A Parallel Implementation for Optimal Lambda-Calculus Reduction

Marco Pedicini
Istituto per le Applicazioni del Calcolo
Consiglio Nazionale delle Ricerche
Viale del Policlinico 137, 00161 Roma, Italy
marco@iac.rm.cnr.it

Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Viale Salaria 113, 00198 Roma, Italy
quaglia@di.uniroma1.it

ABSTRACT

In this paper we present a parallel implementation of Lévy’s optimal reduction for the λ-calculus [11]. In a similar approach to Lampéng’s one in [10], we base our work on a graph reduction technique known as directed virtual reduction [3] which is actually a restriction of Danos-Regnier virtual reduction [4].

The parallel implementation relies on a strategy for directed virtual reduction, namely half combustion, which we introduce in this paper. We embed in the implementation both a message aggregation technique, allowing a reduction of the communication overhead, and a fair policy for distributing dynamically originated load among processors. The aggregation technique is mandatory as the granularity of the computation is fine. Through this technique we obtain a linear speedup close to 80% of the ideal one on a shared memory multiprocessor. This result points out the viability of parallel implementations for optimal reduction.

Categories and Subject Descriptors

F.4.1 [Theory of Computation]: Mathematical Logic and Formal Languages—Lambda calculus and related systems.

General Terms

Languages, Theory, Algorithms.

Keywords


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 [11].

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 [10]; the outcome reduction technique introduced by J. Lampéng, known as sharing reduction, relies on a set of graph rewriting rules.

In [8], Lampéng’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 [6]. This result spurred the research in the field of optimal reduction.

Specifically, in [4] 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 [3], by the introduction of a new graph rewriting technique known as directed virtual reduction (DVR). The authors defined also a strategy to perform DVR, namely combustion, which simplifies the calculus and can simulate individual steps of sharing reduction.

In this paper 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 ([9]) and deriving from the fine decomposition of the λ-calculus β-rule obtained through the analysis provided by the Geometry of Interaction. Specifically, we present a parallel implementation for optimal reduction 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 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 [12] 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 [12] 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 [8]) define an interaction system ([9]); however, contrarily to our work, it does not focus on optimal reduction. Another important 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.

Specifically, while designing our implementation we had to tackle, among others, the following basic issues:

(a) the calculus performed in DVR has fine granularity; therefore, the rate of message exchange must be kept low in order to get an efficient parallel execution;

(b) anytime a computational step of the HC strategy is performed, possibly new edges and nodes are added to the directed virtual net, which represent new computational load; the big challenge in this issue is how to keep the load balanced during the whole lifetime of the parallel execution.

Issue in point (a) has been tackled by using a message aggregation technique which collects application messages with the same recipient process and delivers them using a single physical message. The advantage is a reduction of the number of physical messages, which leads to an increase of the real granularity of the computation. Note, however, that arbitrarily delaying the delivery of application messages may harm performance of the recipient process \( P \) as it is possible that \( P \) is actually idle and waiting for new load to come in. To overcome this drawback, we implement a control mechanism to tune on-line the aggregation parameters in order to get a good balance between the gain from aggregation and the potential harm in delaying the messages.

To tackle issue in point (b), our implementation uses state information related to the load condition to support a fair policy for distributing new load originated by the HC strategy. To spread the knowledge of load conditions we do not use additional physical messages, but a classical piggy-backing technique (i.e. information is attached to application messages). This produces negligible increase in the communication overhead.

We have measured the performance of our implementation on the shared memory multiprocessor Sun Ultra Enterprise 4000. The obtained data show that the speed up achieved is linear and is about 80% of the ideal one for any value of the number of used processors and for any considered benchmark. This result points out the viability of parallel implementations for optimal reduction, which was previously an open issue.

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 what may directly affect the comprehension of the parallel dynamics. The remainder of the paper 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 parallel implementation. The performance data are reported in Section 5.

Acknowledgments. The project of a parallel and optimal interpreter for \( \lambda \)-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/CNR - Bilateral Project n.3132 - 1996/97); some of the ideas used in our implementation arose thanks to discussions of the first author with V. Danos.

2. FROM LAMBDA TERMS TO DIRECTED VIRTUAL REDUCTION

As pointed out in the introduction we deal with a distributed interpreter for lambda-terms based on DVR. For the lack of space it would be difficult to formally recall all the steps that led from the pioneering ideas contained in Lévy’s work on optimal reduction, to the outstanding mathematical bases given by Girard’s “Geometry of Interaction”. 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 [1].

Here we shortly recall VR and the Geometry of Interaction; then we will give a full presentation of DVR. 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.

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 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 \( G_T \). Any node \( p_i \) in the border is either associated with one free variable \( x_i \) or, if \( i = 0 \), \( p_0 \) represents the root of the term. A straight path in a directed graph is a path that never bounces back in the same edge.

The execution formula of \( G_T \) is:

\[
EX(R_T) = \sum_{\phi_{ij} \in P(R_T)} W(\phi_{ij})
\]

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

Danos and Regnier's construction of a confluent, local and asynchronous reduction of lambda-calculus, derived from a semantic setting based on only one move (simple enough to be easily mechanized), was the preliminary step for our work. 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 1. 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.

The algebraic mechanism corresponding to the rest is called the bar; it was introduced in [4] to ensure the preservation of Girard’s execution formula. Note that VR induces bars of bars by definition; this is shown in Figure 2.

DVR, presented in [3], was designed in order to avoid bars of bars, thus allowing any implementation to use simple
data structures for representing edges. DVR is discussed in Section 2.2.

2.1 Geometry of Interaction

The material in this section is identical with similar material in [3]. 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 a parallel calculus. In order to get a computational device from this graphical calculus we should introduce a suitable strategy: by means of the 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 of as the set of partial one-one maps $u$ with composition. The structure is then enriched with partial inverses $u^*$, the codomain operation $[u]$, and the complementary of the codomain $[u]$. Axioms for such a structure are formally introduced below.

**Definition 1.** An inverse monoid (see [13]), or for short an im, is a monoid with an unary function, the star, denoted by $[.]^*$, with

$$
(uv)^* = v^* u^*
$$

$$
(u^*)^* = u
$$

$$
 uu^* u = u
$$

$$
 uu^* v^* = v^* u u^*
$$

We denote by $[u]$ the idempotent $uu^*$. With this notation the last equation becomes $[u] [v] = [v] [u]$ and the one before becomes $[u] u = u$.

**Definition 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

$$
 [1] = 0 \quad \text{and} \quad [0] = 1
$$

$$
 u[0] = [u] 0 = [v] u
$$

Bim’s axioms entail $[u][a] = [u]$, $[a]^* = [a]$ so $[u]$ is an idempotent, and $[u] 0 = 0$, $0 u = 0$ if $v [u] v = v$.

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

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

$$s^* = \sum_i n_i s_i^*, \quad [s] = 1 - [s] = 1 - ss^*.
$$

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

**Proposition 1.** $[S]$ is an inverse monoid with $(.)^*$ defined as above.

The proof is a straightforward calculation.

**Definition 3.** Define the bar closure of $S$, denoted by $[S]$, to be the im obtained by $\omega$ iterations of the complementary closure, that is: let $S_0 = S$ and $S_{i+1} = S_i$ then $[S] = \bigcup_{n \geq 0} S_n$.

**Proposition 2.** $[S] = \omega$ is an inverse monoid with $[.]$ defined as above.

**Definition 4.** The monoid $L^*$ of the Geometry of Interaction is the free monoid with a morphism $! \langle \cdot \rangle$, an involution $(.)^*$ and a zero, generated by $p, q$, and a family $W = \{w_i\}$ of exponential generators such that for any $u \in L^*

$$x^* y = \delta_{xy} \quad \text{for} \quad x, y = p, q, w_i,
$$

$$!(u)w_i = w_i^*!(u)
$$

where $w_i$ 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 $w_i \in \langle i \rangle$ to explicitly note the lift of the generator.

Equations (7) will be called of annihilation and (8) are called equations of swapping.

Orienting the equations (7-8) 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 $ab^*$ where $a$ and $b$ are positive (i.e. written without $^*$). The fact that all non-zero terminals are equal to such an $ab^*$ form is referred to as the “ab property”. From this, one easily gets that the word problem is decidable and that $L^*$ is an inverse monoid.

Every computation, from now on, will take place in the bar closure of $L^*$ in $Z[L^*]$, which we denote by $[L^*]$. Since, as said, this is a bim, results in $[4]$, which were stated and proved for any bar inverse monoid, apply with no further ado. Note that equalities in $[L^*]$, and in $Z[L^*]$ are also decidable by rewriting to stable form.

Set $h_1, \ldots, h_n$ is an idempotent iff the $b_i$’s are orthogonal that is, $[b_i] [b_j] = 0$. 

![Figure 1: composition performed by VR](image1.png)

![Figure 2: VR originating bars of bars](image2.png)
Lemma 1 (Superposition). Let $a$, $b$ and $c$ be positive monomials in $L^*$ such that $(a)(b)$, $(b)(c)$ and $(a)(c) \neq 0$, then $(a)(b)(c) \neq 0$.

Proof. See [3]. □

Definition 5. Let a weight on a directed graph be a function $W$ from the directed graph's involutive category of paths to $[L^*]$.

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

We will say that $\alpha$ coincides with $\beta$ or equivalently that $\alpha$ and $\beta$ are coincident if they have the same target node.

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

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

Definition 6. A straight path is a path that contains no sub-path of the form $\phi \phi^*$, 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 $\phi_1$, $\phi_2$ and $\phi_3$ are such that $(\phi_1)(\phi_2)(\phi_3) = 0$; it is said to be square-free if for any straight path $\phi$, $\phi \phi = 0$.

Definition 7. 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 $\phi_1$, $\phi_2$ and $\phi_3$ such that none is prefix of another are such that

$$(\phi_1)(\phi_2)(\phi_3) = 0.$$

2.2 Directed Virtual Reduction

Definition 8. A directed virtual net $R$ is an acyclic virtual net such that for each edge $\alpha$:

A. $\alpha = [b_1, \ldots, b_n]a$, where $a$, $b_1$, $\ldots$, $b_n$ are positive monomials of $L^*$. We will denote $b_0$ the weight of $\alpha$ 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 $\beta_1$, $\beta_2$ of $\alpha$ along $\tau_1$, $\tau_2$

\[
(\beta_1)(\beta_2) \neq 0 \quad 0^{\beta_1}(a; b_1; b_j) \\
(\beta_1)(\tau_1^*\beta_2^*) = 0 \quad 1^{\beta_1}(a; b_1; \tau_1^*\beta_1) \\
(\tau_1^*\beta_2^*)(\tau_2^*\beta_2^*) = 0 \quad 2^{\beta_1}(a; \tau_1^*\beta_1; \tau_2^*\beta_2)
\]

Given two coincident counter-edges $\alpha$ and $\beta$, with weights $[b_1, \ldots, b_n]a$ and $[a_1, \ldots, a_n]b$, then DVR generates a new node and two new edges linking that node to the sources of $\alpha$ and $\beta$. These new edges have, respectively, weights $b'$ and $a'$ where $a'b'$ is the stable form of $b'a$; this is shown in Figure 3.

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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{construction performed by DVR}
\end{figure}

Definition 9. Given two composable edges $\alpha$ and $\beta$, the two edges $\alpha'$ and $\beta'$ generated by one step of DVR are called residuals of $\alpha$ and $\beta$ respectively. We will denote these residuals by

$$dvr(\alpha, \beta) = (\beta', \alpha').$$

Lemma 2 (Augmentation). Let $R$ be a directed virtual net and $\gamma_2$ be a counter-edge of $\gamma_1$ along $\tau$ in $R$, then

$$(\gamma_1)(\tau^* \gamma_2) = (\gamma_1^*)(\tau^* \gamma_2^*) \neq 0,$$

or equivalently

$$\gamma_1 \gamma^* \gamma_2 = \gamma_1^* \gamma^* \gamma_2^* \neq 0.$$

Proof. See [3]. □

In [3], it has been proved that DVR is sound w.r.t. Girard's execution formula:

Proposition 3 (Invariance). The execution formula EX is an invariant of DVR.

Definition 10. Given a directed virtual net $R$, a total edge $\alpha$ is any edge with at most one counter-edge $\beta$ that is, $\beta$ is the only edge such that

$$\{\alpha\}(\tau_{\beta^*}^*) \neq 0.$$

In this case we say that $\alpha$ is total w.r.t. $\beta$; if $\alpha$ has no counter edge, it is called ghost.

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

Proposition 4. Given two composable edges $\alpha$ and $\beta$ such that $dvr(\alpha, \beta) = (1, \alpha')$, then $\alpha$ is total w.r.t. $\beta$.

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

If $\tau$ is not the empty path, we can apply the same argument to the residual $\gamma'$ of the reduction sequence along the directed path $\tau$, and derive the property by lemma 2 applied to $\alpha$ and $\gamma$. □
Proposition 5. Given three coincident edges $\alpha, \beta, \gamma$ such that

$$dvr(\alpha, \beta) = (\beta', 1)$$

with $\alpha$ and $\gamma$ composable, then the residual of $\gamma$ is not composable with $\beta$.

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

Corollary 1. (Soundness of the optimization of one) If $dvr(\alpha, \beta) = (1, \alpha')$, then no further composable edge $\gamma$ can give 1 as residual of the composition with $\alpha$.

Proof. The two residuals in the source of $\alpha$ have weight 1 so that they are composable and this is a contradiction with the proposition 5. □

This corollary allows an optimization rule, in fact, the configuration produced by the DVR step $dvr(\alpha, \beta) = (1, \alpha')$ 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 $\alpha'$, so we propose to transform this configuration by removing the edge $\beta$ with weight 1 and using the edge $\alpha'$ for linking the target of $\beta$ and the target of $\alpha'$ (see Figure 4).

![Figure 4: an optimization rule](image)

2.3 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 unoriented graphs built from the indexed nodes represented in Figure 5. These nodes are called sharing operators and distinguished in two groups. The first group includes the operators in the original Lamp ing's work: application, and abstraction; the second one is constituted by a family of nodes of the same kind (the so called muxes) in accord to the following definition:

Definition 11. 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:

- If $y_k$ is a port of an application
- If $y_1, \ldots, y_n$ is translated into a directed virtual net in the following way

![Figure 5: sharing graph operators](image)

Definition 12. 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

When the nodes introduced by the translation present the configuration described in the next definition they are reduced by amalgamating edges as in Figure 6.

Definition 13. A node with $n$ coincident edges $\alpha_1, \ldots, \alpha_n$ and an edge $\beta$ with source the target of the $\alpha_i$'s is erased and all the $\alpha_i$'s are replaced by edges $\alpha'_i$ where the source of $\alpha'_i$ is the source of $\alpha_i$, the target of $\alpha'_i$ is the target of $\beta$ and the weight of $\alpha'_i$ is $\alpha_i \beta$.

With the help of an example we show how to change a lambda term into a directed virtual net. In Figure 7, starting from the syntactic graph of the $\lambda$-term representing the Church numeral 2 applied to the identity that is, by using Krivine's notation, $(\lambda f \lambda x(f)(f)x)\lambda x$, we obtain a sharing graph...
graph by adding the control operators, expressed in the multiplexer syntax, and annotating each node by level indices. Then edges are oriented, unfolded and labeled with monomials. In this way, 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.

3. HALF COMBUSTION STRATEGY

In [3], 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 10) 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. This may originate inefficiency in the case few processors host nodes of valence zero at anytime.

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 9. In this example, the coincident counter-edges of edges with weight \([a_1, \ldots, a_m]b_i\) are among those edges weighted with \(a_1, a_2, \ldots, a_m\).

Below we give the definition and provide the soundness of the HC strategy.
consequence, all the coincident filtered edges (including

positive weight, see Figure 11.

weight in a semifull directed virtual net are not composable with each other. Thus the obtained

directed virtual net is semifull. Therefore, no

the set of non-positive edges composable with \( \beta \) can possibly be empty, in this case

HC just composes \( \alpha \) and \( \beta \).

Definition 14. Given a composable edge \( \alpha \) with positive

weight in a semifull directed virtual net \( R \), we have to con-

sider two cases:

1. if the set \( \{ \beta_1, \ldots, \beta_n \} \) of non-positive edges composable

with \( \alpha \) is non-empty then the half combustion strategy

(HC) performs all the possible compositions of \( \alpha \) with

the \( \beta_i \).

2. if \( \alpha \) has a positive coincident counter-edge \( \beta \) and no

non-positive one, then HC performs the composition of \( \beta \) with \( \alpha \) and possibly with every non-positive edge

composable with \( \beta \).

Proposition 6. 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 \( \alpha \) having positive weight \( a \) as in

the definition 14, and suppose that all the composable

edges with non-positive weights coincident with \( \alpha \) are the

\( \beta_i \)'s with weights \( [a_{1i}, \ldots, a_{ji}] \beta_i \) for \( i = 1, \ldots, n \) as in Figure 10.

If we apply a step of the HC strategy by performing a

DVR step between \( \alpha \) and \( \beta_i \) for \( 1 \leq i \leq n \), we obtain

\[ \text{dvr}(\alpha, \beta_i) = (\beta_i', \alpha_i') \]

where the weight of \( \alpha \) is \( [b_1, \ldots, b_n] \alpha \) and the weight of \( \beta_i \)

is \( [a_{1i}, \ldots, a_{ji}, a_i] \beta_i \) and the two new edges \( \beta_i' \) and \( \alpha_i' \) have a

positive weight, see Figure 11.

Therefore, now the set of the coincident filtered edges has been enlarged with \( \alpha \), but \( \alpha \) is no more composable with

the \( \beta_i \)'s because of its filter and all the generated edges \( \alpha_i \)'s and

\( \beta_i \)'s have positive weights by definition of DVR. As a

consequence, all the coincident filtered edges (including \( \alpha \))

are not composable with each other. Thus the obtained
directed virtual net is semifull.

Figure 10: edges in a semi-full node

Figure 11: edges after the composition performed by HC

We recall that the translation presented in Section 2.3

associates with any \( \lambda \)-term a full directed virtual net (see also [4, 8]). As full nets are particular instances of semifull

ones, HC actually represents a reduction mechanism for \( \lambda \)

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 com-
busted. When created, edges are marked as incoming. One

step of reduction consists of picking an incoming edge \( \alpha \) and

performing all the compositions with coincident combusted

edges. Then \( \alpha \) 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

dges coincident with \( \alpha \) are not composable with \( \alpha \), with

the particular case where the set of combusted edges coincident with \( \alpha \) is empty as in case 2 of the definition. On

the other hand, at any step any incoming edge has a pos-

itive weight. As an edge is marked combusted only after

having been (successfully or not) composed with every coinci-
dent combusted edges, one easily sees that two combusted

dges are never composable. Thus this suggests that we can

organize the computation in such a way that the only mean-
ging associated with filters is about the belonging of an edge
to the first or to the second set (thus, like in the combus-
tion strategy, filters can be actually discarded). We have

embedded this simplification in the parallel implementation

presented in the next section.

4. PARALLEL IMPLEMENTATION

4.1 Data Structures and Code Organization

Each processor \( i \) of the parallel architecture runs a pro-

cess \( P_i \) which is an instance of the executable code associated

with the parallel algorithm. 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 \( (id(v_1), id(v_2), W(e)) \).

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 12 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 when describing the load balancing module that establishes how new nodes must be distributed among processes.

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 timestamp value assigned by \( P_i \). The timestamp 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 \( P_i \) and \( P_j \) 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 edge \( e_3 \) 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 will allow us to keep low the communication overhead due the send and receive of notification messages.

Applying the delayed creation technique to the example in Figure 12 means that node \( v_1 \) is created by \( P_h \) 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 \( P_h \)). 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 \( P_h \)).

By previous considerations we get that any message exchanged between two processes carries exclusively the information of a new edge. As each node is identified by the previously mentioned triple, a message carrying the information associated with the edge \( e(v_1, v_2) \) has a payload consisting of the triple \( [id(v_1), id(v_2), W(e)] \) where \( id(v_1) = [t, P_i, P_j], id(v_2) = [t, P_i, P_j] \) and \( W(e) \) is the weight of \( e \).

\( P_i \) 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 paper 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 \( nodes_i(v).\text{combined} \), containing the edges incident on the node \( v \) which have already been composed (i.e. the combust edges of the HC strategy).

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, \( P_i \) is such that its list \( nodes_i \) is empty but its buffer \( incoming_i \) 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 \( P_i \) 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 13 we show the high level structure of our code. Before entering the 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 \( m.\text{target} \) the node identified by the information \( id(v_2) \) carried by \( m \) (recall that \( id(v_2) \) is the previously described triple) and as \( m.\text{source} \) the node identified by the information \( id(v_1) \) carried on the same message. \( e.\text{target} \) and \( e.\text{source} \) have similar meaning when referring to an edge \( e \).

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 the negative case, it extracts the first message from the buffer incoming$_i$ and performs the composition of the corresponding incoming edge.

```
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 $\in$ nodes$_i$, 'node already in the local list')
7                 then
8                     for each edge e $\in$ nodes$_i$,(m.target).com busted do
9                         (compose the edge carried by m with e);
10                        (select the destination process P$_j$ for hosting
11                          the node originated by the composition);
12                        (send the edges produced by the composition to
13                          P$_j$ and P$_k$ hosting m.source and e.source
14                          respectively)
15                 endfor
16             else (add m.target to nodes$_i$); 'delayed creation'
17                 add the edge carried by m to
18                 nodes$_i$,(m.target).com busted
19             endwhile;
20       (end_computation = check_termination());
21   endwhile
```

![Figure 13: pseudo code for process P$_i$](image)

The structure of the code in Figure 13, 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 [5], probe functions may be prohibitively expensive (for example because they require context switch to the kernel), 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, 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 13 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. Finally we will point out some other aspects related to the code structure.

### 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 $s \times oh$, where $s$ is the size (in bytes) of the message and $oh$ is the send/receive time per byte. $oh$ typically includes the context switch to the kernel, the time to pack/unpack the message and the time to setup the physical network path. Instead, $oh$ takes into account any cost that scales with the size of the message.

$oh$ is usually higher than $oh$, 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 will allow 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 $3oh + 3s \times oh$ to $oh + 3s \times oh$.

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 $outbuff_{ij}$. 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 $outbuff_{ij}$ by periodically incrementing a local counter $c_{ij}$. The value of $c_{ij}$ 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$_i$, $c_{ij}$ is increased by one if at least one message is currently stored in the aggregation buffer $outbuff_{ij}$. 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 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 [2] for communication modules supporting fine grain par-
allel discrete event simulations. In VAB, anytime the mes-
gages aggregated in $out_{buff}$ 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 af-
fter 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 that buffer is increased by one as many
messages are supposed to arrive in the aggregate in short
time and this will increase AG. If the arrival rate falls below
$est_{i,j}$ then the maximum age is decrease 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 ap-
messages $upm$ stored in the buffer $incoming$, 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). $UPM_i[k]$ records the current value of the number of unprocessed application messages of $P_j$. $UPM_i[j]$ 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 $UPM_i[j]$ is piggybacked on the message, denoted $M.UPM_i$. Whenever a physical message $M$ sent by $P_j$ to $P_i$ is received from $P_j$, $UPM_i[j]$ is updated from $M.UPM_i$ (i.e. $UPM_i[j] \leftarrow M.UPM_i$). The information on the load conditions kept by the $UPM$ vectors is approximated for two reasons:

1. There exists the possibility that when $P_i$ receives $M$
from $P_j$, the current value of $UPM_j[j]$ is different from $M.UPM_i$;

2. The current value of $UPM_i[j]$ is not an exact representa-
tion of the current load of $P_j$ as it does not count
application messages carried by pending physical mes-
sages; these application messages represent work to be
performed which has not yet been incorporated into
the buffer $incoming$.

We note, however, that obtaining more accurate state in-
formation 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 un-
acceptable negative effects on the performance. Anyway, it
is important to remark that the FIFO property for the com-
munication channels guarantees that each time a physical
message $M$ sent by $P_j$ is received from $P_j$, the piggy-backed
value $M.UPM$ refers to a more recent load condition as com-
pared to the one indicated by the current value of $UPM_j$.[j].

Based on the values stored in $UPM_i$, we have imple-
mented a selection policy for the destination process of a
new node which is a modified round-robin. It works as fol-
loows. $P_i$ 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 pro-
cess which should host the new node according to the
round-robin policy. $P_i$ actually selects $P_{rr_i}$ as destination
if $UPM_j[rr_i] < UPM_i[j]$; otherwise $P_i$ selects itself as des-

tination 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 Final Remarks

At the end of the computation we get that for all the
nodes of the final graph the incident edges are ghost. How-
ever 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 hav-
ing only ghost 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. Although not discussed in
previous sections for space limitation, we have embedded
in our implementation a technique to discard on-line nodes
that do not belong to the normal form. Basically this tech-
nique tracks whenever a node becomes of valence zero and
removes it if there is no directed path towards nodes of the
border. This technique implements a sort of garbage col-
lection which allows us to keep small the size of the list $node_s$, on the process $P_i$, thus allowing fast management of the list
itself.

For what concerns the termination detection, we have imple-
mented it through additional control messages. Any-
time 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 and already elaborated and
the number of application messages produced for other pro-
cesses (such message will be referred to as status mes-

cage). When the master $P_0$ detects that each process has
already elaborated all the application messages produced for it, \( P_i \) discovers that the computation is over and notifies the termination to the slave processes (through the send of a termination message). By looking at the structure of the code in Figure 13, 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` 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 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.

## 5. PERFORMANCE RESULTS

We report in this section some performance data in order to evaluate the efficiency of our parallel implementation. As already pointed out, we have realized the implementation by using MPI as message passing layer.

Before entering the description of speedup results, we recall that there exists an implementation from Asperti (the so-called BOHM, Bologna Higher Order Machine) which is actually a sequential machine for optimal reduction that has been demonstrated to provide better performance compared to non-optimal interpreters such as CAML and HASKELL (see \([1]\)). BOHM embeds an optimization, known as safe operators that allows the merging of many control operators in a compound one acting as the sequence, this may sometimes provide an exponential decrease of the number of interactions. A direct comparison between our code (run on a single processor of the Sun Ultra Enterprise 4000 shared memory multiprocessor) and the sequential BOHM machine shows that BOHM allows faster execution for applications where safe operators actually allow the merging of control operators. On the other hand the mechanism of safe operators is not useful when constraints like the ones introduced by Girard in systems with intrinsic complexity ELL (or LLL) are satisfied, see \([7]\); in these cases it never happens that operators can be merged following Asperti's rules. Anyway, we have investigated on the possibility to embed safe operators in our implementation and currently we have preliminary results on their compatibility with DVR.

To test the speedup achievable by running our interpreter on more than one processor of the Enterprise 4000 \(^3\) we used two different benchmarks, namely EXP3 and DD4.

EXP3 corresponds to the ELL term \( \text{Ite}(\text{Mult}2,1,\text{Ite}(\text{Mult}2,1,\text{Ite}(\text{Mult}2,1,4))) \), whose normal form represents the iterated exponential \( 2^{2^{2^{2^4}}} \). DD4 corresponds to the \( \lambda \)-term \( (\delta)(\delta)(\delta) \delta \) where \( \delta = \lambda x(x)x \) represents the self-application. The normal form of this term represents the Church's integer \( (4^4)^{4^4} \). For these two benchmarks, the shared result of the HC strategy has a number of nodes which is in the order of hundreds of thousands, therefore they are definitely large case studies to stress the behavior of our implementation.

In this section we report two series of plots. The first one (see Figure 14) relates to speedup as a function of the number of used processors (from 1 to 4). Specifically, this series contains speedup results for both the case of parallel execution with no message aggregation and for the case of parallel execution with VAB (the dynamic message aggregation technique based on the variable age of the aggregation buffer, described in Section 4.2). These results demonstrate that the parallel execution with load balancing but with no careful technique to bound and control the communication overhead could actually lead to limited performance gain. In particular, the observed speedup for both the benchmarks is limited to about 50% of the ideal speedup. This is not the case when VAB is used. Specifically, this dynamic technique allows speedup in the order of 80% of the ideal one. Nevertheless, the speedup curves remain linear for both the cases of no aggregation and VAB, thus demonstrating how our parallel algorithm structure does not adversely affect the performance when the size of the underlying architecture is increased at some extent. As a consequence, further reduction of the execution time should be likely to be obtained on an architecture with more than 4 processors (e.g. 6-8).

To complete the description of the results for the case of VAB, we report in Figure 15 the average number of application messages (ANAM) delivered through a single MPI message. These results demonstrate that the communication overhead is reduced of about 50% for the case of DD4 and up to 85% for the case of EXP3.

As regard to statistical significance of these results we have observed that running the same experiments several times leads to results that are within the 3% of each other.

![Figure 14: speedup vs the number of processors](image-url)

### 6. REFERENCES


Figure 15: average number of aggregated messages vs the number of processors


