemented without synchronization. A change of perspective will allow to recover Prop. 5, at the expense of allowing threads to write to each other's task lists.

Table 3 contains the rules for the new abstract machine. Rules III and IV involve two threads. We will consider that the list of threads may be accessed as an *array*, a partial map from indexes to threads, $L : NT \hookrightarrow \text{State}$. When $i \notin dom(L)$, $L[i \mapsto t_i]$ denotes the union of L with the singleton $\{(i, t_i)\}$. We will still use list notation if convenient, and t_i will abbreviate $L(i)$.

$$\frac{\langle B \mid X \mid [t_i] \rangle \longrightarrow \langle \widehat{B} \mid \widehat{X} \mid [\widehat{t_i}] \rangle \qquad i \notin dom(L)}{\langle B \mid X \mid L[i \mapsto t_i] \rangle \overset{dtlmr}{\longrightarrow} \langle \widehat{B} \mid \widehat{X} \mid L[i \mapsto \widehat{t_i}] \rangle}$$

$$\frac{\langle B \mid X \mid [t_a, t_b] \rangle \longrightarrow_{III,IV} \langle \widehat{B} \mid \widehat{X} \mid [\widehat{t_a}, \widehat{t_b}] \rangle \qquad a, b \notin dom(L)}{\langle B \mid X \mid L[a \mapsto t_a, b \mapsto t_b] \rangle \overset{dtlmr}{\longrightarrow} \langle \widehat{B} \mid \widehat{X} \mid L[a \mapsto \widehat{t_a}, b \mapsto \widehat{t_b}] \rangle}$$

$\longrightarrow_{III,IV}$ denotes reduction using one of rules III or IV. A possible particular case for these two rules is that $i(s) = k$. For this reason the definition of $\overset{dtlmr}{\longrightarrow}$ includes the possibility of a single-thread reduction using rule III or IV.

Properties. Propositions 3 and 5 hold, with Σ_0 in the conditions of Def. 7 and $\overset{dtlmr}{\longrightarrow}$ replacing $\overset{dtl}{\longrightarrow}$. A consequence of Prop. 5 is that

Table 3. DTLW abstract machine with mutual writing

I		Net	B	$add(B, \sigma(s), d, \varphi)$
		WPaths	X	X
	t_k	State	$\mathsf{store}(s, d, \varphi)$	$\mathsf{enlist}(\mathcal{I}'_{cs}(\sigma(s), d, \varphi))$
II		Net	B	B
$s \neq N$ or $d \neq N$		WPaths	X	$add(X, s, d, \varphi)$
	t_k	State	$\mathsf{compose}(s, d, \varphi)$	$\mathsf{enlist}(\mathcal{I}_{cs}(s, d, \varphi))$
III	$t_{i(s)}$	STasks	S	$(s, d, \varphi) : S$
	t_k	State	$\mathsf{enlist}((s, d, \varphi) : T_s, T_c)$	$\mathsf{enlist}(T_s, T_c)$
IV	$t_{i(s)}$	CTasks	C	$(s, d, \varphi) : C$
	t_k	State	$\mathsf{enlist}(\varepsilon, (s, d, \varphi) : T_c)$	$\mathsf{enlist}(\varepsilon, T_c)$
V	t_k	State	$\mathsf{enlist}(\varepsilon, \varepsilon)$	delist
VI	t_k	STasks	$(s, d, \varphi) : S$	S
		State	delist	$\mathsf{store}(s, d, \varphi)$
VII		CTasks	$(s, d, \varphi) : C$	C
	t_k	STasks	ε	ε
		State	delist	$\mathsf{compose}(s, d, \varphi)$

- the element indexed by (s, d) in matrix B is only written by thread $t_{i(\sigma(s))}$ but can be read by any thread;
- the element indexed by (s, d) in matrix X is only written by thread $t_{i(s)}$ and only read by thread $t_{i(\sigma(d))}$ (when applying function $\mathcal{I}'_{cs}$).

Proposition 4 no longer holds for free: a judicious choice of initial configurations and definition of storePred are now necessary, guaranteeing that storePred is verified at some point for all the paths calculated concurrently. Notably, this means these paths should not overlap. Let α and β be two subpaths of the execution path such that β starts inside α. If when the port where β starts is reached, β has already been stored, then the part of α that has been computed will be composed with β, and the storePred predicate will never be applied to α.

We propose as an example the following criterion: consider a set P of terminal ports, and include in C_0 all the paths $\Pi^{\bullet}_{i,j}$ such that $i \in P$, and let

$$\mathsf{storePred}(s, d, \varphi) = (\sigma(d) \in P \textbf{ or } d = N) \textbf{ and } s \neq N$$

where the condition $d = N$ is necessary to store the last subpath. This guarantees that paths do not overlap since each path ends where another one starts.

Implementation. The access to matrices B and X is naturally protected – no two threads can write to the same position in B or X. Locks are required, for the individual lists of all threads, to be used as follows:

- for executing rule III, thread t_k must own the S-lock of thread $t_{i(s)}$;
- for executing rule IV, thread t_k must own the C-lock of thread $t_{i(s)}$;
- for executing rule VI, thread t_k must own its own S-lock;
- for executing rule VII, thread t_k must own its own C-lock.

Table 4. Distributed-everything abstract machine

I	t_k	Net	B	$add(B, \sigma(s), d, \varphi)$
		WPaths	X	X
		State	$\mathsf{store}(s, d, \varphi)$	$\mathsf{enlist}(\mathcal{I}'_{cs}(\sigma(s), d, \varphi))$
II	t_k	Net	B	B
$s \neq N$ or $d \neq N$		WPaths	X	$add(X, s, d, \varphi)$
		State	$\mathsf{compose}(s, d, \varphi)$	$\mathsf{enlist}(\mathcal{I}_{cs}(s, d, \varphi))$
III	$t_{i(\sigma(s))}$	STasks	S	$(s, d, \varphi) : S$
	t_k	State	$\mathsf{enlist}((s, d, \varphi) : T_s, T_c)$	$\mathsf{enlist}(T_s, T_c)$
IV	$t_{i(d)}$	CTasks	C	$(s, d, \varphi) : C$
	t_k	State	$\mathsf{enlist}(\varepsilon, (s, d, \varphi) : T_c)$	$\mathsf{enlist}(\varepsilon, T_c)$
V	t_k	State	$\mathsf{enlist}(\varepsilon, \varepsilon)$	delist
VI	t_k	STasks	$(s, d, \varphi) : S$	S
		State	delist	$\mathsf{store}(s, d, \varphi)$
VII	t_k	CTasks	$(s, d, \varphi) : C$	C
		STasks	ε	ε
		State	delist	$\mathsf{compose}(s, d, \varphi)$

7 Distributed-Everything Abstract Machine

In our final machine, threads keep individual copies of the B and X matrices.

Definition 8. *A* DE-configuration *is a list* $[t_1, \ldots t_m]$ *where each* $t_k = \langle B_k \mid X_k \mid S_k \mid C_k \mid st_k \rangle$ *is a* thread*, with* B_k *and* X_k *matrices of weights of dimension* N*;* S_k *and* C_k *the* storage *and* composition task lists *of* t_k*, and* st_k *its* state*.*

Table 4 defines a reduction $\longrightarrow$ on one- and two-thread configurations. Then:

Definition 9 (DE reduction). $\xrightarrow{de}$ *is the smallest relation verifying:*

$$\frac{[t_i] \longrightarrow [\widehat{t_i}] \qquad i \notin dom(L)}{L[i \mapsto t_i] \xrightarrow{de} L[i \mapsto \widehat{t_i}]} \qquad \frac{[t_a, t_b] \longrightarrow_{III,IV} [\widehat{t_a}, \widehat{t_b}] \qquad a, b \notin dom(L)}{L[a \mapsto t_a, b \mapsto t_b] \xrightarrow{de} L[a \mapsto \widehat{t_a}, b \mapsto \widehat{t_b}]}$$

Each thread now writes composition tasks to the task list of the thread corresponding to the *goal* port of the respective path.

Proposition 6. *Let* $i : N_D \to N_T$ *be a map, and consider a configuration* $\Sigma_0 = [t_1^0, \ldots t_m^0]$*, each* $t_k^0 = \langle B_k^0 \mid [0]_N \mid \varepsilon \mid C_k^0 \mid \mathsf{delist} \rangle$*, and* C_k^0 *containing only paths* $(s, d, \Pi^{\bullet}_{s,d})$ *with* $i(d) = k$*. If* $\Sigma_0 \xrightarrow{de} \Sigma$ *and* t_k *is any thread in* Σ *with state* $\mathsf{compose}(s, d, \varphi)$*, then* $i(d) = k$*; if* t_k *has state* $\mathsf{store}(s, d, \varphi)$*, then* $i(\sigma(s)) = k$*.*

Corollary 1. *With* Σ_0 *in the conditions of Prop. 6,* $B_{s,d}$ *is only read and written by thread* $t_{i(s)}$*, and* $X_{s,d}$ *is only read and written by thread* $t_{i(d)}$*.*

Each thread only needs to read from exactly the same positions of B and X that it writes to, thus the local copies of B and X do not need to be kept consistent.

Implementation. In an implementation of this machine, synchronization is only needed for accessing the task lists of individual threads, used for communication.

In practice it is not necessary that threads keep copies of the entire matrices: rows of B and columns of X can be distributed so that thread t_k keeps only the rows of B and columns of X indexed by d such that $i(d) = k$.

A Message-passing Machine. We now propose a change of perspective: consider that the task lists are *communication buffers*, where messages sent to a thread are kept before they are received by the thread. Then the enlist operation is a synchronous buffered *send* operation, which puts a task into the destination thread's buffer. delist is a *receive* operation, by which a thread removes a message from one of its buffers. Two types of messages (compose and store) may be sent to a thread, which will be stored in different buffers.

The message-passing mechanisms provided by any parallel-programming library ensure that messages are naturally ordered on arrival and placed sequentially in the corresponding buffer (thus replacing synchronization).

8 Conclusions and Further Work

The fact that the abstract machines allow to identify the necessary synchronization mechanisms is of great importance: an important product of this is the wait-free abstract machine of Sect. 5. Wait-free implementations are typically difficult to obtain (to understand the need for synchronization in VR the reader should think of a situation like $\cdot \longrightarrow \cdot \longrightarrow \cdot \longrightarrow \cdot$ where a critical pair exists).

The abstract machines are parameterized on the initial paths to be extended, as well as on the criterion to stop extending paths. This allows to implement different strategies for path computations (unlike virtual reduction, which, being a local reduction relation, has no built-in strategy). An instance of the abstract machines given here always incorporates a precise strategy, and this allows notably to eliminate synchronization as well as useless computations.

The parameterization we have given in Sect. 6 guarantees the correctness of the machines in sections 6 and 7, but does not allow for a subpath ϕ of the execution path to be used to extend another subpath ϕ': only the execution path can be extended using already computed subpaths. This has the advantage of simplicity, but it remains to study other efficient criteria.

The appropriate technologies exist for implementing the given machines in widely available architectures, both for shared-memory (for instance POSIX threads on SMP architectures) and distributed-memory (message-passing libraries such as MPI or PVM). It is worth mentioning that we have implemented the DE-machine using MPI, and started testing it over a local-area network. With respect to shared-memory implementations, it will be important to compare the wait-free (Sect. 5) and the redundancy-free (Sect. 6) machines.

References

1. Vincent Danos, Marco Pedicini, and Laurent Regnier. Directed virtual reductions. In M. Bezem and D. van Dalen, editors, *Computer Science Logic, 10th International Workshop, CSL '96*, volume 1258 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
2. Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of Girard's execution formula). In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 296–306. IEEE Computer Society Press, 1993.
3. Vincent Danos and Laurent Regnier. Proof-nets and the Hilbert space. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 307–328. 1995.
4. Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
5. Jean-Yves Girard. Geometry of interaction 2: Deadlock-free algorithms. In Per Martin-Löf and G. Mints, editors, *International Conference on Computer Logic, COLOG 88*, pages 76–93. Springer-Verlag, 1988. Lecture Notes in Computer Science 417.
6. Jean-Yves Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. North Holland Publishing Company, Amsterdam, 1989.
7. Jean-Yves Girard. Geometry of interaction III : accommodating the additives. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 329–389. 1995.
8. Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. Linear logic without boxes. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science (LICS'92)*, pages 223–234. IEEE Press, 1992.
9. Ian Mackie. *The Geometry of Implementation.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.
10. Ian Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208. ACM Press, January 1995.
11. M. Pedicini and F. Quaglia. A parallel implementation for optimal lambda-calculus reduction. In *Proceedings of the 2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000).* ACM press, 2000.
12. Jorge Sousa Pinto. *Parallel Implementation with Linear Logic (Applications of Interaction Nets and of the Geometry of Interaction).* PhD thesis, École Polytechnique, 2001.
13. Laurent Regnier. *Lambda-Calcul et Réseaux.* PhD thesis, Université Paris VII, January 1992.