PELCR: PARALLEL ENVIRONMENT FOR OPTIMAL LAMBDA-CALCULUS REDUCTION

MARCO PEDICINI
ISTITUTO PER LE APPLICAZIONI DEL CALCOLO "M. PICONE" - CNR
AND
FRANCESCO QUAGLIA
UNIVERSITÀ DI ROMA "LA SAPIENZA"

Abstract. In this article we present the implementation of an environment supporting Lévy's *optimal reduction* for the λ-calculus [Lév78] on parallel (or distributed) computing systems. In a similar approach to Lamping's one in [Lam90], we base our work on a graph reduction technique known as *directed virtual reduction* [DPR97] which is actually a restriction of Danos-Regnier virtual reduction [DR93].

The environment, which we refer to as PELCR (Parallel Environment for optimal Lambda-Calculus Reduction) relies on a strategy for directed virtual reduction, namely *half combustion*. While developing PELCR we have adopted both a message aggregation technique, allowing a reduction of the communication overhead, and a fair policy for distributing dynamically originated load among processors. Additionally, we have used a set of other optimizations, e.g. allowing the maintenance of relatively low size for the manipulated data structures so not to incur problems related to their management at the application level or due to the management of large process memory images at the operating system level.

We also present an experimental study demonstrating the ability of PELCR to definitely exploit parallelism intrinsic to λ-terms while performing the reduction. We show how PELCR allows achieving up to 70/80% of the ideal speedup on last generation multiprocessor computing systems. As a last note, the software modules have been developed with the `C` language and using a standard interface for message passing, i.e. MPI, thus making PELCR itself a highly portable software package.

1. Introduction

Jean-Jacques Lévy formally characterized the meaning of the word *optimal* relatively to a reduction strategy for λ-calculus, referring to it as the property that the strategy reaches the normal form (if it exists) and does not duplicate the work of reducing similar β-redexes [Lév78].

Key words and phrases. Functional Programming, Optimal Reduction, Linear Logic, Geometry of Interaction, Virtual Reduction, Parallel Implementation.

[0]An earlier version of this article by the same authors, with title "A Parallel Implementation for Optimal Lambda-Calculus Reduction" appeared in *Proc. of the 2nd ACM SIGPLAN Int. Conference on Principles and Practice of Declarative Programming (PPDP 2000)* .

Authors' addresses: M. Pedicini, Istituto per le Applicazioni del Calcolo "M. Picone", Consiglio Nazionale delle Ricerche, Viale del Policlinico 137, 00161 Roma, Italy, email `marco@iac.cnr.it`; F. Quaglia, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy, email `quaglia@dis.uniroma1.it`.

This characterization was formalized in terms of *families of redexes* that is, redexes with the same origin, possibly this origin being a virtual one in the sense that two families coming in a configuration producing a new redex originate a new family. Redexes belonging to different families cannot be successfully shared during reduction; whereas for two redexes in the same family, one could find an optimal strategy (i.e. reducing all of them in a single step).

Data structures suitable for an implementation of optimal reduction were presented a long time later [Lam90]; the outcome reduction technique introduced by J. Lamping, known as *sharing reduction*, relies on a set of graph rewriting rules.

In [GAL92], Lamping's sharing reduction was proved to be a way to compute Girard's execution formula, which is an invariant of closed functional evaluation obtained from the "Geometry of Interaction" interpretation of λ-calculus [Gir89]. This result stirred the research in the field of optimal reduction. Specifically, in [DR93] a graphical local calculus, namely *virtual reduction* (VR), was defined as a mechanism to perform optimal reduction by computing the Girard's execution formula. Such a calculus was later refined in [DPR97], by the introduction of a new graph rewriting technique known as *directed virtual reduction* (DVR). The authors also defined a strategy to perform DVR, namely *combustion*, which simplifies the calculus and can simulate individual steps of sharing reduction.

In this article we describe a technique for the implementation of functional calculi. This technique exploits both *locality and asynchrony* of the computation which is typical in interaction nets ([Laf90]) and derives from the fine decomposition of the λ-calculus β-rule obtained through the analysis provided by the Geometry of Interaction. Specifically, we present the implementation of a Parallel Environment for optimal Lambda-Calculus Reduction (PELCR) which relies on DVR and on a new strategy to perform DVR that will be referred to as *half-combustion* (HC).

Let us stress that any interpreter of an ML-like functional language based on our technique ensures the execution of programs in a parallel (or distributed) environment in a way completely transparent to the user.

To the best of our knowledge, our work is the first attempt for parallel implementations of optimal λ-calculus reduction. Actually in [Mac97] issues on the possibility of parallel implementations for Lafont's interaction nets are discussed. In that work Mackie is faced to problems of load balancing and fine grain parallelism. The solution proposed in [Mac97] is a *static analysis* of the initial interaction net which aims at setting up a favorable initial distribution of the nodes among processors. His work is related to optimal reduction since optimal rules (e.g. in [GAL92]) define an interaction system [Laf90]. However, contrarily to our work, it does not focus on optimal reduction. Another fundamental difference between our work and Mackie's study is that our approach is dynamic: load distribution is decided at run-time and the message passing overhead is controlled dynamically as well. Also, our implementation embeds a set of other optimizations further allowing improved run-time behavior, e.g. for what concerns memory performance at both the application level and the operating system level.

The implementation has been developed with the `C` language using a standard interface, namely MPI, for supporting message passing functionalities among processes involved in the computation. These peculiarities make PELCR a highly portable software package, easy to install on a wide set of, possibly heterogeneous, computing platforms.

We also report the results of an experimental evaluation of our software package. By the experimental data, we show how it has the ability to definitely exploit any form of parallelism intrinsic to the reduction of λ-terms. As a result, we obtain up to 70/80% of the ideal speedup, i.e. the ideal acceleration as compared to a sequential case, while performing λ-term reductions on last generation multiprocessor systems. This, in its turn, also allows decreasing the wall-clock time for the reduction from several tens of seconds to few seconds. This points out how PELCR has the potential to cope with response time requirements for the satisfaction of an interactive end-user even in case of jobs that would require large computation time if executed in a classical sequential fashion.

We analyze the problem (parallel implementation of functional calculi) from a pragmatic point of view, and the theory of (directed) virtual reduction is here considered mainly for how it can give rise to parallel dynamics. However, while recalling such a theory, we also propose a few optimization rules allowing an increase in the effectiveness of the DVR approach (see Section 2.3), which have been taken into account while developing the implementation. The remainder of the article is structured as follows. In Section 2 we recall DVR. In Section 3 the HC strategy for DVR is introduced. In Section 4 we report the description of our implementation. The experimental results are reported in Section 5.

Acknowledgments. The project of a parallel and optimal interpreter for λ-calculus started as a joint effort between the University of Paris 7 and the "Istituto per le Applicazioni del Calcolo" in Rome (see "Optimal and parallel evaluations in functional languages" CNR/CNRS - Bilateral Project n.3132 - 1996/97); some of the ideas used in our implementation arose thanks to discussions of the first author with Vincent Danos. The authors also wish to thank Carlo Giuffrida for his support while developing some software modules.

2. From Lambda-Terms to Directed Virtual Reduction

As pointed out in the introduction we deal with an evaluator for λ-terms based on DVR, to be executed on parallel/distributed computing system. The pioneering ideas contained in Lévy's work on optimal reduction where finally realized by Lamping and then related to semantical questions about operational aspects of computations. In fact almost at the same time Girard gave the foundations of an outstanding mathematical base for the study of operational semantics.

Just to enumerate them we should cite Lamping's first work on sharing reduction, the connection with Geometry of Interaction discovered by Gonthier and finally the work of Danos-Regnier on VR and DVR. There is no way to get a complete and self-contained presentation of all this material, therefore, for a complete survey about the optimal implementation of functional programming languages we refer the reader to [AG98].

Here we shortly recall VR and the Geometry of Interaction (Section 2.1); then we will give a full presentation of DVR (Section 2.2), with the introduction of some properties (Section 2.3) which will be taken into account in the implementation. To ease the comprehension of this reduction technique and to make a more direct connection with Lamping's graphs we finally present an encoding of such graphs into directed virtual nets (Section 2.4).

The basic ingredient in Gonthier and Danos-Regnier works is the use of the invariance of the execution formula as a consistency criterion for the reduction

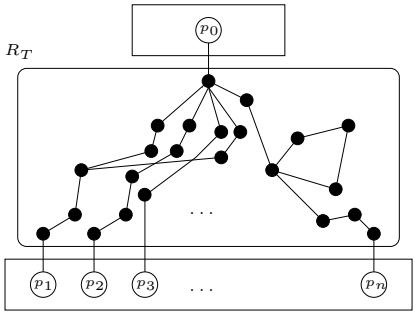


FIGURE 1. An explicative picture for execution paths.

technique. The execution formula associated with a term T with free variables $\{x_1, \ldots, x_n\}$ is given by the set of its border-to-border weighted straight paths in its dynamic graph R_T. Any node p_i in the border node is either associated with one free variable x_i or, if $i = 0$, p_0 represents the root of the term, see for example Figure 1. A straight path in a directed graph is a path that never bounces back in the same edge.

The execution formula of R_T is:

$$\mathsf{EX}(R_T) = \sum_{\phi_{ij} \in \mathcal{P}(R_T)} W(\phi_{ij})$$

where $W(.)$ is a morphism from the involutive category of paths $\mathcal{P}(R_T)$ to the monoid of the Geometry of Interaction, so that for any straight path ϕ_{ij} from p_i to p_j, $W(\phi_{ij})$ is an element of that monoid.

The preliminary step for our work is the Danos and Regnier's construction of a confluent, local and asynchronous reduction of λ-calculus, derived from a semantic setting based on a unique type of move (simple enough to be easily mechanized). Their graph reduction technique, namely VR, can be explained also as an efficient way to compute the execution formula. The one and only reduction rule is the composition of two edges in the graph as described in Figure 2. Whenever two edges of the virtual net are composable (i.e. the product of their weights is non-null), VR derives from them a new edge. The original edges are then marked by the *rest* of the composition, here denoted by weight within brackets.

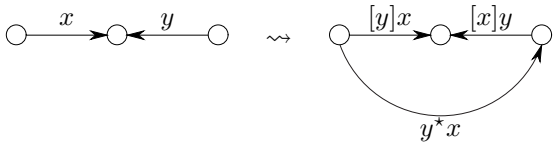


FIGURE 2. Composition performed by VR.

The algebraic mechanism corresponding to the rest is called the *bar*; it was introduced in [DR93] to ensure the preservation of Girard's execution formula. Note that VR induces bars of bars by definition; this is shown in Figure 3. DVR, presented in [DPR97], was designed in order to avoid bars of bars, thus allowing any implementation to use simple data structures for representing edges.

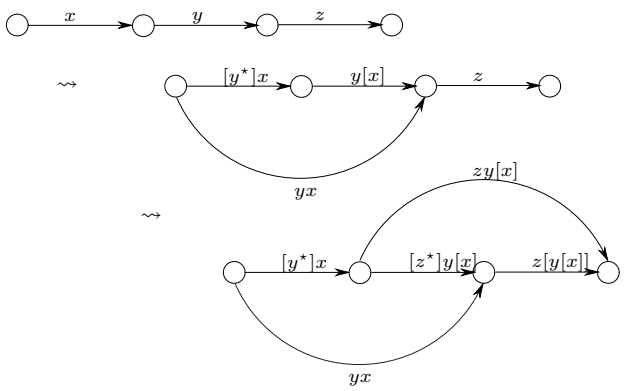


FIGURE 3. VR originating bars of bars.

2.1. **Geometry of Interaction.** The basic geometrical construction consists of a directed graph with weights in the dynamic algebra. The most important point is that the computation of Girard's "Execution Formula" is performed in a way that appears to be the natural candidate for parallel computation. In order to get a computational device from this graphical calculus, a suitable strategy has been introduced: by means of the *combustion* strategy it was proved that not only the mechanism of DVR computes the execution formula but also that it can do it in the same way as Lamping's algorithm for sharing graphs.

The Geometry of Interaction basic step is the introduction of a suitable algebraic structure, in view of the modeling of the dynamics of the reduction. This structure can be thought as of the set of partial one-to-one maps u with composition. The structure is then enriched with partial inverses $u^\star$, the codomain operation $\langle u\rangle$, and the complementary of the codomain $[u]$. Axioms for such a structure are formally introduced below.

Definition 2.1. *An* inverse monoid *(see* [Pet84]*), or for short an* im*, is a monoid with an unary function, the* star*, denoted by* $(.)^\star$*, with*

$$(uv)^\star = v^\star u^\star, \tag{1}$$

$$(u^\star)^\star = u, \tag{2}$$

$$uu^\star u = u, \tag{3}$$

$$uu^\star vv^\star = vv^\star uu^\star. \tag{4}$$

We denote by $\langle u\rangle$ the idempotent $uu^\star$. With this notation the last equation becomes $\langle u\rangle\langle v\rangle = \langle v\rangle\langle u\rangle$ and the one before becomes $\langle u\rangle u = u$.

Definition 2.2. *A* bar inverse monoid*, or for short a* bim*, is an im with a zero, denoted by 0, and an unary function, the* bar*, denoted by* $[.]$*, with*

$$[1] = 0 \quad \textit{and} \quad [0] = 1, \tag{5}$$

$$u[v] = [uv]u. \tag{6}$$

Bim's axioms entail:

(1) $[u]u = 0$, in fact $u[1] = [u\,1]u$ thus $0 = [u]u$;
(2) $[u][u] = [u]$, in fact $[u][u] = [[u]u][u] = [0][u] = [u]$;

(3) $[u]^\star = [u]$, $[u]^\star[u] = [u]^\star[u][u]^\star[u] = [u]^\star[u][u]^\star[u]^\star[u] = [u]^\star[u][u][u]^\star[u]^\star[u] = [u]^\star[u][u][u]^\star[u]^\star[u][u]^\star = [u]^\star[u][u][u]^\star = [u]^\star[u][u]^\star[u][u][u]^\star = [u]^\star[u][u][u]^\star = [u]^\star[u][u]^\star = [u]^\star$ then we have $[u] = [u][u]^\star[u] = [u][u]^\star[u][u]^\star[u] = [u][u]^\star[u]^\star[u] = [u]^\star[u][u][u]^\star = [u]^\star[u][u]^\star = [u]^\star$;

(4) and $vu = 0$ iff $v[u] = v$. In fact $v[u] = [vu]v = [0]v = v$, on the other hand if $v[u] = v$ then $v[u]u = vu$ but $[u]u = 0$ thus $v[u]u = 0 = vu$.

Now we give the construction of the free bim generated by a given im. So let S be an im, and $\mathbf{Z}[S]$ denote the free contracted algebra over S with coefficients in $\mathbf{Z}$ (the ring of integers). In other words $\mathbf{Z}[S]$ is the algebra of maps from S to $\mathbf{Z}$ with finitely many non-zero values. In other words $\mathbf{Z}[S]$ is the algebra of linear combinations over S with coefficients in $\mathbf{Z}$.

For any such linear combination, $s = \sum n_i s_i$, define

$$s^\star = \sum n_i {s_i}^\star, \qquad [s] = 1 - \langle s \rangle = 1 - ss^\star. \tag{7}$$

Define the *complementary closure* of S in $\mathbf{Z}[S]$, denoted by $[S]$, as the monoid generated in $\mathbf{Z}[S]$ by the union of S and $\{1 - \langle u \rangle,\ u \in S\}$.

Proposition 2.3. *$[S]$ is an inverse monoid with $(.)^\star$ defined as in (7).*

Proof. The proof is a straightforward calculation. In fact we have that for every element $s \in [S]$ can be written as a combination

$$s = s_1[u_1]s_2 \dots [u_n]s_{n+1}. \tag{8}$$

Let us introduce the length $|s|$ of an element $s \in [S]$ as the smallest n such that (8).

Now we prove that properties (1)-(4) in Definition 2.1 hold for $[S]$.

(1) First we prove that for every $u, v \in [S]$ we have $(uv)^\star = v^\star u^\star$, by double induction on the lengths of u and v.

If $|u| = 0$ and $|v| = 0$ then $u, v \in S$ and the property holds by definition, because S is an inverse monoid.

Let be $|u| = 0$ and for every v such that $|v| \leq n$ the property holds, then we prove that for every v' such that $|v'| = n + 1$, $(uv')^\star = {v'}^\star u^\star$.

Let be $v' = v[u_{n+1}]s_{n+2}$ and $u = s$, then

$$\begin{aligned}
(uv')^\star &= (sv[u_{n+1}]s_{n+2})^\star = \\
&= (sv(1 - \langle u_{n+1} \rangle)s_{n+2})^\star = \\
&= (svs_{n+2} - sv\langle u_{n+1} \rangle s_{n+2})^\star = \\
&= (svs_{n+2})^\star - (sv\langle u_{n+1} \rangle s_{n+2})^\star = \\
&= {s_{n+2}}^\star v^\star s^\star - {s_{n+2}}^\star \langle u_{n+1} \rangle v^\star s^\star = \\
&= {s_{n+2}}^\star [u_{n+1}] v^\star s^\star = {v'}^\star u^\star,
\end{aligned}$$

in fact $|v's_{n+2}| = n$ and from $\langle u_{n+1} \rangle s_{n+2} \in S$ we have $|v'\langle u_{n+1} \rangle s_{n+2}| = n$; thus by induction hypothesis $(svs_{n+2})^\star = {s_{n+2}}^\star v^\star s^\star$ and $(sv'\langle u_{n+1} \rangle s_{n+2})^\star = {s_{n+2}}^\star \langle u_{n+1} \rangle {v'}^\star s^\star$. Now suppose the property holds for any u such that $|u| \leq n$ and for every $v \in [S]$. We show that it holds for $u' \in [S]$ such that

$|u'| = n + 1$.

$$\begin{aligned}(u'v)^\star &= (u[u_{n+1}]s_{n+2}v)^\star = \\ &= (us_{n+2}v - uu_{n+1}{u_{n+1}}^\star s_{n+2}v)^\star = \\ &= (us_{n+2}v)^\star - (uu_{n+1}{u_{n+1}}^\star s_{n+2}v)^\star,\end{aligned}$$

we have $|us_{n+2}| = |u| = n$ and $|uu_{n+1}{u_{n+1}}^\star s_{n+2}| = n$ by induction hypothesis, we have $(us_{n+2}v)^\star = v^\star {s_{n+2}}^\star u^\star$ and

$$(uu_{n+1}{u_{n+1}}^\star s_{n+2}v)^\star = v^\star {s_{n+2}}^\star u_{n+1}{u_{n+1}}^\star u^\star,$$

thus

$$\begin{aligned}(u'v)^\star &= v^\star {s_{n+2}}^\star u^\star - v^\star {s_{n+2}}^\star u_{n+1}{u_{n+1}}^\star u^\star = \\ &= v^\star {s_{n+2}}^\star [u_{n+1}]u^\star = \\ &= v^\star (u[u_{n+1}]s_{n+2})^\star = v^\star u'^\star.\end{aligned}$$

(2) Now we prove that $u^{\star\star} = u$. Again by induction on the length $|u|$. It is clear that if $|u| = 0$ then $u \in S$ and $u^{\star\star} = u$ by definition of inverse monoid.

Suppose that the for every $u \in [S]$ such that $|u| \leq n$ we have $u^{\star\star} = u$, and consider u' such that $|u'| = n+1$. We may write $u' = u[u_{n+1}]s_{n+2}$, then we have $u'^\star = (u[u_{n+1}]s_{n+2})^\star$ by the previous proof we have $(u[u_{n+1}]s_{n+2})^\star = {s_{n+2}}^\star [u_{n+1}]u^\star$ and

$$\begin{aligned}u'^{\star\star} =&(s_{n+2}{}^\star [u_{n+1}]u^\star)^\star = \\ =&u^{\star\star}[u_{n+1}]{s_{n+2}}^{\star\star} = \\ =&u[u_{n+1}]s_{n+2},\end{aligned}$$

since by induction hypothesis $u^{\star\star} = u$.

(3) Let us prove that for every $u \in [S]$ we have $uu^\star u = u$. Case $|u| = 0$, implies $u \in S$ and follows from the definition of S.

Suppose, $uu^\star u = u$ for every $u \in [S]$ such that $|u| \leq n$. Consider $u' = u[u_{n+1}]s_{n+2}$ such that $|u| = n$ and $|u'| = n + 1$. Then

$$\begin{aligned}u'u'^\star u' &= u[u_{n+1}]s_{n+2}(u[u_{n+1}]s_{n+2})^\star u[u_{n+1}]s_{n+2} = \\ &= u[u_{n+1}]s_{n+2}{s_{n+2}}^\star [u_{n+1}]u^\star u[u_{n+1}]s_{n+2} = \\ &= u(1 - \langle u_{n+1}\rangle)s_{n+2}{s_{n+2}}^\star (1 - \langle u_{n+1}\rangle)u^\star u(1 - \langle u_{n+1}\rangle)s_{n+2} = \\ &= us_{n+2}{s_{n+2}}^\star u^\star us_{n+2} \\ &\quad - u\langle u_{n+1}\rangle s_{n+2}{s_{n+2}}^\star u^\star us_{n+2} \\ &\quad - us_{n+2}{s_{n+2}}^\star \langle u_{n+1}\rangle u^\star us_{n+2} - us_{n+2}{s_{n+2}}^\star u^\star u\langle u_{n+1}\rangle s_{n+2} \\ &\quad + u\langle u_{n+1}\rangle s_{n+2}{s_{n+2}}^\star \langle u_{n+1}\rangle u^\star us_{n+2} \\ &\quad + u\langle u_{n+1}\rangle s_{n+2}{s_{n+2}}^\star u^\star u\langle u_{n+1}\rangle s_{n+2} \\ &\quad + us_{n+2}{s_{n+2}}^\star \langle u_{n+1}\rangle u^\star u\langle u_{n+1}\rangle s_{n+2} \\ &\quad - u\langle u_{n+1}\rangle s_{n+2}{s_{n+2}}^\star \langle u_{n+1}\rangle u^\star u\langle u_{n+1}\rangle s_{n+2} =\end{aligned}$$

$$
\begin{aligned}
&= us_{n+2} - us_{n+2}s_{n+2}{}^\star u^\star us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle s_{n+2}\\
&\quad - us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star us_{n+2}\\
&\quad - us_{n+2}s_{n+2}{}^\star u^\star u\langle u_{n+1}\rangle s_{n+2}\\
&\quad + u\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star us_{n+2}\\
&\quad + u\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star u^\star u\langle u_{n+1}\rangle s_{n+2}\\
&\quad + us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star u\langle u_{n+1}\rangle s_{n+2}\\
&\quad - u\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star u\langle u_{n+1}\rangle s_{n+2} =\\
&= us_{n+2} - us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle s_{n+2}\\
&\quad - us_{n+2}s_{n+2}{}^\star u^\star us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle s_{n+2}\\
&\quad - us_{n+2}s_{n+2}{}^\star u^\star u\langle u_{n+1}\rangle s_{n+2}\\
&\quad + us_{n+2}s_{n+2}{}^\star u^\star us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle s_{n+2}\\
&\quad + us_{n+2}s_{n+2}{}^\star u^\star us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle\langle u_{n+1}\rangle s_{n+2}\\
&\quad + us_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star u\langle u_{n+1}\rangle s_{n+2}\\
&\quad - u\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star u\langle u_{n+1}\rangle s_{n+2} =\\
&= us_{n+2} - 3u\langle u_{n+1}\rangle s_{n+2} + 3u\langle u_{n+1}\rangle s_{n+2} - u\langle u_{n+1}\rangle s_{n+2} =\\
&= us_{n+2} - u\langle u_{n+1}\rangle s_{n+2} =\\
&= u[u_{n+1}]s_{n+2} =\\
&= u',
\end{aligned}
$$

we used $us_{n+2}s_{n+2}{}^\star u^\star us_{n+2} = us_{n+2}$ by applying the induction hypothesis to u.

(4) We prove that for any $u, v \in [S]$, $uu^\star vv^\star = vv^\star uu^\star$. Let us fix $|u| = 0$ and let us show the property for any v by induction on $|v|$.

So $|u| = 0$ and if $|v| = 0$ then $u, v \in S$ and there is nothing to prove.

Let us apply induction hypothesis, and for a fixed n suppose the property holds for any $|u| = 0$ and v such that $|v| \leq n$. We then prove the property holds for any v' such that $|v'| = n + 1$, in this case we may suppose that $v' = v[u_{n+1}]s_{n+2}$, moreover by definition of $[u_{n+1}] = 1 - u_{n+1}u_{n+1}{}^\star$ and by distribution,

$$
\begin{aligned}
ss^\star v[u_{n+1}]s_{n+2}s_{n+2}{}^\star[u_{n+1}]v^\star &= ss^\star v[u_{n+1}][u_{n+1}]s_{n+2}s_{n+2}{}^\star v^\star =\\
&= ss^\star v[u_{n+1}]s_{n+2}s_{n+2}{}^\star v^\star =\\
&= ss^\star v(1 - \langle u_{n+1}\rangle)s_{n+2}s_{n+2}{}^\star v^\star =\\
&= ss^\star vs_{n+2}s_{n+2}{}^\star v^\star - ss^\star v\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star v^\star.
\end{aligned}
$$

In the last expression we have $|vs_{n+2}| = n$, thus from the basis of the induction

$$ss^\star vs_{n+2}s_{n+2}v^\star = vs_{n+2}s_{n+2}v^\star ss^\star.$$

In a similar way $\langle u_{n+1}\rangle, s_{n+2} \in S$ and so

$$
\begin{aligned}
v\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star v^\star &= v\langle u_{n+1}\rangle\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star v^\star =\\
&= v\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle v^\star,
\end{aligned}
$$

and $|v\langle u_{n+1}\rangle s_{n+2}| = n$, thus

$$ss^\star v\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star v^\star = v\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle v^\star ss^\star$$

by induction hypothesis.

Now we complete the proof by an induction on $|u|$, suppose that the property holds for any u such that $|u| \leq n$ and for any $v \in [S]$ and consider $u' = u[u_{n+1}]s_{n+2}$, then

$$\begin{aligned} u[u_{n+1}]s_{n+2}s_{n+2}{}^\star[u_{n+1}]u^\star vv^\star = \\ &= u[u_{n+1}][u_{n+1}]s_{n+2}s_{n+2}{}^\star u^\star vv^\star = \\ &= u[u_{n+1}]s_{n+2}s_{n+2}{}^\star u^\star vv^\star = \\ &= us_{n+2}s_{n+2}{}^\star u^\star vv^\star - u\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star u^\star vv^\star = \\ &= us_{n+2}s_{n+2}{}^\star u^\star vv^\star - u\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star vv^\star, \end{aligned}$$

since $[u_{n+1}]s_{n+2} \in S$ we have $|u[u_{n+1}]s_{n+2}| = n$ and by induction hypothesis we obtain

$$\begin{aligned} vv^\star us_{n+2}s_{n+2}{}^\star u^\star - vv^\star u\langle u_{n+1}\rangle s_{n+2}s_{n+2}{}^\star\langle u_{n+1}\rangle u^\star = \\ = vv^\star u[u_{n+1}]s_{n+2}s_{n+2}{}^\star[u_{n+1}]u^\star. \end{aligned}$$

□

Definition 2.4. *Define the* bar closure *of S, denoted by $[S]_\omega$, to be the im obtained by ω iterations of the complementary closure, that is: let $S_0 = S$ and $S_{n+1} = [S_n]$ then $[S]_\omega = \bigcup_{n\geq 0} S_n$.*

Proposition 2.5. *$[S]_\omega$ is a bar inverse monoid with $[.]$ defined as above.*

Proof. For every $u, v \in [S]_\omega$ there exists n s.t. $u, v \in [S]_n$ so $u\,v$, $u^\star$ and $[u]$ belong to $[S]_n$ and so to $[S]_\omega$; for the same reason bim's axioms are satisfied. □

Definition 2.6. *The monoid $\mathsf{L}^\star$ of the Geometry of Interaction is the free monoid with a morphism $!(.)$, an involution $(.)^\star$ and a zero, generated by p, q, and a family $W = (w_i)_i$ of exponential generators such that for any $u \in \mathsf{L}^\star$:*

$$x^\star y = \delta_{xy} \qquad \text{for } x, y = p, q, w_i, \tag{9}$$

$$!(u)w_i = w_i!^{e_i}(u), \tag{10}$$

where e_i is an integer associated with w_i called the lift *of w_i, i is called the* name *of w_i and we will often write $w_{i,e(i)}$ to explicitly note the lift of the generator.*

Equations (9) will be called of annihilation *and (10) are called equations of* swapping.

Orienting the equations (9-10) from left to right, one gets a rewriting system which is terminating and confluent. The non-zero normal forms, known as *stable forms*, are the terms $ab^\star$ where a and b are *positive* (i.e. written without *s). The fact that all non-zero terms are equal to such an $ab^\star$ form is referred to as the "$ab^\star$ property". From this, one easily gets that the word problem is decidable and that $\mathsf{L}^\star$ is an inverse monoid.

Every computation, from now on, will take place in the bar closure of $\mathsf{L}^\star$ in $\mathbf{Z}[\mathsf{L}^\star]$, which we denote by $[\mathsf{L}^\star]_\omega$. Since, as said, this is a bim, results in [DR93], which were stated and proved for any bar inverse monoid, apply with no further ado. Note that equalities in $[\mathsf{L}^\star]_\omega$ and in $\mathbf{Z}[\mathsf{L}^\star]$ are also decidable by rewriting to stable form.

Set $[b_1, \ldots, b_n] = 1 - b_1b_1{}^\star - \cdots - b_nb_n{}^\star$; $[b_1, \ldots, b_n]$ is an idempotent iff the b_i's are *orthogonal* that is, $\langle b_i\rangle\langle b_j\rangle = 0$.

Lemma 2.7 ((superposition))**.** *Let a, b and c be positive monomials in $\mathsf{L}^\star$ such that $\langle a\rangle\langle b\rangle$, $\langle b\rangle\langle c\rangle$ and $\langle a\rangle\langle c\rangle \neq 0$, then $\langle a\rangle\langle b\rangle\langle c\rangle \neq 0$.*

Proof. See [DPR97]. □

Definition 2.8. *Let a* weight *on a directed graph be a functor W from the directed graph's involutive category of paths to $[\mathsf{L}^\star]_\omega$.*

Most of the time, we will simply write ϕ for $\mathrm{W}(\phi)$ to ease the reading of definitions and proofs.

We will say that α *coincides* with β or equivalently that α and β are coincident if they have the same target node.

An edge β is called a *counter-edge* of α along τ if $\beta \neq \alpha$ and τ is a directed path from the α's target to the β's one, not ending with β, such that $\langle\alpha\rangle\langle\tau^\star\beta\rangle \neq 0$.

Two coincident counter-edges α and β are said to be *composable* (i.e. $\langle\alpha\rangle\langle\beta\rangle \neq 0$ or equivalently they are reciprocally counter edges along the empty path).

Definition 2.9. *A* straight path *is a path that contains no sub-path of the form $\phi\phi^\star$, i.e. that never bounces back in the same edge.*

A weighted directed graph is said to be *split* if any three coincident paths of length one ϕ_1, ϕ_2 and ϕ_3 are such that $\langle\phi_1\rangle\langle\phi_2\rangle\langle\phi_3\rangle = 0$; it is said to be *square-free* if for any straight path ϕ, $\phi\phi = 0$.

Definition 2.10. *A weighted directed graph is said to be a* virtual net *if it is split and square-free.*

Splitness can be rephrased as: any three paths ϕ_1, ϕ_2 and ϕ_3 such that none is prefix of another are such that

$$\langle\phi_1\rangle\langle\phi_2\rangle\langle\phi_3\rangle = 0.$$

2.2. **Directed Virtual Reduction.**

Definition 2.11. *A directed virtual net R is an acyclic virtual net such that for each edge α:*

A. *$\alpha = [b_1, \ldots, b_n]a$, where a, b_1, …, b_n are positive monomials of $\mathsf{L}^\star$. We will denote by α^+ the weight of α without its* filter *$[b_1, \ldots, b_n]$ that is, the monomial a.*

B. *for any $i \neq j$ and for any two counter-edges β_1, β_2 of α along τ_1, τ_2*

$$\begin{array}{lll} \langle b_i\rangle\langle b_j\rangle & = 0 & \mathbf{0}^R(\alpha; b_i; b_j) \\ \langle b_i\rangle\langle \tau_1{}^\star\beta_1^+\rangle & = 0 & \mathbf{1}^R(\alpha; b_i; \tau_1{}^\star\beta_1) \\ \langle \tau_1{}^\star\beta_1^+\rangle\langle \tau_2{}^\star\beta_2^+\rangle & = 0 & \mathbf{2}^R(\alpha; \tau_1{}^\star\beta_1; \tau_2{}^\star\beta_2) \end{array}$$

Given two coincident counter-edges α and β, with weights $[b_1, \ldots, b_n]a$ and $[a_1, \ldots, a_m]b$, then DVR originates a new node and two new edges linking that node to the sources of α and β. These new edges have, respectively, weights b' and a' where $a'b'^\star$ is the stable form of $b^\star a$; this is shown in Figure 4.

Note that new edges produced by a step of reduction have positive weights so that the resulting computation of the execution formula is more appealing for the implementation, as opposed to VR, by the fact that bars are not propagated on residuals.

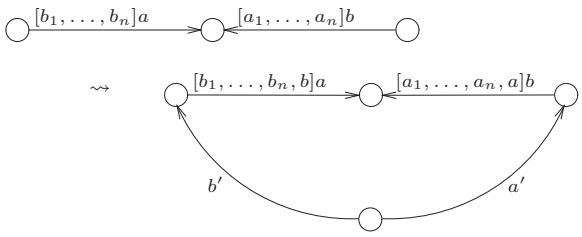


FIGURE 4. Composition performed by DVR.

Definition 2.12. *Given two composable edges α and β, the two edges α' and β' generated by one step of DVR are called* residuals *of α and β respectively. We will denote these residuals by*

$$\mathtt{dvr}(\alpha, \beta) = (\beta', \alpha').$$

Lemma 2.13 ((augmentation))**.** *Let R be a directed virtual net and γ_2 be a counter-edge of γ_1 along τ in R, then*

$$\langle \gamma_1 \rangle \langle \tau^\star \gamma_2 \rangle = \langle \gamma_1^+ \rangle \langle \tau^{+\star} \gamma_2^+ \rangle \neq 0,$$

or equivalently

$$\gamma_1{}^\star \tau^\star \gamma_2 = \gamma_1^{+\star} \tau^{+\star} \gamma_2^+ \neq 0.$$

Proof. See [DPR97]. □

In [DPR97], it has been proved that DVR is sound w.r.t. Girard's execution formula:

Proposition 2.14 ((Invariance))**.** *The execution formula is an invariant of DVR.*

Proof. See [DPR97]. □

2.3. **Optimization Rules.** In this section we prove two properties in order to make DVR more effective, which are also exploited while developing the implementation. These properties are immediate consequences of the orthogonality conditions satisfied by virtual nets and help to gain effectiveness in the computation and to increase the intrinsic parallelism.

Definition 2.15. *Given a directed virtual net R, a* total *edge α is any edge with at most one counter-edge β that is, β is the only edge such that*

$$\langle \alpha \rangle \langle \tau^\star_{\alpha\beta} \beta \rangle \neq 0.$$

In this case we say that α is total w.r.t. *β; if α has no counter edge, it is called* ghost.

This relation is not symmetric: for any two coincident edges α and β such that α is total w.r.t. β, we observe that it is possible that β is not total w.r.t. α, in fact: suppose $\beta = 1$, then any γ coincident with β is composable with β (thus β cannot be total w.r.t. α) but α is not composable with γ otherwise it would contradict the splitness condition, thus α is total w.r.t. β.

Proposition 2.16. *Given two composable edges α and β such that $\mathtt{dvr}(\alpha, \beta) = (1, \alpha')$, then α is total w.r.t. β.*

Proof. In order to get a contradiction suppose that there exists a counter-edge γ of α along the directed path τ. Suppose τ is the empty path, therefore γ coincides with α and $\alpha^\star\gamma \neq 0$ so that $\mathtt{dvr}(\alpha,\gamma) = (\gamma', \alpha'')$; in this case we compute $\langle\gamma\rangle\langle\alpha\rangle\langle\beta\rangle$ and we have $\gamma\gamma^\star\alpha\alpha^\star\beta\beta^\star = \gamma\alpha''\gamma'^\star\alpha'^\star\beta^\star \neq 0$ since it is a stable form, and we get a contradiction with the splitness condition.

If τ is not the empty path, we can apply the same argument to the residual γ' of the reduction sequence along the directed path τ, and derive the property by lemma 2.13 applied to α and γ. □

Proposition 2.17. *Given three coincident edges α, β, γ such that*

$$\mathtt{dvr}(\alpha,\beta) = (\beta', 1)$$

with α and γ composable, then the residual of γ is not composable with β'.

Proof. Suppose $\gamma'^\star\beta' \neq 0$ so this product has a stable form, let say $\tilde{\beta}'\tilde{\gamma}'^\star$, then $\langle\gamma\rangle\langle\alpha\rangle\langle\beta\rangle = \gamma\gamma^\star\alpha\alpha^\star\beta\beta^\star = \gamma\alpha''\gamma'^\star\beta'\beta^\star = \gamma\alpha''\tilde{\beta}'\tilde{\gamma}'^\star\beta^\star \neq 0$ and we get a contradiction with the splitness condition. □

Corollary 2.18. *(soundness of the optimization of one) If $\mathtt{dvr}(\alpha,\beta) = (1,\alpha')$, then no further composable edge γ can give 1 as residual of the composition with α.*

Proof. The two residuals in the source of α have weight 1 so that they are composable and this is a contradiction with the proposition 2.17. □

This corollary allows an optimization rule, in fact the configuration produced by the DVR step $\mathtt{dvr}(\alpha,\beta) = (1,\alpha')$ acts as a compound operator: the edge with weight 1 is there just to say that all the coincident edges have to be transferred on the source of the edge α', so we propose to transform this configuration by removing the edge β' with weight 1 and using the edge α' for linking the target of β' and the target of α' (see Figure 5).

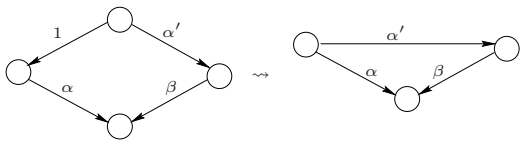


FIGURE 5. An optimization rule.

Now we will prove another property of DVR, which states that when two edges α'_1 and α'_2 are residuals of directed virtual reduction of α_1 and α_2 against the same edge β, they are coincident on the source of β (evident by definition of a DVR step), but are not composable because of splitness:

Proposition 2.19. *Given an edge β composable with two coincident edges α_1 and α_2 we have that $\langle\alpha'_1\rangle\langle\alpha'_2\rangle = 0$, where α'_1 is the residual of α_1 and α'_2 is the residual of α_2.*

Proof. Suppose $\langle\alpha'_1\rangle\langle\alpha'_2\rangle \neq 0$ then we have, $\alpha'_1\alpha'^\star_1\alpha'_2\alpha'^\star_2$ and so $\alpha'^\star_1\alpha'_2 = \alpha''_2\alpha''^\star_1 \neq 0$.

By augmentation lemma we have

$$\langle\alpha_1\rangle\langle\beta\rangle\langle\alpha_2\rangle = \langle\alpha_1^+\rangle\langle\beta^+\rangle\langle\alpha_2^+\rangle,$$

this is

$$\alpha_1 {\alpha_1}^\star \beta \beta^\star \alpha_2 {\alpha_2}^\star$$

and by reduction we obtain

$$\alpha_1 \beta' \alpha_2'' {\alpha_1''}^\star {\beta''}^\star \alpha_2^\star$$

and so this is a non null stable form so that it is different from zero. □

Property 2.19 allows the implementation of another optimization rule. Specifically, we know that every new node v created after a DVR step is the source of only two edges, say β_1 and β_2, therefore all the edges coincident in v can be separated in two sets: the residuals of a DVR step involving β_1 and the residuals of a DVR step involving β_2. Each edge in a set is orthogonal to any edge in the same set, therefore there is no need to perform DVR steps between edges belonging to the same set since the composition will actually produce a null result.

2.4. **Translation of Sharing Graphs into Directed Virtual Nets.** In order to solve the problem of the pairing of duplication operators Gonthier et al. added to the sharing graphs a local level structure. Each operator is decorated with an integer tag that specifies the *level* at which it lives. Furthermore in order to manage these levels a set of control operators is required.

More precisely sharing graphs are non-oriented graphs built from the indexed nodes represented in Figure 6. These nodes are called sharing operators and distinguished in two groups. The first group includes the operators in the original Lamping's work: application, and abstraction; the second one is constituted by a family of nodes of the same kind (the so called *muxes*) accordingly to the following definition:

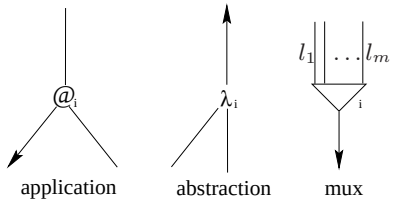
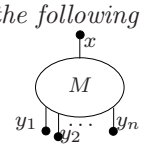


FIGURE 6. Sharing graph operators.

Definition 2.20. *A node* mux *or multiplexer is a node with an arbitrary number of premises each one having a* name n *and a* lift l_n*; like the other nodes, muxes have an index of level* i*.*

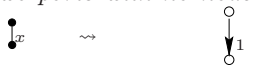
The translation of a sharing graph with muxes is defined by induction:

Definition 2.21. *A sharing graph* M *with root* x *and context* $y_1, \ldots, y_n$ *is translated into a directed virtual net in the following way*

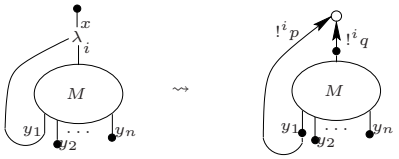


where bullets indicates ports of a sharing graph,

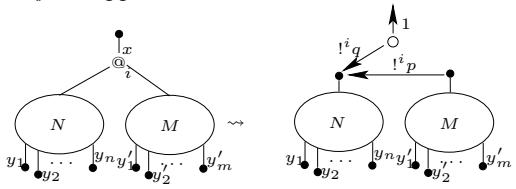
- *If x is a link between two ports with no node*



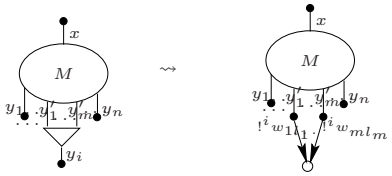
- *If x is a port of an abstraction*



- *If x is a port of an application*



- *If y_i is a port of a mux*



When the nodes introduced by the translation present the configuration described in the next definition they are reduced by amalgamating edges as in Figure 7.

Definition 2.22. *A node with n coincident edges $\alpha_1, \ldots, \alpha_n$ and an edge β with source the target of the α_i's is erased and all the α_i's are replaced by edges α'_i where the source of α'_i is the source of α_i, the target of α'_i is the target of β and the weight of α'_i is $\alpha_i\beta$.*

FIGURE 7. Amalgamation of edges.

With the help of an example we show how to change a λ-term into a directed virtual net. In Figure 8, starting from the syntactic graph of the λ-term representing the Church numeral 2 applied to the identity that is, by using Krivine's notation, $(\lambda f \lambda x (f)(f)x)\lambda x x$, we obtain a sharing graph by adding the control operators, expressed in the multiplexer syntax, and annotating each node by level indexes. Then edges are oriented, unfolded and labeled with monomials in $\mathsf{L}^\star$ in accord to the rules expressed by Definition 2.21 and Definition 2.22. Last step consists of grouping together arrows going in the same direction, the result of this operation is a directed virtual net, see Figure 9.

Let us check that the obtained net is indeed a directed virtual net:

- it is obviously a directed graph with no circuits,

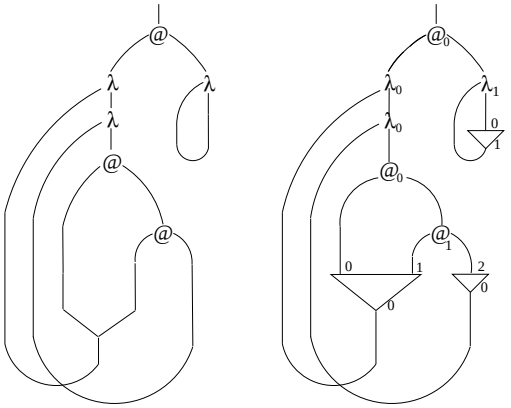


FIGURE 8. Representation of a λ-term.

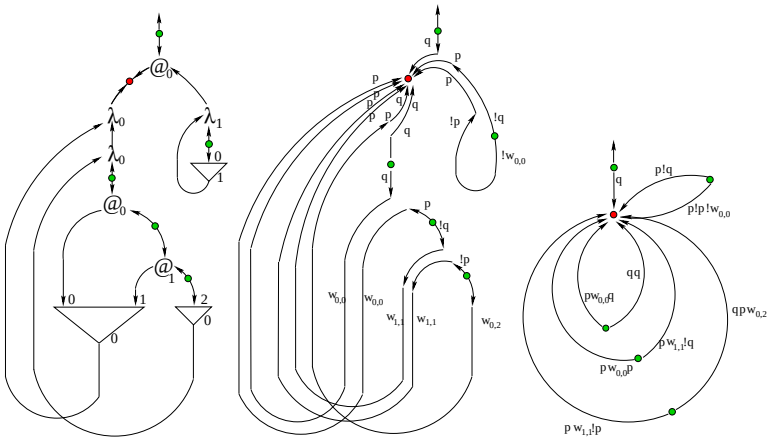


FIGURE 9. Encoding of a sharing graph into a directed virtual net.

- square-freeness can be proved by induction on the translation of λ-terms,
- splitness has to be verified for all the triples of coincident edges; as an example we explicitly test the splitness condition for three coincident edges:

  $$\langle q\rangle\langle qpw_{02}\rangle\langle qq\rangle = qq^\star qpw_{02}w_{02}p^\star q^\star qqq^\star q^\star = qpw_{02}w_{02}p^\star qq^\star q^\star = 0$$

  because of $p^\star q = 0$; the rest of this verification is left to the reader.

3. Half Combustion Strategy

In [DPR97], a strategy called *combustion* is presented in order to organize DVR in such a way that no filter must be kept. This strategy works on *full* directed virtual nets that are directed virtual nets where each edge is either ghost (see Definition 2.15) or has a positive weight.

Since a ghost edge is an edge for which no more compositions will occur, sources of ghost edges never receive residual edges of ghost edges, thus let us define the

(out-)*valence* of a node as the number of non-ghost edges having that node as source.

The combustion strategy of a full net starts from a node v of valence zero (i.e. with no future incoming edge or equivalently having only ghost outgoing edges) and composes all the pairs of coincident counter-edges on v as an atomic action. Using the combustion strategy we can give up filters because after the composition is performed, all those edges become ghost edges.

From the point of view of a parallel implementation, the drawback of this strategy is that the composition of the coincident counter-edges can be started only when a node becomes of valence zero. More specifically, in case many processes are used to perform DVR (recall this is desirable anytime we want to fully exploit the computing power of parallel or distributed systems equipped with a large number of processors), we might incur the risk that, at a given time instant, only a subset of those processes host nodes of valence zero. In such a case, all the other processes cannot simultaneously proceed with DVR steps (i.e. they need to wait until some node they host becomes of valence zero), thus limiting the degree of parallelism while performing the reduction.

We define below the HC strategy that like combustion does not require to keep filters and, in addition, allows the composition to be performed even on nodes having valence greater than zero, thus allowing high degree of parallelism. HC relies on the following notion of *semifull* directed virtual net which is a generalization of the notion of full directed virtual net.

Let us call *semifull* directed virtual net a directed virtual net in which each edge either is weighted by a positive monomial (i.e. its weight has no filter) or all its coincident counter-edges are weighted by a positive monomial (i.e. it can be composed exclusively with edges having a positive weight). An example of a node in a semifull directed virtual net is shown in Figure 10. In this example, the coincident counter-edges of edges with weight $[a_{i1}, \ldots, a_{ij_1}]b_i$ are among those edges weighted with a_1, a_2, ..., a_m.

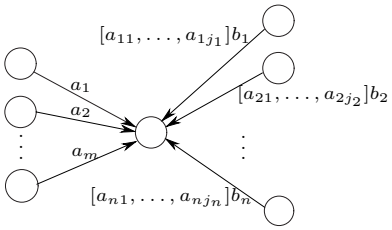


FIGURE 10. A node in a semifull directed virtual net.

Below we give the definition and provide the soundness of the HC strategy.

Definition 3.1. *Given a composable edge α with positive weight in a semifull directed virtual net R, we have to consider two cases:*

1. *if α has no non-positive coincident counter-edge and a positive one β, then the* half combustion strategy *(HC) performs the composition of β with α and possibly with every non-positive edge composable with β;*
2. *if the set $\{\beta_1, \ldots, \beta_n\}$ of non-positive edges composable with α is non-empty then HC performs all the possible compositions of α with the β_is.*

Proposition 3.2. *If R' is obtained from the directed virtual net R by the HC strategy and R is semifull then so is R'.*

Proof. Consider an edge α having positive weight a as in the Definition 3.1, and suppose we stay in the second case of the Definition, all the composable edges with non positive weights coincident with α are the β_i's with weights $[a_{i1}, \ldots, a_{ij_i}]b_i$ for $i = 1, \ldots, n$ as in Figure 11.

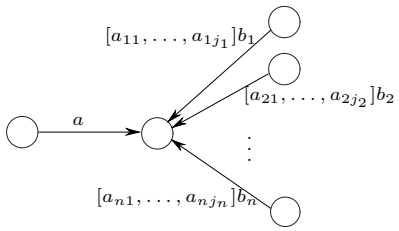


FIGURE 11. Edges in a semi-full node.

If we apply a step of the HC strategy by performing a DVR step between α and β_i for $1 \leq i \leq n$, we obtain

$$\mathtt{dvr}(\alpha, \beta_i) = (\beta_i', \alpha_i')$$

where the weight of α is $[b_1, \ldots, b_n]a$ and the weight of β_i is $[a_{i1} \ldots a_{ij_i}, a]b_i$ and the two new edges β' and α' have a positive weight, see Figure 12.

Therefore, now the set of the coincident filtered edges has been enlarged with α, but α is no more composable with the β_i's because of its filter and all the generated edges α_i's and β_i's have positive weights by definition of DVR. As a consequence, all the coincident filtered edges (including α) are not composable with each other. Thus the obtained directed virtual net is semifull.

If we stay in the first case of the definition 3.1, performing the composition of β with all its non-positive coincident counter-edges we obtain the same configuration as in the previous case, moreover we compose α with (the residual of) β and so all the non-positive edges incident in the node are not further composable. Note that the set of non-positive edges composable with β can possibly be empty, in this case HC just composes α and β.

□

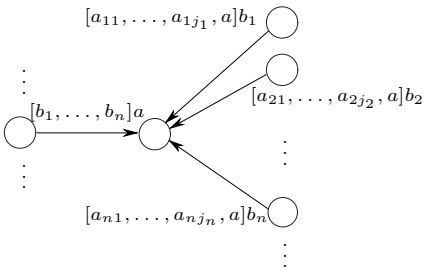


FIGURE 12. Edges after the composition performed by HC.

We recall that the translation presented in Section 2.4 associates with any λ-term a full directed virtual net (see also [DR93, GAL92]). As full nets are particular instances of semifull ones, HC actually represents a reduction mechanism for λ-calculus.

Beyond the exploitation of parallelism, another interesting property of HC is that we can separate the edges ending on a node in two distinguished sets. In other words, the strategy associates a mark with each edge: *incoming* or *combusted*. When created, edges are marked as incoming. One step of reduction consists of picking an incoming edge α and performing all the compositions with coincident combusted edges. Then α is marked as combusted.

Note that an edge may be marked as combusted even when it has a positive weight, namely if all the combusted edges coincident with α are not composable with α, with the particular case where the set of combusted edges coincident with α is empty as in case 1 of Definition 3.1. On the other hand, at any step any incoming edge has a positive weight. As an edge is marked combusted only after having been (successfully or not) composed with every coincident combusted edges, one easily sees that two combusted edges are never composable. Thus this suggests that we can organize the computation in such a way that the only meaning associated with filters is about the belonging of an edge to the first or to the second set (thus, like in the combustion strategy, filters can be actually discarded). We have embedded this simplification among others in the parallel implementation we present in the next section.

4. The Implementation

This section is devoted to the description of the implementation of PELCR and is organized as follows. We first provide the outline of data structures we have used and the high level description of the parallel program. Then we enter details on any aspect and/or any optimization characterizing the implementation. Actually, the material presented in this section describes the implementation independently of the specific language used to develop it (the `C` language for our case).

4.1. Data Structures and Code Organization.

Each processor i of the architecture hosting PELCR runs a process P_i which is an instance of the executable code associated with the parallel program. We assume there is a master process, that for the sake of clarity will be identified as P_0. All the other processes will be referred to as slave processes. Processes communicate exclusively by exchanging messages and the communication channels among processes are assumed to be FIFO (this is not a limitation as the most widely used message passing layers, such as PVM or MPI, actually provide the FIFO property to communication channels). We call *pending* message any message already stored in the communication channel, which has not yet been received by the recipient process.

We associate with each node v an identifier, namely $id(v)$. Each edge $e = (v_1, v_2)$ is therefore associated with the pair of node identifiers $(id(v_1), id(v_2))$ thus the weighted edge is represented by the triple $(id(v_1), id(v_2), W(e))$.

As discussed in Section 2.3, by Property 2.19 any edge e incident on a node v can be seen as belonging to one of two distinct sets depending on which between the two edges having v as source originated e through composition. We call the two sets of edges as LEFT set and RIGHT set, and we associate with each edge e an additional information, namely $Side(e)$, indicating whether e belongs to the

LEFT or the RIGHT set. This information allows us to reduce the number of edge compositions according to the HC strategy which must be performed during the computation. Specifically, given two edges e and e' incident on a same node v, if the side of the two edges is the same, no composition involving e and e' must be performed at all since we a priori know that it will produce null result. On the other hand, if $Side(e) \neq Side(e')$, composition must be performed to determine the result, which can be either null or non-null.

In the general case, each process P_i hosts only a subset of the nodes of the graph. Therefore, given an edge $e = (v_1, v_2)$, there is the possibility that v_1 and v_2 are hosted by distinct processes. In Figure 13 we show an example of this. The interesting point in the example is that when process P_i performs the composition between the edges e_1 and e_2 incident on node v according to HC, then a new node, namely v' is originated together with two new edges, namely e_3 and e_4 incident on nodes v_1 and v_2 respectively. The new node v' can be hosted by any process, and process P_i is the one which establishes where v' must be actually located; in our example, P_i selects P_j. We will come back to the selection issue in Section 4.3 when describing the load balancing module that establishes how new nodes must be distributed among processes. Note that, in case one of the newly produced edges should have weight one, the *optimization of one rule* described in Section 2.3 (see Figure 5), allows avoiding the real creation of that edge. Also, the only edge really created has as source a node already within the directed virtual net, thus no new node needs to be created and addressed to some process.

In our implementation $id(v')$ is a triple $[t, P_i, P_j]$ where P_i is the process that created the node v', P_j is the process hosting that node and t is a time-stamp value assigned by P_i. The time-stamp is managed by P_i as follows: it is initialized to zero and anytime P_i originates a new node, it is increased by one.

When the new node v' is originated by P_i, the creation must be notified to P_j. Furthermore, both P_k and P_h must be notified of the new edges e_3 and e_4 incident, respectively, on v_1 and v_2. In our implementation we use message exchange only for the notification of new edges, while we avoid to explicitly notify the creation of the new node v' to P_j. Process P_j will actually create the node v' upon the receipt of the first message notifying a new edge incident on v'. We will refer to this type of node creation as *delayed creation*. It allows us to reduce the amount of notification messages exchanged among processes.

Applying the delayed creation technique to the example in Figure 13 means that node v_1 is created by P_k only upon the receipt of the message carrying the information of the edge e_3 incident on v_1 (recall that this message is sent by P_i). Similarly, P_h will create v_2 only upon the receipt of the notification message for the edge e_4 (also this message is sent by P_i).

By previous considerations we get that any message exchanged between two processes carries the information of a new edge. Specifically, a message carrying the information associated with the edge $e(v_1, v_2)$ has a payload consisting of the tuple $[[t, P_i, P_j], [t', P_l, P_m], W(e), Side(e)]$ where $id(v_1) = [t, P_i, P_j]$, $id(v_2) = [t', P_l, P_m]$, $W(e)$ is the weight of e and $Side(e)$ is the edge side.

P_i keeps track of information related to local nodes in a list $nodes_i$. Any element in $nodes_i$ has a compound structure. In the remainder of the article we identify the structure in $nodes_i$ associated with a node v as $nodes_i(v)$. As relevant field of the structure $nodes_i(v)$ we have a list, namely $nodes_i(v).combusted$, containing the

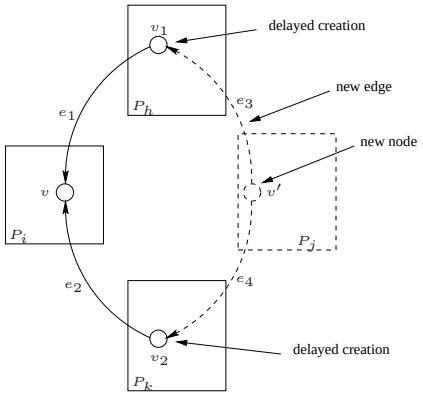


FIGURE 13. Creation of a new node.

edges incident on the node v which have already been composed (i.e. the combusted edges of the HC strategy). The list $nodes_i(v).combusted$ is partitioned into two sub-lists, namely $nodes_i(v).combusted.LEFT$ and $nodes_i(v).combusted.RIGHT$, containing edges having $Side()$ equal to LEFT and RIGHT respectively.

A buffer $incoming_i$ associated with P_i is used to store received messages. For what we have explained above, any message stored in $incoming_i$ carries information related to a new edge which must be added to the virtual net and composed with already combusted edges, if any, incident on the same node. Such an edge is actually an incoming edge of the HC strategy. Therefore, the buffer $incoming_i$ represents a kind of *work list* for process P_i, as, according to HC, any incoming edge associated with a message stored in $incoming_i$ requires P_i to compose it with all the already combusted edges incident on the same node. Performing such a composition represents the work associated with the message carrying the edge.

For each process P_i, except the master process P_0, both $incoming_i$ and $nodes_i$ are initially empty, meaning that initially there is no node of the directed net managed by P_i, nor there are incoming edges for it. Instead, P_0 is such that its list $nodes_0$ is empty but its buffer $incoming_0$ contains a set of messages, one for each initial edge of the virtual net (recall that the initial edges are all incoming). Note that this does not mean P_0 is a bottleneck for the parallel execution since the load balancing mechanism we have implemented (see Section 4.3) promptly distributes new edges produced in the early phase of the execution among all the processes.

In Figure 14 we show the high level structure of the algorithm implemented by the software modules we have developed. Before entering the pseudo code description, we recall that the HC strategy is such that, any incoming edge of which process P_i becomes aware by extracting the corresponding message from $incoming_i$, must be immediately composed with the preexisting edges incident on the same node, without additional delay. Furthermore, given a message m carrying the information of a new edge $e = (v_1, v_2)$, we denote as $m.target$ the node identified by the information $id(v_2)$ carried by m (recall that $id(v_2)$ is the previously described triple) and as $m.source$ the node identified by the information $id(v_1)$ carried on the same message. $e.target$ and $e.source$ have similar meaning when referring to an edge e. Also, we denote as e_m the edge carried by m.

```
program P_i;
1  initialize();
2  while not end_computation do
3     ⟨collect all incoming messages and store them in incoming_i⟩
4     while not empty(incoming_i) do
5        ⟨extract a message m from incoming_i⟩;
6        if m.target ∈ nodes_i 'node already in the local list'
7        then
8           for each edge e ∈ nodes_i(m.target).combusted do
9              if Side(e_m) ≠ Side(e)
10             then
11                ⟨compose e_m with e⟩;
12                ⟨select the destination process P_j for hosting the node possibly
                   originated by the composition⟩;
13                ⟨send the edges produced by the composition to P_k and P_h
                   hosting m.source and e.source respectively⟩
14          endfor
15       else ⟨add m.target to nodes_i⟩; 'delayed creation'
16       ⟨add e_m to nodes_i(m.target).combusted.Side(e_m)⟩
17    endwhile;
18    ⟨end_computation = check_termination()⟩;
19 endwhile
```

FIGURE 14. Pseudo code for process P_i.

The procedure *initialize*() sets the initial values for all the data structures. The procedure *empty*() checks whether the buffer storing received messages is empty. In the positive case, process P_i has no work to be performed, thus it invokes the procedure *check_termination*() to check if the computation is actually ended, i.e. no message will arrive (in Section 4.4 we will provide details on how the detection of the termination is implemented). In the negative case, it extracts a message from the buffer $incoming_i$ and performs the composition of the corresponding incoming edge.

By the test in line 11 we exploit information about $Side(e_m)$ to avoid unnecessary edge compositions. The pseudo code structure also points out that process P_i checks for the presence of pending messages only when $incoming_i$ is empty (i.e. when P_i has no more work to be performed unless new pending messages carry it). This behavior aims at reducing the communication overhead. Specifically, a procedure to check whether there are pending messages is realized typically by using `probe` functions supported by the used communication layer. P_i invokes the execution of a `probe` function to test if there is at least a pending message. If there is at least one such message, then a `recv` procedure is executed to receive that message and store it into $incoming_i$. As pointed out in other contexts [DNRD96], `probe` functions may be expensive, therefore, they should be executed only when a further delay could actually produce negative effects on performance. In the general case, delaying the `probe` call until all the messages stored in $incoming_i$ have been processed should not produce negative effects. This is the reason why, in the general case, we suggest to perform the `probe` call only when $incoming_i$ becomes empty. However, we noted that depending on the particular hardware/software architecture and on the adopted message passing layer, excessive delays in receiving pending messages could impact negatively on the performance of the communication layer due to buffer saturation. This is the case we have observed for our implementation based on MPI. For this reason we have done a light modification to the general code

structure in Figure 14 in order to avoid excessively infrequent `probe` calls (and message receipts).

Beyond the overhead due to probe calls, another important issue is the overhead related to send and receive operations. A solution to bound this overhead will be discussed in the following subsection. Then we will present the policy we have selected for balancing the load among processes and other relevant aspects related to the implementation.

4.2. **Message Aggregation.** The cost of sending and receiving a physical message, paid part by the sender and part by the receiver, can be divided into two components: (i) an overhead that is independent of the message size, namely oh, and (ii) a cost that varies with the size of the message, namely $s \times oh_b$ where s is the size (in bytes) of the message and oh_b is the send/receive time per byte. oh typically includes the context switch to the kernel, buffer reservation time, the time to pack/unpack the message and, in case of distributed memory systems, the time to setup the physical network path. Instead, oh_b takes into account any cost that scales with the size of the message.

oh is usually higher than oh_b, as shown in [XH96] up to two orders of magnitude, therefore it results usually more efficient to deliver several information units (i.e. more than one application message) with a single physical message, in such a way that a single pair of send/receive operations is sufficient to download many data at the recipient process. This allows the reduction of the static overhead oh for each information unit, thus originating efficient parallel executions, especially in the case of fine grain computations like DVR. As an example, if three application messages of size s constitute the payload of a single physical message then the cost to send and receive these application messages is reduced from $3oh + 3s \times oh_b$ to $oh + 3s \times oh_b$.

We present below the optimization we have embedded in the communication modules via the aggregation of application messages in a single physical message. Each process P_i collects application messages destined to the same remote process P_j into an aggregation buffer $out\_buff_{i,j}$. Therefore, there is an aggregation buffer associated with each remote process. Application messages are aggregated and are infrequently sent via a single physical message. The higher the number of application messages aggregated, the greater the reduction of the static communication cost per application message; we call this positive effect Aggregation Gain (AG). However, the previous simple model for the communication cost ignores the effects of delaying application messages on the recipient process. More precisely, there exists the risk that the delay produces idle times on the remote processes which have already ended their work and are therefore waiting for messages carrying new work to be performed; we call this negative effect Aggregation Loss (AL). Previous observations outline that establishing a suited value for the aggregation window (defined as the number of application messages sent via the same physical message) is not a simple task.

In our implementation, the module controlling the aggregation keeps an age estimate for each aggregation buffer $out\_buff_{i,j}$ by periodically incrementing a local counter $c_{i,j}$. The value of $c_{i,j}$ is initialized to zero and is set to zero each time the application messages aggregated in the buffer are sent. At the end of the composition phase of an incoming edge extracted from the local work list $incoming_i$, $c_{i,j}$ is increased by one if at least one message is currently stored in the aggregation

buffer $out\_buff_{i,j}$. Therefore, one tick of the age counter is equal to the average combustion time of an incoming edge and the counter value represents the age of the oldest message stored in the aggregation buffer.

The simplest way to use previous counters is to send the aggregate when the associated counter reaches a fixed value, referred to as maximum age for the aggregate, or when the work list of the process is empty. In this case there is no need to delay the aggregate anymore as the probability to put more application messages into it in short time is quite small, so the delay will not increase AG and will possibly produce an increase of AL. We will refer to this policy as Fixed Age Based (FAB). Although this policy is simple to implement and does not require any monitoring for the tuning of the maximum age over which the aggregate must be sent, it may result ineffective whenever a bad selection of the maximum age value is performed.

To overcome this problem we have implemented a Variable Age Based (VAB) policy, which is an extension of FAB, having similarities with an aggregation technique presented in [CAGRW98] for communication modules supporting fine grain parallel discrete event simulations. In VAB, anytime the messages aggregated in $out\_buff_{i,j}$ are sent, the message rate achieved by the aggregate is calculated. This rate is used to determine what the maximum age for the next aggregate should be. The dynamic change of the maximum age after which an aggregate must be sent, allows the aggregation policy to adapt its behavior to the behavior of the overlying application.

To implement VAB, P_i is required to maintain an estimate $est_{i,j}$ of the expected arrival rate in each aggregation buffer $out\_buff_{i,j}$ (the higher such rate, the higher AG for that buffer). This estimate can be computed by using statistics related to a temporal window. If the arrival rate for the current aggregate in $out\_buff_{i,j}$ is higher than $est_{i,j}$ then the maximum age for the next aggregate into that buffer is increased by one since the application is likely to start a period of bursty exchange of application messages from P_i to P_j. Therefore a slight increase in the maximum age is likely to relevantly increase AG. If the arrival rate falls below $est_{i,j}$, then the maximum age is decreased by one (provided it is greater than one). An upper limit on the maximum age can be imposed in order to avoid negative effects due to AL (i.e. in order to avoid excessive delay for the delivery of the aggregate at the recipient process).

4.3. **Load Balancing.** Whenever the composition between two edges is performed by a process P_i then a new node is originated and P_i must select a process P_j (possibly $j = i$) which will host the new node. In order to provide good balance of the load we have implemented a selection strategy for the destination process which uses approximated state information related to the load condition on each process.

In our solution we identify the number of unprocessed application messages upm stored in the buffer $incoming_i$ as the state information related to the load condition on P_i. P_i keeps track of the values of upm related to itself and to the other processes into a vector UPM_i of size n (where n is the number of processes). $UPM_i[i]$ records the current value of the number of unprocessed application messages of P_i. $UPM_i[j]$ records the value of the number of unprocessed application messages of P_j known by P_i. These values are spread as follows. Whenever P_i sends a physical message M to P_j, the value of $UPM_i[i]$ is piggy-backed on the message, denoted $M.UPM$

([1]). Whenever a physical message M sent by P_j to P_i is received from P_i, then $UPM_i[j]$ is updated from $M.UPM$ (i.e. $UPM_i[j] \leftarrow M.UPM$). The information on the load conditions kept by the UPM vectors is approximated for two reasons:

- there exists the possibility that when P_i receives M from P_j the current value of $UPM_j[j]$ is different from $M.UPM$;
- the current value of $UPM_i[i]$ is not an exact representation of the current load of P_i as it does not count application messages carried by pending physical messages; these application messages represent work to be performed which has not yet been incorporated into the buffer $incoming_i$.

We note, however, that obtaining more accurate state information on the load condition of a process would require the exchange of additional physical messages or, at worst, a synchronization among processes which could produce unacceptable negative effects on the performance. Anyway, it is important to remark that the FIFO property for the communication channels guarantees that each time a physical message M sent by P_j is received from P_i, the piggy-backed value $M.UPM$ refers to a more recent load condition as compared to the one indicated by the current value of $UPM_i[j]$.

Based on the values stored in UPM_i, we have implemented a selection policy for the destination process of a new node which is a modified round-robin. It works as follows. P_i keeps a counter rr_i initialized to zero which is updated (module n) each time a new node is produced by P_i. The current value of rr_i is the identifier of the process which should host the new node according to the round-robin policy. P_i actually selects P_{rr_i} as destination if $UPM_i[rr_i] < UPM_i[i]$; otherwise P_i selects itself as destination for the new node. In other words, each process distributes the load in round-robin fashion unless, at the time the load distribution decision must be taken, the local load is lower than that of the remote process which should be selected.

4.4. **Termination Detection.** The implementation of the termination detection relies on the use of additional control messages. Specifically, anytime a process distinct from the master P_0 has no more work to be performed, it sends to P_0 a message carrying information about both the number of application messages received from other processes, which have already been processed and the number of application messages produced for other processes. Such a message will be referred to as `status` message. When the master P_0 detects that each process has already elaborated all the application messages produced for it, P_0 discovers that the computation is over and notifies the termination to the slave processes. This is done through the send of a `termination` message. By looking at the structure of the code in Figure 14, it can be seen that process P_i executes the *check_termination*() procedure only when no work to be performed has been detected (i.e. when $incoming_i$ is empty). This points out that no synchronization is required (i.e. P_i sends its `status` message without blocking to receive an acknowledgment; it will possibly receive the `termination` message during a future execution of the *check_termination*() procedure) ([2]). On the other hand, the master P_0 checks for incoming `status` messages

[1] We use "M" to denote a physical message in order to distinguish it from an application message previously denoted as "m".

[2] Actually, to keep low the overhead due to `status` messages, P_i sends one such message to the master only if its status has changed since the last `status` message was sent.

and possibly sends the `termination` messages only when it has no more work to be performed. Note that the consistency of the information collected by P_0 through `status` messages is guaranteed by the FIFO property of communication channels.

4.5. **On-the-Fly Storage Recovery.** At the end of the computation we get that for all the nodes of the final graph the incident edges are ghost. However only some of those nodes belong to the normal form of the reduction. We recall that the nodes of the initial graph belonging to the normal form are the nodes initially having only ghost incident edges; this set of nodes will be referred to as the *border*. Starting from the border, we can determine the whole normal form: it contains those nodes linked to the border by a directed path.

We have embedded in the implementation a technique to discard on-the-fly nodes that do not belong to the normal form. We have taken this design choice for keeping low memory usage with the twofold aim of (i) increasing the efficiency of the underlying virtual memory system, and (ii) allowing efficient management of the data structures maintained at the application level. As respect to the latter issue, anytime an unprocessed application message m is extracted from the buffer $incoming_i$ (see line 5 of the pseudo code in Figure 14), process P_i must access information associated with the node $m.target$, if it already exists. Such an information is maintained in the structure $nodes_i(m.target)$ (see lines 6 on of the pseudo code in Figure 14). To retrieve the virtual memory address for this structure we have used an hash table with chaining for handling collisions, which keeps an active entry for each node in the list $nodes_i$. Discarding nodes of valence zero that do not belong to the normal form allows keeping low the number of entries of the hash table, thus allowing efficient access to the table at anytime.

The on-the-fly storage recovery technique we have implemented tracks whenever a node becomes of valence zero and removes it if there is no directed path towards nodes of the border. This is implemented through a particular type of application messages we call `EOT` (End-of-Transmission) messages. Specifically, for each initial node v that does not belong to the border we insert an `EOT` message. If we ensure that `EOT` messages are processed only after all the messages carrying edges destined to v have been already processed, then we detect upon processing of the `EOT` messages that no new edge will have v as its target. This means that node v, and all the edges pointing to it, can be deleted. Before removing this node, the `EOT` message is propagated to the sources of the edges pointing to v. We note that, since each node is source of two edges, we expect the arrival of two `EOT` messages (one from both sides) before handling the removal of the node. Therefore, if for a node v we have no `EOT` message destined to it, or at most only one of such messages, it means that v has a directed path to the border, thus it belongs to the normal form. In this case no removal takes place.

Actually, guarantees that the `EOT` messages will be processed only after all the messages carrying incoming edges destined to the same node have already been processed, is trivially achieved thanks to the FIFO property of communication channels.

5. Experimental Results

In this section we report experimental results demonstrating the effectiveness of our implementation, and thus of both the HC strategy underlying it and the combination of all the optimizations for the run-time behavior we have presented

and embedded within PELCR. As already pointed out, the implementation has been developed using the `C` language and MPI as the underlying message passing layer. A major advantage of using such a standard interface for message passing functionalities is that it makes the software highly portable. This allowed us to test the implementation on a wide set of platforms such as SMP machines with Linux, IBM mainframes like SuperPower 3 and SuperPower 4 with AIX, shared memory Sun Ultra Sparc machines and Alpha Digital Microchannel clusters.

In this section we report performance results obtained in the case of an IBM server pSeries 690, with 32 Power4 CPUs - 1.3 GHz, 64 GB RAM, running IBM AIX 5.2 ML1+ as the operating system. We have selected the results obtained with this machine as representative especially because of the larger number of available processors, as compared to the other architectures. This allows us to better observe whether and how the performance provided by PELCR scales while increasing the computing power.

Before entering details related to the experimental results, we note that there exists an approach to reduce the computation time in optimal reduction systems based on an optimization known as *safe operators* [AC97]. It allows the merging of many control operators in a compound one acting as the sequence, thus exhibiting the ability to strongly decrease the number of interactions. Actually, this approach has been the basis for the implementation of the so called BOHM (Bologna Higher Order Machine), which is a sequential machine for optimal reduction that has been demonstrated to provide better performances compared to non-optimal interpreters such as CAML and HASKELL (see [AG98]). Compared to this approach, we tackle the issue of increasing the speed of the reduction in an orthogonal way. Specifically, we do not use merging of operators to reduce the number of interactions, instead we exploit computing capabilities of multiple processors within the architecture to keep low the reduction time via parallel computation of the reduction itself. This approach can be applied to any computation issued from the Geometry of Interaction. Note that we experimentally observe speedup in parallel evaluation of terms typable in systems with intrinsic complexity ELL (or LLL) [Gir95]; these terms are evaluated in such a way that the safe operators optimization is not needed. As a consequence, our approach is expected to speedup the execution even for the cases in which safe operators cannot be effectively employed, or even when optimal reduction does not affect the efficiency of computation like in non-higher order terms.

The results we report in this section refer to the following two different test cases:

- `DD4`, which corresponds to the λ-term $(\delta)(\delta)\underline{4}$ where $\delta = \lambda x(x)x$ represents the self application. The normal form of this term represents the Church's integer $(4^4)^{4^4}$.
- `EXP3`, which corresponds to the ELL term

  $$\texttt{Ite}((\texttt{Mult})2, 1, \texttt{Ite}((\texttt{Mult})2, 1, \texttt{Ite}((\texttt{Mult})2, 1, 4))),$$

  whose normal form represents the iterated exponential $2^{2^{2^4}}$. For a precise relation between this ELL term and the multiplicative linear logic proof net from which the dynamic graph to be executed is obtained, we refer the reader to [Ped96].

For both these two cases, the shared result of the HC strategy has a number of nodes which is on the order of hundreds of thousands (for `DD4` this number even

reaches about one million and half), therefore they are large enough case studies to stress the behavior of our implementation.

Before presenting the results, we provide details on the main parameters we have measured in the experiments.

5.1. **Measured Parameters.** A measure of success of any parallel implementation is how significantly it accelerates the computation. Typically the acceleration is expressed by the so called *speedup*, evaluated as the ratio between the sequential execution time on a single processor and the parallel execution time on multiple processors. Actually, this is a fundamental parameter to consider, not only because it expresses the amount of increase in the execution speed while increasing the power of the underlying computing system, but also because the speedup curve provides indications on how the execution speed scales while increasing the computing power. Linear speedup means that the execution speed scales linearly vs the computing power. This is an indication that the parallel implementation maintains the same effectiveness independently of the number of used processors, thus the implementation itself does not suffer, e.g., from excessive increase in the communication overhead while increasing the number of processes involved in the parallel execution. Actually, we also report the ratio between the observed speedup and the ideal speedup that can be achieved with a given degree of parallelism, i.e. with a given amount of used processors. (We recall that the ideal speedup on n processors is equal to n, which means we experience no overhead but only gain by distributing the work to be performed on the n processors.) This parameter provides indications on the extent to which the parallel implementation can be considered effective, independently of the shape of the speedup curve. Specifically, if we have a linear speedup curve but a low ratio over the ideal speedup, it means that the parallel implementation, although not particularly suffering from increase in the overhead due to the parallelization while increasing the amount of processors, is anyway ineffective, e.g. due to inadequate structuring of the parallel algorithm it implements.

Beyond speedup, another parameter we report is the *wall-clock time* for the reduction. This parameter expresses the real time cost for a given reduction and also how it varies while increasing the computing power of the underlying platform. It is a fundamental parameter to report since it provides indications on whether the speedup curve has been evaluated over a representative interval for what concerns the number of used processors. Specifically, if wall-clock time values of few seconds or less are achieved while increasing the number of processors, then it means that an additional increase in the computing power does not make sense for this specific reduction (this is because response time of few seconds or less is typically considered satisfactory even for the case of an interactive end-user, i.e. the case in which responsiveness is actually a critical issue to address), hence the speedup has been evaluated over an adequate interval for what concerns the degree of parallelism.

The wall-clock time and the speedup are parameters that express the effectiveness of the parallel implementation when evaluated globally. However, we are also interested in observing the effects of specific optimizations we have proposed. As respect to this point, we also report data that allow the evaluation of the benefits from the VAB message aggregation technique discussed in Section 4.2 and the effectiveness of the load balancing policy presented in Section 4.3. To evaluate how VAB impacts the communication cost while increasing the amount of processors,

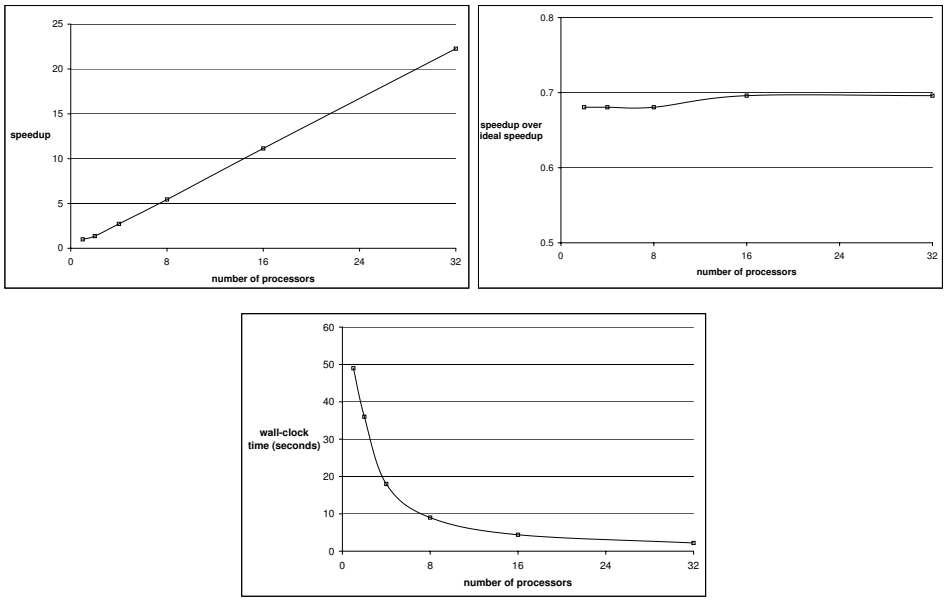


FIGURE 15. Speedup and Wall-Clock Time results for DD4.

we report the product between the average number of application messages delivered through a single MPI message, i.e. the average size of the aggregate which we will refer to as AAS, and the number of used processors. This product is representative of the system capacity to send application messages at the time cost of sending a single MPI message. Specifically, when using n processors, the hosted processes can perform send operations of MPI messages concurrently. Therefore, within the wall-clock time of a single send operation, we are, on the average, able to send n MPI messages in parallel. As a consequence, if the product between AAS and the number of processors increases, we have that the time cost for the send of each application message gets reduced, with consequent reduction of the overall communication overhead on each processor. (For completeness, we also report the plot for AAS, so as to show its behavior while increasing the number of processors.) For what concerns load distribution, we report plots related to the variation, over time, of the amount of unprocessed application messages, namely *upm*, stored in the *incoming* buffer at different processes (recall that *upm* has been used in Section 4.3 as the information on current load on each processor to determine the distribution of new nodes dynamically originated during the computation). This parameter is representative of the effectiveness of the load balancing strategy we have adopted since it provides indications on whether the work list keeping track of the amount of edges to be composed is approximatively the same on all processes at any time during the execution.

5.2. **Results for DD4.** The experimental measures obtained for DD4 are reported in Figures 15, 16 and 17. By the plots in Figure 15, we observe that the speedup curve remains linear over the whole interval for what concerns the number of user processors (i.e. up to 32); also the speedup value is constantly on the order of the

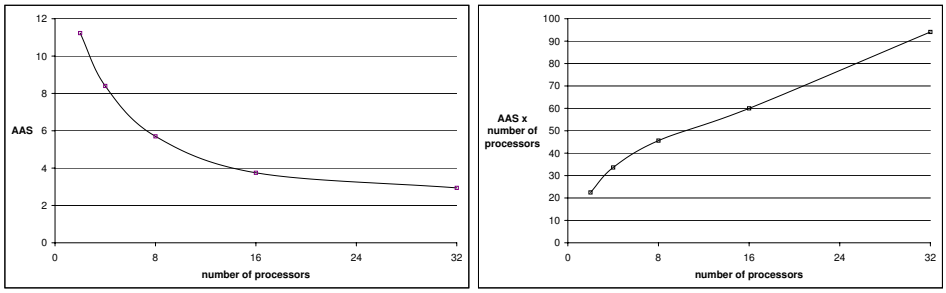


FIGURE 16. VAB message aggregation results for DD4.

70% of the ideal speedup. Combined toghether, these two plots indicate that the parallel implementation is effective for what concerns both the structuring of the parallel algorithm (this provides the ability to reach high values with respect to the ideal speedup) and the capability of remaining performance efficient while increasing the amount of used processors. Also, the wall-clock time curve, always reported in Figure 15, demonstrates that the speedup plots provide reliable performance indications, in the sense that speedup has been evaluated over an adequate interval for what concerns the amount of used processors. Specifically, with 32 processors, the wall-clock time for the computation gets on the order of 2.2 seconds, which is not only a definitely reduced value as compared to the sequential execution time (i.e. the execution time on a single processor, namely about 50 seconds), but also represents a satisfactory response time for an interactive end-user.

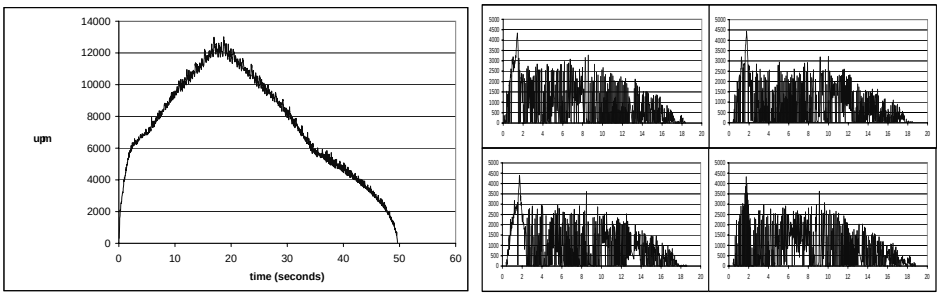


FIGURE 17. Variation of *upm* over time for DD4 (single processor case on the left - four processors case on the right).

The plots related to the behavior of the VAB aggregation technique in Figure 16 and to the variation of *upm* over time in Figure 17 additionally help understanding the reason why the implementation remains effective while increasing the amount of used processors. The AAS curve in Figure 16 shows that the average amount of application messages aggregated within each MPI message gets reduced from about 11.5 to about 3 while moving from the single processor execution to the execution on 32 processors. This is an expected behavior when thinking that a larger amount of used processors means that each process P_i involved in the parallel execution needs to manage an increased amount of channels towards other processes. As a consequence the application messages produced by P_i must be distributed over a

larger amount of aggregation buffers $out\_buff_{i,j}$, which means a reduced capacity to aggregate messages within a given time unit in each single buffer. However, observing the curve related to AAS multiplied by the number of used processors, always in Figure 16, we have a clear indication that the system capacity to send application messages at a given cost linearly increases with the number of processors, with a slope of about 0.8 (recall the ideal case for the curve of AAS multiplied by the amount of processors would be for slope equal to 1). Specifically, when moving from k processors to $2k$ processors, the system increases its capacity of sending application messages at the same time cost of about 1.6 times, which is a clear indication that VAB allows the communication overhead to scale well vs the size of the underlying computing system. For what concerns the variation of *upm* over time, in Figure 17 we report both the case of single processor execution and the case of execution on four processors. By the plots we observe that the load of unprocessed application messages, stored by each process P_i within the $incoming_i$ buffer, is well distributed on each of the four processors during the whole execution period, thus supporting the claim of the effectiveness of the load balancing mechanism described in Section 4.3.

5.3. **Results for `EXP3`.** The data obtained for the `EXP3` application allow, in the light of the already observed results for `DD4`, the determination of additional information on the run time behavior of our implementation, also in terms of the effects of the particular application on the achievable performance. The main difference with respect to the case of `DD4`, is in that this time the speedup curve does not remain linear vs the number of processors. Specifically, the plots in Figure 18 show that the speedup asymptotically tends to a constant value (on the order of 12), with a consequent decrease of the ratio over the ideal speedup. By the data related to the effects of VAB and to the variation of *upm* over time, it can be deduced that the cause for such a behavior is not due to ineffectiveness of the parallel implementation (e.g. in terms of increase of the communication overhead while increasing the amount of processors or load imbalance). Specifically, the curve in Figure 19 related to AAS multiplied by the number of used processors clearly shows that also in this case the implementation is able to carefully control the communication overhead while the size of the underlying computing system gets increased. More precisely, the linearity of such a curve, with slope on the order of about 1, i.e. the ideal slope value, provides indication that the implementation is able to control the communication overhead even in a more effective way that what done for the case of `DD4` in the previous section (recall the slope for the same curve for `DD4` was 0.8, thus lower than what we observe in this case). Also, the plots for *upm* in Figure 20 show that load remains balanced while moving from the single processor case to the execution on multiple processors.

Actually, that type of behavior for the speedup curve is due to the fact that `EXP3` exhibits a final computation phase made of a very limited amount (i.e. few units) of unprocessed application messages. This can be clearly observed when looking at the plots in Figure 20 related to the variation of *upm* over time. In other words, during that final phase the computation becomes intrinsic sequential (few unprocessed application messages produce, once extracted from the *incoming* buffer and composed through HC, few new application messages carrying new edges for the virtual net). As a consequence, during the final phase, parallelism cannot be exploited for `EXP3`, which is exactly the reason why speedup asymptotically tends

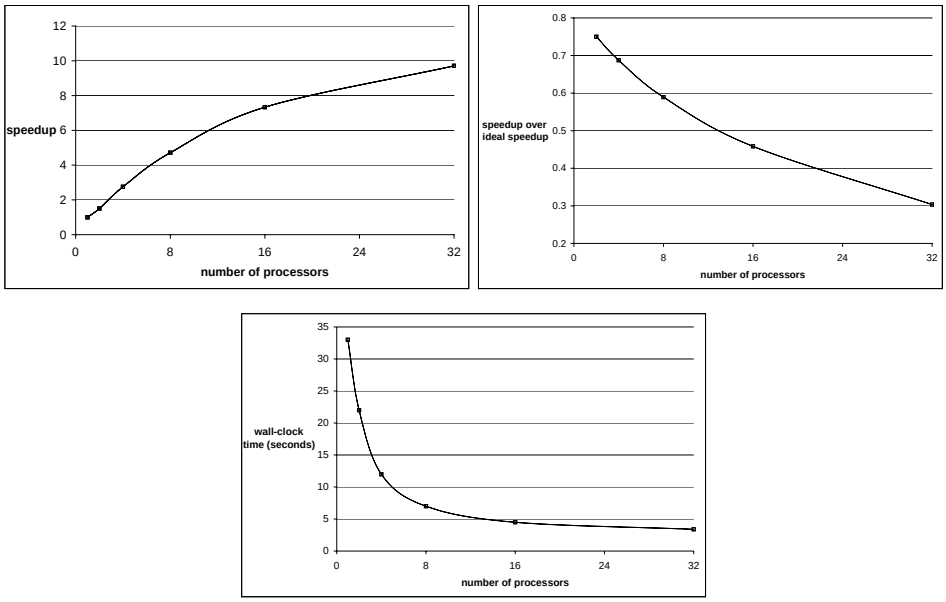


FIGURE 18. Speedup and Wall-Clock Time results for `EXP3`.

to a constant value. This type of problem has been already observed by Mackie in [Mac97] for the case of parallelism in the form of adequate assignment of the initial nodes of the interaction net over the used processors (recall this solution is based on a static analysis of the initial interaction nets and differs form our proposal in that we dynamically control load distribution, and other performance indexes at run time). Specifically, also for Mackie's approach, the benefits from the exploitation of multiple processors in the computing system are bounded by phases of sequential computation, if any, intrinsic to the specific application. However, as an additional support to the fact that our implementation is definitely able to exploit parallelism, whenever present within specific execution phases, we note that in case the speedup results were computed by excluding the final sequential phase (by the plots in Figure 20 such a phase lasts about three seconds, which is the reason why the wall-clock time asymptotically tends to three seconds while increasing the number of used processors), they would be even better that those obtained for the case of `DD4` in the previous section. Specifically, speedup would be on the order of at least 75% of the ideal over the whole interval for the amount of used processors.

6. SUMMARY

The definition of a formal system in [DR93, DPR97] for the computation of the execution formula [Gir89] of the λ-calculus terms constitutes the original starting point for the work we have presented in this article. Such a formal system was initially motivated as the mathematical settlement of an operational semantics for λ-calculus and for other functional programming languages. In making the precise definition of this system, properties of locality and asynchrony pointed out the potential for distributing the execution of programs.

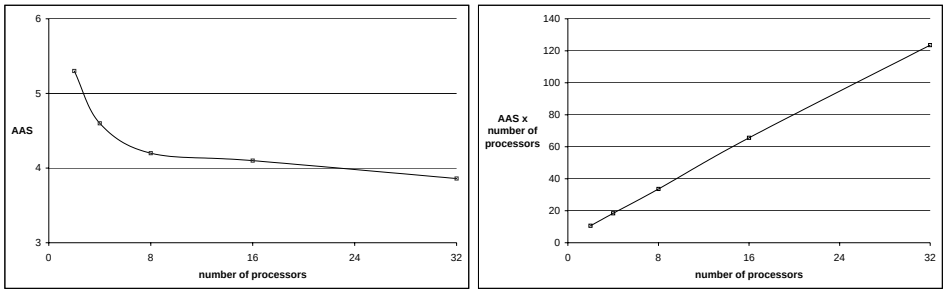


FIGURE 19. VAB message aggregation results for `EXP3`.

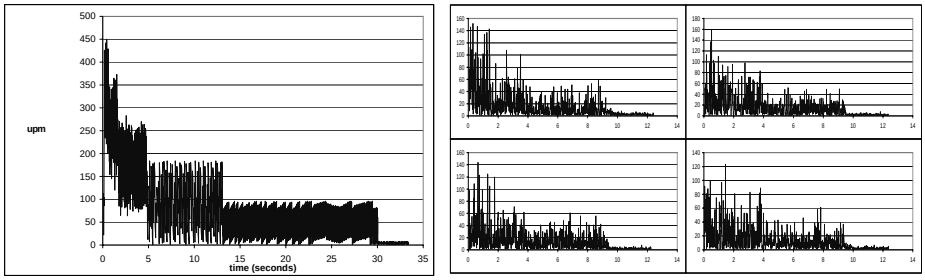


FIGURE 20. Variation of *upm* over time for `EXP3` (single processor case on the left - four processors case on the right).

The main contribution of this work relies on showing how it is possible to make a functional language, based on λ-calculus, transparently executable on a parallel/distributed environment. This result has been achieved by exploiting the decomposition of beta-reduction into a set of more elementary execution steps, each one independent of the others, which gives an execution model extremely flexible and very prone to be supported by a parallel/distributed environment. Specifically, we exploited the properties of locality and asynchrony of directed virtual reduction, namely the formal system providing the previously mentioned decomposition to develop the PELCR software package. This package manages the distribution of computational load due to the evaluation of a λ-term in a totally transparent way, by dynamically controlling/tuning any parameter potentially affecting the efficiency of the run-time behavior.

Our presentation integrates a solid theoretical background with many practical techniques coming from current parallel computing developments and provides a full featuring facility for the parallel/distributed execution of functional programs.

REFERENCES

[AC97] A. Asperti and J. Chroboczek. Safe operators: brackets closed forever: optimizing optimal λ-calculus implementations. *Appl. Algebra Engrg. Comm. Comput.*, 8(6):437–468, 1997.

[AG98] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[CAGRW98] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey. Optimizing communication in Time-Warp simulators. In *Proceedings of the 12th ACM/IEEE/SCS Workshop, Banff, Alberta, (Canada), May 1998*, Parallel and Distributed Simulation, pages 64–71. IEEE Computer Society Press, 1998.

[DNRD96] P. M. Dickens, D. Nicol, P. F. Reynolds, and J.M. Duva. Analysis of bounded Time Warp and a comparison with YAWNS. *ACM Trans. on Modeling and Computer Simulation*, 6(4):297–320, 1996.

[DPR97] V. Danos, M. Pedicini, and L. Regnier. Directed virtual reductions. In M. Bezem D. van Dalen, editor, *Computer Science Logic, 10th International Workshop, CSL '96*, volume 1258 of *Lecture Notes in Computer Science*, pages 76–88. EACSL, Springer Verlag, 1997.

[DR93] V. Danos and L. Regnier. Local and asynchronous beta-reduction (an analysis of Girard's EX-formula). In *Logic in Computer Science*, pages 296–306. IEEE Computer Society Press, 1993. Proceedings of the Eight Annual Symposium on Logic in Computer Science, Montreal, 1993.

[GAL92] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Principles of Programming Languages*, pages 15–26. ACM Press, New York, 1992. Proceedings of the 19th Annual ACM Symposium, Albuquerque, New Mexico, January 1992.

[Gir89] J.-Y. Girard. Geometry of interaction 1: Interpretation of system F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, pages 221–260. North-Holland, 1989.

[Gir95] J.-Y. Girard. Light linear logic. In Daniel Leivant, editor, *Proceedings of the International Workshop on Logic and Computational Complexity (LCC'94)*, volume 960 of *LNCS*, pages 145–176, Berlin, GER, October 1995. Springer.

[Laf90] Y. Lafont. Interaction nets. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 95–108. Association for Computing Machinery, 1990. Papers presented at the Symposium, San Francisco, California, January 17-19, 1990.

[Lam90] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. of 17th Annual ACM Symposyum on Principles of Programming Languages*, pages 16–30, San Francisco, California, January 1990. ACM.

[Lév78] J.-J. Lévy. *Lambda-Calcul Etiqueté*. PhD thesis, Université Paris VII, 1978.

[Mac97] I. Mackie. Static analysis of interaction nets for distributed implementations. In P. van Hentenryck, editor, *Proceedings of the 4th International Static Analysis Symposium (SAS'97)*, number 1302 in Lecture Notes in Computer Science, pages 217–231. Springer-Verlag, September 1997.

[Ped96] M. Pedicini. Remarks on elementary linear logic. In *A Special Issue on the "Linear Logic 96, Tokyo Meeting"*, volume 3 of *Electronic Notes in Theoretical Computer Science*, Amsterdam, The Netherlands, 1996. Elsevier.

[Pet84] M. Petrich. *Inverse Semigroups*. Pure and Applied Mathematics. John Wiley & sons, 1984.

[XH96] Z. Xu and K. Hwang. Modeling communication overhead: MPI and MPL performance on the IBM SP. *IEEE Parallel & Distributed Technology*, 4(1):9–24, 1996.