Elimination of Redundant Messages with a Two-Pass Static Analysis Algorithm*

Alain Girault†

Abstract

Eliminating redundant messages in distributed programs leads to the reduction of communication overhead, and thus to the improvement of the overall performances of the distributed program. Therefore, a lot of work has been done recently to achieve this goal. In this paper, we present an algorithm for eliminating redundant messages in parallel programs that have been distributed automatically. This algorithm works on program whose control flow is as general as possible, i.e., contains gotos. Precisely, the control flow is a finite deterministic automaton with a DAG of actions in each state. Our algorithm proceeds in two passes: First a global data-flow analysis which computes, for each state of the automaton, the set of distant variables that are known at the beginning of the state. Then a local elimination which removes redundant messages locally in each state of the automaton. We present the algorithms as well as the corresponding time and memory costs.

Keywords

Redundant message elimination, static analysis, data-flow analysis, parallel program optimisation, distributed memory machine.

1 Introduction

1.1 Distributed Programs

Many computer systems have to be distributed on several computing locations with distributed memory, for various reasons: Increase in performance, location of sensors and actuators, fault tolerance, and so on. This is particularly the case for real-time critical computer systems where these objectives are emphasised. Classic examples of such systems are automatic pilots on aircrafts, nuclear plant controllers, and so on. Such systems are labelled critical because the consequences of a software error can be disastrous with respect to human lives, the environment, and the economy. This critical aspect calls for rigorous design methods.

Since designing a distributed program directly is always harder and error-prone, an approach that has arisen recently involves first programming a

---

* A short version of this article has been published in the Euromicro Workshop on Parallel and Distributed Processing, Mantova, Italy, February 2001.
† INRIA Rhône-Alpes, BIP project, 655 avenue de l’Europe, 38330 Montbonnot Saint-Martin, FRANCE, Tel: +33 476 61 53 51, Fax: +33 476 61 54 54, Email: Alain.Girault@inrialpes.fr
centralised system and then distributing it automatically. See, for instance, [8, 5, 18, 12], or [20] for a recent survey. The other advantage of this approach is the possibility to debug and formally verify the centralised program before its distribution, which is always easier and faster than debugging a distributed program.

In this article, we do not focus on how to obtain a distributed program automatically, but rather on a post-processing optimisation, namely redundant message elimination. Therefore our starting point is a distributed program.

Our model of a distributed program consists of a set of \( n \) finite deterministic automata, one for each computing location. The actions performed by an automaton are classic imperative instructions: variable assignment, arithmetic operations, input/output operations, expression testing, external procedure call, and so on. The set of variables manipulated by the distributed program is partitioned into \( n \) subsets, one for each computing location. Since the target architecture is a distributed memory one,

- each automaton maintains a local copy of all the distant variables, i.e., those belonging to the automaton of another location,
- each automaton can only modify its local variables (owner computes rule),
- and automata communicate by exchanging values through a fully connected network of FIFO queues.

Two operations allow the exchange of values through a FIFO queue:

- An automaton can send the value of one of its local variables to another automaton with a \textit{send} operation that is non-blocking.
- An automaton can receive a value and assign it to its local copy of the corresponding distant variable with a \textit{receive} operation that is blocking while the queue is empty.

It is the job of the distribution algorithm to make sure that \textit{send} and \textit{receive} operations match each other and are without deadlocks.

### 1.2 Redundant Messages

One drawback of automatically distributed programs is that they are rarely optimal with respect to messages, computations, and so on. Indeed, during the distribution process, it is not always possible to determine the list of distant variables that are already known by the automaton of a given location. As a result, redundant messages can occur. A message is said to be redundant when the variable exchanged is already known by the automaton of the destination location. We are particularly interested in eliminating redundant messages since they cause overhead in the execution of the distributed program, and therefore reduce the utility of automatic distribution [22].

In this paper, we propose an algorithm that computes the set of known distant variables for the automaton of each location and at the beginning of each action, and suppresses redundant messages accordingly. This problem is related, albeit it is more difficult to solve, to the data-flow problem of finding available expressions in a basic block. Here the difficulty arises for two reasons:
1. The program is distributed, so there is a separate data-flow problem for each computing location.

2. Local variables become available through assignment while distant variables become available through communication, so there are two types of the so-called $gen$ sets to deal with.

### 1.3 Comparison With Related Work

There have been numerous papers published on the topic of communication optimisation [17, 16, 4, 13, 25, 23, 19, 1] in data-parallel programs. However, all these approaches take as a starting point a parallel program whose control flow is well structured, i.e., containing only loops and conditionals. In particular, loops are assumed to have a unique entry point. This is, for instance, the case of the Give-N-Take framework [25], which requires the interval flow graph to be reducible, i.e., each loop must have a unique header node, and to have no critical edge, i.e., which connects a node with multiple outgoing edges to a node with multiple incoming edges. Also, Agrawal's method [1] is based on the Modified Morel and Renvoise Algorithm (MMRA [13]), which works on well structured control flow graphs, i.e., built with `while...do...end`, `for...do...end`, and `if...then...else` constructs. Because these algorithms work on well structured programs, they put a lot of emphasis on moving communications outside nested loops.

Our method, however, works equally well with programs whose control flow is general, i.e., containing also `gotos`. Hence we address loops having more than one entry and exit point. In fact, our model is general enough to cope with all finite binary deterministic labelled transition systems. N-ary transition systems could also be addressed, but for the sake of simplicity we only address binary branchings, which correspond to conditionals in the source level programming language.

Nonetheless, our method does not handle arrays, i.e., each array element must be dealt with independently. The algorithms presented in [17, 16, 25, 23, 19, 1] handle arrays more efficiently since they treat array regions. Moreover, Ref. [1] deals with inter-procedural redundancy elimination, which does not concern us since our model of a distributed program is a set of finite deterministic automata with only external procedure calls.

All these differences arise from the fact that the language used for programming data-parallel programs in the above-mentioned papers is High Performance FORTRAN (HPF [15]), which is a data computation oriented language suited to scientific applications. In contrast, finite deterministic automata are control oriented. In the domain of real-time critical systems, these can be obtained for instance from synchronous language compilers, e.g. [21, 7, 24]. Typical applications designed with these languages are nuclear plant controllers, flight control systems, and so on. These applications are labelled reactive, that is, they must react continuously to their environment, at a speed determined by the latter. This class of systems contrasts, on the one hand with transformational systems (classic programs whose inputs are available at the beginning of their execution, and which deliver their outputs when terminating: compilers, for instance), and on the other hand with interactive systems (which react continuously to their environment, but at their own speed: operating systems, for instance). Among
reactive systems can be found most of the industrial real-time systems (control, supervision, and signal-processing systems), as well as man-machine interfaces.

For both kinds of applications (scientific applications with HPF and real-time critical applications with synchronous languages), the performance of the program is a key issue. In both cases eliminating redundant messages is useful because it reduces the communication overhead.

Finally in both cases (HPF and synchronous languages), pointers are not allowed. Indeed, pointers are a potential source of programming errors, which should particularly be avoided when considering the intended application field (real-time critical systems). This is why we have chosen not to take them into account in our program model.

1.4 Paper Overview

We first introduce an example of an automatically distributed program, which has a redundant message (Section 2). Such redundant messages are inserted by automatic distribution algorithms because, at the time communication messages are inserted, only local information about the data dependencies is known. Our idea is to apply global data-flow analysis techniques [3, 2] to retrieve the list of variables that are certainly known in each state of the automaton and for each computing location. These are the local variables of the location, plus the distant variables that have not been assigned since the last time they were received. Indeed, the local copy of each such distant variable is up to date and does not need to be re-sent by its owning computing location. Then, we apply a local elimination algorithm to remove all redundant messages. Our elimination method is therefore a two-passes static analysis.

We present the two passes successively: First the global data-flow analysis (Section 3), and then the local elimination algorithm (Section 4). Each time we illustrate the algorithms with our example. Finally, we run our elimination algorithm on a digital wristwatch controller (Section 5), we analyse the complexity of our elimination algorithm (Section 6), and give some concluding remarks (Section 7).

2 Program Model and Examples

2.1 Sequential Program Model

Our model of a sequential program consists of a finite deterministic automaton. This state graph can be cyclic, but in each state, there is sequential acyclic code, represented by a binary directed acyclic graph (DAG) of actions. Besides, a program manipulates three kinds of variables:

- an input variable can be read from the environment and can only appear in the right-hand side of an assignment,
- an output variable can be written to the environment and can only appear in the left-hand side of an assignment,
- a local variable can appear both in the right-hand and in the left-hand side of an assignment.
A DAG has one root, several unary and binary nodes, and several leaves:

- An unary node is a sequential action, which can be either:
  - an input read: `input(var)`,
  - an output write: `output(var)`,
  - an assignment to an internal variable: `var:=exp`,
  - an external procedure call: `proc(...,var,...)(...,val,...)`,
    where `var` and `val` are respectively the variable and value parameters of the procedure.

- A binary node is a binary deterministic branching: `if(exp) then p else q endif`. The statement `p` is bound to the left branch while the statement `q` is bound to the right branch.

- A leaf is a state change: `goto(st)`.

Figures 1 and 2 illustrate our program model. Figure 1(a) shows an automaton with 6 states. In Figure 1(b), the DAG of each state has been expanded. Each cyan/gray edge leaving the leaf of one of the DAGs is a goto. The corresponding control flow graph is shown in Figure 1(c): Each magenta/gray edge with a white arrow is a critical edge.

![Figure 1: A general control graph.](image)

Figure 2 shows a simpler automaton with 2 states. The set of variables is `{x,y,v}`, where `x` is an input, `y` is an output, and `v` is a local variable. The drawing depicts the cyclic state graph, and for each state, a textual representation of the corresponding DAG is given. Here, both DAGs have only one leaf, but in general there can be several leaves, and thus several gotos, as shown in Figure 1(b). The program `foo` shown in Figure 2 will serve as a running example for our algorithms.
In the sequel, we will refer to a transition of the automaton as the code of the current state’s DAG, starting from its root and ending at one of its leaves. Finally, all the variables are supposed to be initialised prior to the beginning of the initial state.

This program model is the most general one since programs written with a classic imperative programming language can be compiled into this automaton model. Any such automaton can be translated into a flow graph of basic blocks, and vice-versa.

2.2 Distributed Program Model

Our model of a distributed program consists of a set of $n$ finite deterministic automata, one for each computing location. The format for each automaton is the same as in Section 2.1. Some communication mechanism remains to be chosen. Asynchronous communications allow us to place the sending actions as soon as possible in the program, and the receiving actions as late as possible, therefore minimising the waiting time induced by the communication network [14]. We choose to have two FIFO queues for each pair of locations, one in each direction. This is quite cheap in terms of execution environment, and has proven to work satisfactorily [11]. Concretely, we use two communication primitives:

- The send primitive $\text{send}(\text{dst}, \text{var})$ sends the current value of variable $\text{var}$ to location $\text{dst}$ by inserting it into the queue directed towards $\text{dst}$.

- The receive primitive $\text{var}:=\text{receive}(\text{src})$ extracts the head value from the queue starting at location $\text{src}$ and assigns it to the variable $\text{var}$. Since the target architecture is a distributed memory one, $\text{var}$ is the local copy of the distant variable maintained by the distant computing location $\text{src}$.

These primitives perform both the data-transfer and the synchronisation between the source and the destination locations: When the queue is empty, the receive is blocking. The only requirement on the network is that it must preserve the integrity and the ordering of messages.
One drawback of automatic distribution algorithms is that they often generate redundant messages. Consider the centralised program of Figure 2, distributed over two locations, according to the following specifications:

<table>
<thead>
<tr>
<th>location L</th>
<th>location M</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>x,v</td>
</tr>
</tbody>
</table>

Table 1: Distribution specifications for foo.

These specifications yield the distributed program of Figure 3. The message in gray exchanged in state 1 is clearly redundant.

![Figure 3: The program foo distributed over two locations.](image)

This is the distributed program obtained with the automatic distribution algorithm described in [12]. This algorithm is implemented in the ocrep tool [10]. The reason why it generates a redundant message in state 1 is that it proceeds one state at the time and only has a local knowledge of the data dependencies. The distribution algorithm is conservative and assumes that, at the beginning of a state, the only known variables are the ones belonging to the current computing location. The redundant message elimination algorithm we propose in this paper fits into the automatic distribution algorithm of [12]. Yet, since the underlying model is general, it can be applied to any distribution framework where distributed programs can be modelled by finite deterministic transition systems.

Note that on location M, the branching if (v<0) is useless, in the sense that both branches have the same observable behaviour. An algorithm based on observational equivalence and “on the fly” bisimulation can be found in [9]: It removes such useless branchings in distributed programs.

Now it is uneasy to remove a redundant message from a distributed program because it requires to remove both the send action and the corresponding
receive action. Therefore we choose to work on an intermediate program, right after the dependence analysis. This is also convenient since most automatic distribution algorithms proceed in three successive steps: First a computation of the non local data dependencies, then various optimisations, and finally the insertions of send and receive actions. We represent a dependence by a \texttt{dep(src, var, dst)} statement, meaning that the computing location \texttt{src} needs to send the current value of variable \texttt{var} to the computing location \texttt{dst}.

The program example we will work on will therefore be:

\begin{center}
\begin{tabular}{|l|l|}
\hline
location & state 0 \\
\hline
M & input(x); \\
M & v:=2*x; \\
LM & dep(M,v,L); \\
L & y:=v; \\
L & output(y); \\
LM & goto 1; \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|l|l|}
\hline
location & state 1 \\
\hline
LM & dep(M,v,L); \\
LM & if (v<0) then \\
L & y:=-v; \\
LM & else \\
L & y:=v; \\
LM & endif \\
L & output(y); \\
LM & goto 1; \\
\hline
\end{tabular}
\end{center}

Table 2: Intermediate code for the program \texttt{foo}.

\section{Global Data-Flow Analysis}

The technique presented here is derived from [3] and [2]. Since the internal format we use differs, we choose not to keep the classic terminology of the \textit{gen}, \textit{kill}, \textit{in}, and \textit{out} sets.

\subsection{Formalisation}

In any given state \( q \), the known variables are the variables which were already known in \( q \)'s previous states and which have not been assigned during the transition leading to \( q \), as well as the variables which have been received during this transition. Formally, we define the following sets:

\begin{itemize}
\item \( \mathcal{T} \triangleq \{ \text{variables of the programs} \} \). In our example, \( \mathcal{T} = \{ x, y, v \} \).
\item \( Q \triangleq \{ \text{states of the automaton} \} \); the states are numbered from 0 to \( |Q|-1 \); by convention 0 is the initial state. In our example, \( Q = \{ 0, 1 \} \).
\item \( \text{pre}(q) \triangleq \{ \text{immediate predecessors of state } q \} \). In our example, \( \text{pre}(0) = \emptyset \) and \( \text{pre}(1) = \{ 0, 1 \} \).
\end{itemize}
\[ E_{\text{loc}}(q) = \{ \text{variables known by location } \text{loc} \text{ at the beginning of state } q \} \]

\[ A_{\text{loc}}(p, q) = \{ \text{variables assigned by location } \text{loc} \text{ during the transition } p \rightarrow q \} \]

\[ R_{\text{loc}}(p, q) = \{ \text{variables received by location } \text{loc} \text{ and not assigned during the transition } p \rightarrow q \} \]

When the location is unspecified, we will note \( E(q) \) instead of \( E_{\text{loc}}(q) \) for instance.

Now, according to our program model (Sections 2.1 and 2.2), all the variables are initialised prior to the execution of the automaton; we thus have \( E(0) \).

Then, according to the above definitions, for any \( q \neq 0 \), \( E(q) \) is the greatest fix-point of the following equation:

\[ E(q) = \bigcap_{p \in \text{pre}(q)} \left[ \left( E(p) - A(p, q) \right) \cup R(p, q) \right] \]

(1)

For any transition \( p \rightarrow q \) of the automaton, we can compute the sets \( A(p, q) \) and \( R(p, q) \) directly. However, in order to compute the sets \( E(q) \), we need to solve a system of fix-point equations, with one equation (1) for each state \( q \).

We first get rid of the subtraction operator over sets “−”:

\[ E(q) = \bigcap_{p \in \text{pre}(q)} \left[ (E(p) \cap A(p, q)) \cup R(p, q) \right] \]

\[ = \bigcap_{p \in \text{pre}(q)} \left[ (E(p) \cup R(p, q)) \cap (A(p, q) \cup R(p, q)) \right] \]

\[ = \bigcap_{p \in \text{pre}(q)} \left[ E(p) \cup R(p, q) \right] \cap \bigcap_{p \in \text{pre}(q)} \left[ A(p, q) \cup R(p, q) \right] \]

We now define, for any state \( q \), the set \( \Psi(q) \) such that:

\[ \Psi(q) = \begin{cases} \top & \text{if } q = 0 \\ \bigcap_{p \in \text{pre}(q)} [A(p, q) \cup R(p, q)] & \text{if } q \neq 0 \end{cases} \]

(2)

We thus have to solve the following system of linear equations over the set \( Q \) of the automaton states:

\[
\begin{align*}
\vdots &= \vdots \\
E(q) &= \bigcap_{p \in \text{pre}(q)} \left[ E(p) \cup R(p, q) \right] \cap \Psi(q) \\
\vdots &= \vdots 
\end{align*}
\]

(3)

3.2 Coefficients of the Linear System

Each line of the linear system is possibly a greatest fix-point equation, depending whether \( q \) is one of its predecessors or not. We define the set \( \Phi(q) \) as:
\[ \Phi(q) = \bigcap_{p \in \text{pre}(q), p \neq q} \left[ E(p) \cup R(p, q) \right] \cap \Psi(q) \]  

(4)

If \( q \notin \text{pre}(q) \), then equation (3) becomes \( E(q) = \Phi(q) \). Otherwise, we can rewrite equation (3) to isolate \( E(q) \) in the right hand side:

\[ E(q) = \left[ E(q) \cup R(q, q) \right] \cap \Phi(q) \]  

(5)

We now solve this fix-point equation iteratively, starting from the full set \( \top \):

\[ E(q)^0 = \top \]

\[ E(q)^1 = \left[ \top \cup R(q, q) \right] \cap \Phi(q) = \top \cap \Phi(q) = \Phi(q) \]

\[ E(q)^2 = \left[ \Phi(q) \cup R(q, q) \right] \cap \Phi(q) = \Phi(q) = E(q)^1 \]

Therefore, in both cases (\( q \in \text{pre}(q) \) and \( q \notin \text{pre}(q) \)), equation (3) becomes \( E(q) = \Phi(q) \). As a result, we must now solve the following linear system:

\[
\begin{cases}
E(q) = \Phi(q) = \bigcap_{p \in \text{pre}(q), p \neq q} \left[ E(p) \cup R(p, q) \right] \cap \Psi(q) \\
E(q)^1 = \left[ \top \cup R(q, q) \right] \cap \Phi(q) = \top \cap \Phi(q) = \Phi(q) \\
E(q)^2 = \left[ \Phi(q) \cup R(q, q) \right] \cap \Phi(q) = \Phi(q) = E(q)^1 \\
\vdots
\end{cases}
\]  

(6)

Note that the matrix of this system has only empty sets on its diagonal. This will make the solving easier, as we will see in Section 3.4.

For both locations \( \text{loc}=L,M \) of the automaton of Figure 3, the linear system (6) is:

\[
\begin{cases}
E_{1\text{loc}}(0) = \top \\
E_{1\text{loc}}(1) = \left[ E_{1\text{loc}}(0) \cup R_{1\text{loc}}(0,1) \right] \cap \Psi_{1\text{loc}}(1)
\end{cases}
\]

3.3 Computation of the System’s Coefficients

The first step involves computing the sets \( A, R, \) and \( \Psi \), i.e., the coefficients of the linear system’s matrix and vector. For each location \( \text{loc} \), we define two matrices \( A_{1\text{loc}} \) and \( R_{1\text{loc}} \) of size \( |Q| \times |Q| \), and a vector \( \Psi_{1\text{loc}} \) of size \( |Q| \).

For the automaton of Figure 3, each matrix is of dimension 2 × 2 and is initialised as follows:

\[
A_{1\text{loc}} = \begin{pmatrix}
A_{1\text{loc}}[0,0] & A_{1\text{loc}}[0,1] \\
A_{1\text{loc}}[1,0] & A_{1\text{loc}}[1,1]
\end{pmatrix} := \begin{pmatrix}
\emptyset & \emptyset \\
\emptyset & \emptyset
\end{pmatrix}
\]

\[
R_{1\text{loc}} = \begin{pmatrix}
R_{1\text{loc}}[0,0] & R_{1\text{loc}}[0,1] \\
R_{1\text{loc}}[1,0] & R_{1\text{loc}}[1,1]
\end{pmatrix} := \begin{pmatrix}
\top & \top \\
\top & \top
\end{pmatrix}
\]
In each state of the automaton, we compute for each location \( \text{loc} \), the set \( \text{Assigned}_\text{loc} \) of distant variables that have been assigned by another location than \( \text{loc} \), and the set \( \text{Received}_\text{loc} \) of distant variables received by location \( \text{loc} \). For each state \( p \), the algorithm involves, for each location \( \text{loc} \), placing two empty sets \( \text{Assigned}_\text{loc} \) and \( \text{Received}_\text{loc} \) at the root of the DAG of \( p \), and then propagating these sets forward in the DAG in the following way:

1. When reaching a dependence \( \text{dep} (\text{src}, \text{var}, \text{dst}) \), add \( \text{var} \) to the set \( \text{Received}_\text{dst} \).

2. When reaching an assignment \( \text{var} := \text{exp} \), a read \( \text{input} (\text{var}) \), or an external procedure call \( \text{proc} (\ldots, \text{var}, \ldots)(\ldots) \), then for each location \( \text{loc} \) that does not own \( \text{var} \), add \( \text{var} \) to the set \( \text{Assigned}_\text{loc} \) and remove \( \text{var} \) from the set \( \text{Received}_\text{loc} \).

3. When reaching a branching \( \text{if} (\text{exp}) \), for each location \( \text{loc} \), duplicate the sets \( \text{Assigned}_\text{loc} \) and \( \text{Received}_\text{loc} \), and proceed in each branch \( \text{then} \) and \( \text{else} \).

4. When reaching a branching closure, for each location \( \text{loc} \), proceed with \( \text{Assigned}_{\text{then}} \cup \text{Assigned}_{\text{else}} \) and \( \text{Received}_{\text{then}} \cap \text{Received}_{\text{else}} \).

5. When reaching a leaf \( \text{goto} q \), fill the two matrices \( A \) and \( R \):

\[
\begin{align*}
A_{\text{loc}[p,q]} & := A_{\text{loc}[p,q]} \cup \text{Assigned}_\text{loc} \\
R_{\text{loc}[p,q]} & := R_{\text{loc}[p,q]} \cap \text{Received}_\text{loc}
\end{align*}
\]

Indeed, the results computed in \( \text{Assigned}_\text{loc} \) and \( \text{Received}_\text{loc} \) are only partial since there can be several leaves corresponding to the same transition \( p \rightarrow q \) in the automaton.

Once the matrices \( A_{\text{loc}} \) and \( R_{\text{loc}} \) are computed, we compute the coefficients of the linear system’s vector according to equation (2):

\[
\begin{align*}
\Psi_{\text{loc}[0]} & := T \\
\forall q \neq 0, \Psi_{\text{loc}[q]} & := \bigcap_{p \in \text{pre}(q)} \left( A_{\text{loc}[p,q]} \cup R_{\text{loc}[p,q]} \right)
\end{align*}
\]

Now let us apply this algorithm to our example. For state 0 of the automaton of Figure 3, we have:

<table>
<thead>
<tr>
<th>location</th>
<th>state 0</th>
<th>( \text{Assigned}_L )</th>
<th>( \text{Assigned}_M )</th>
<th>( \text{Received}_L )</th>
<th>( \text{Received}_M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>input(x);</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>M</td>
<td>v:=2*x;</td>
<td>( {x} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>L,M</td>
<td>\text{dep}(M,v,L);</td>
<td>( {x,v} )</td>
<td>( \emptyset )</td>
<td>( {v} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>L</td>
<td>y:=v;</td>
<td>( {x,v} )</td>
<td>( {y} )</td>
<td>( {v} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>L</td>
<td>output(y);</td>
<td>( {x,v} )</td>
<td>( {y} )</td>
<td>( {v} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>L,M</td>
<td>\text{goto} 1;</td>
<td>( {x,v} )</td>
<td>( {y} )</td>
<td>( {v} )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Table 3: Computation of the system’s coefficients for state 0.
And for state 1 we have:

\begin{tabular}{|c|c|c|c|c|}
\hline
\text{location} & \text{state 1} & \text{Assigned}_L & \text{Assigned}_R & \text{Received}_L & \text{Received}_R \\
\hline
L,M & \text{dep}(M,v,L); & \emptyset & \emptyset & \{v\} & \emptyset \\
L,M & \text{if } (v<0) \text{ then} & \emptyset & \{v\} & \{v\} & \emptyset \\
L & y':=v; & \emptyset & \{y\} & \{v\} & \emptyset \\
L,M & \text{else} & \emptyset & \{y\} & \{v\} & \emptyset \\
L,M & \text{output}(y); & \emptyset & \{y\} & \{v\} & \emptyset \\
L,M & \text{goto } 1; & \emptyset & \{y\} & \{v\} & \emptyset \\
\hline
\end{tabular}

Table 4: Computation of the system’s coefficients for state 1.

As a result, the matrices coefficients are:

\begin{align*}
\text{location } L: \quad & A_L = \begin{pmatrix} \emptyset & \{x,v\} \\ \emptyset & \emptyset \end{pmatrix}, \quad R_L = \begin{pmatrix} \top & \{v\} \\ \top & \{v\} \end{pmatrix} \\
\text{location } M: \quad & A_M = \begin{pmatrix} \emptyset & \{y\} \\ \emptyset & \{y\} \end{pmatrix}, \quad R_M = \begin{pmatrix} \top & \emptyset \\ \top & \emptyset \end{pmatrix}
\end{align*}

We can now compute the vector $\Psi_L$ iteratively:

\begin{align*}
\Psi_L[0] &= \top \\
\Psi_L[1] &= \bigcap_{p \in \text{pre}(q)} \left( A_L[p,q] \cup R_L[p,q] \right) \\
&= \left( A_L[0,1] \cup R_L[0,1] \right) \cap \left( A_L[1,1] \cup R_L[1,1] \right) \\
&= \left( \{x,v\} \cup \{v\} \right) \cap \left( \emptyset \cup \{v\} \right) \\
&= \left( \{y\} \cup \{v\} \right) \cap \left( \top \cup \{v\} \right) \\
&= \emptyset \cap y \quad \top \\
&= \{y,v\}
\end{align*}

We compute the vector $\Psi_M$ in the same manner and we get:

$\Psi_L = \begin{pmatrix} \top \\ \{y,v\} \end{pmatrix}, \quad \Psi_M = \begin{pmatrix} \top \\ \{x,v\} \end{pmatrix}$

### 3.4 Substitution

To solve the linear system (6), we make it triangular. We thus make substitutions in the sequence 0, \ldots, |Q| - 1. It works because the matrix of the system has only empty sets on its diagonal. In order to substitute easily a given equation into the subsequent equations of the system, we note that if $p \notin \text{pre}(q)$, then $R(p,q) = \top$, and therefore $E(p) \cup R(p,q) = E(p) \cup \top = \top$. We thus have:
\[ \bigcap_{p \neq q} [E(p) \cup R(p, q)] = \bigcap_{p \in \text{pre}(q), \ p \neq q} [E(p) \cup R(p, q)] \cap \bigcap_{p \notin \text{pre}(q), \ p \neq q} [E(p) \cup R(p, q)] \]
\[ = \bigcap_{p \in \text{pre}(q), \ p \neq q} [E(p) \cup R(p, q)] \cap \top \]
\[ = \bigcap_{p \in \text{pre}(q), \ p \neq q} [E(p) \cup R(p, q)] \]

As a result, each equation of the system (6) becomes:
\[ E(q) = \bigcap_{p \neq q} [E(p) \cup R(p, q)] \cap \Psi(q) \quad (7) \]

Starting from the modified linear system, we make substitutions in the sequence 0, \ldots, |Q| - 1. At step \( i \), we substitute \( E(i) \) in the lines \( i + 1 \) to \( |Q| - 1 \), the \( i \) former substitutions being already done:

\[
\begin{align*}
\vdots & \\
E(i) & = \bigcap_{i' > i} [E(i') \cup R(i', i)] \cap \Psi(i) \\
\vdots & \\
E(j) & = \bigcap_{j' > i, \ j' \neq j} [E(j') \cup R(j', j)] \cap \Psi(j) \\
\vdots & 
\end{align*}
\quad (8)
\]

Now, if \( i \notin \text{pre}(j) \), then there is no substitution to make. Otherwise, we substitute \( E(i) \) in line \( j \) in system (8):

\[
\begin{align*}
E(j) = & \bigcap_{j' > i} [E(j') \cup R(j', j)] \cap [E(i) \cup R(i, j)] \cap \Psi(j) \\
= & \bigcap_{j' > i} [E(j') \cup R(j', j)] \cap \bigcap_{i' > i} [E(i') \cup R(i', i) \cup R(i, j)] \\
& \cap [\Psi(i) \cup R(i, j)] \cap \Psi(j) \\
= & \bigcap_{j' > i} [E(j') \cup (R(j', j) \cap (R(j', i) \cup R(i, j)))] \\
& \cap [E(j) \cup R(j, i) \cup R(i, j)] \cap [\Psi(i) \cup R(i, j)] \cap \Psi(j) \\
\end{align*}
\quad (9)
\]

Equation (9) is again a fix-point equation of the form \( E(j) = [E(j) \cup R(i, j) \cup R(j, i)] \cap \Omega(i, j) \), where \( \Omega(i, j) \) is the set defined as:
\[ \Omega(i, j) = \bigcap_{j' > i, j' \neq j} \left[ E(j') \cup \left( R(j', j) \cap \left( R(j', i) \cup R(i, j) \right) \right) \right] \cap \left[ \Psi(i) \cup R(i, j) \right] \cap \Psi(j) \]

(10)

Since we are looking for the greatest fix-point, we solve it iteratively starting from \( \top \). Just as with equation (5), we find \( E(j) = \Omega(i, j) \), which amounts to suppressing the term \( \left[ E(j) \cap R(j, i) \cap R(i, j) \right] \) in equation (9). Thus the line \( j \) of the linear system (8) becomes:

\[ E(j) = \bigcap_{j' > i, j' \neq j} \left[ E(j') \cup \left( R(j', j) \cap \left( R(j', i) \cup R(i, j) \right) \right) \right] \cap \left[ \Psi(i) \cup R(i, j) \right] \cap \Psi(j) \]

(11)

Therefore, substituting \( E(i) \) in line \( j \) in system (8) amounts to performing the following computations on the matrix \( R \) and the vector \( \Psi \):

\[
R[j', j] := R[j', j] \cap \left( R[j', i] \cup R[i, j] \right), \quad \forall j' > i \text{ and } j' \neq j
\]

(12)

\[
\Psi[j] := \Psi[j] \cap \left( \Psi[i] \cup R[i, j] \right)
\]

(13)

Performing these computations directly on \( R \) and \( \Psi \) allows us to save memory space.

By applying these steps to both locations \( \text{loc}=L,M \) of the automaton of Figure 3, we get the following system:

\[
\begin{align*}
E_{\text{loc}}(0) &= \top \\
E_{\text{loc}}(1) &= \Psi_{\text{loc}}(1)
\end{align*}
\]

3.5 Solving the System

Once all the substitutions are done, our linear system is triangular:

\[
\begin{align*}
E(0) &= \top \\
\vdots \\
E(i) &= \bigcap_{i' > i} \left[ E(i') \cup R(i', i) \right] \cap \Psi(i) \\
\vdots \\
E([Q] - 1) &= \Psi([Q] - 1)
\end{align*}
\]

(14)

We thus compute the sets \( E(i) \) in the sequence \([Q] - 1, \ldots, 0\), by replacing each time the set previously computed in the remaining lines. When the set \( E(i) \) is computed, replacing its value in line \( j \) (with \( j < i \)) amounts to modifying the row \( j \) of vector \( \Psi \) in the following way:

\[
\Psi[j] := \Psi[j] \cap \left( \Psi[i] \cup R[i, j] \right)
\]

(15)

For both locations \( \text{loc}=L,M \) of the automaton of Figure 3, the linear system (14) is trivially solved as:
A. Girault / Parallel Computing, 28(2002), 433–453

3.6 Analysis of our Example

For the program of Figure 3, we have the following sets of known variables:

\[
\text{location } L: \begin{cases} E_L(0) = \{x, y, v\} \\ E_L(1) = \{y, v\} \end{cases} \quad \text{location } M: \begin{cases} E_M(0) = \{x, y, v\} \\ E_M(1) = \{x, v\} \end{cases}
\]

In the initial state 0, all the variables are known: This is consistent with our assumptions of Section 3.1. In state 1, since location L owns y, \( y \in E_L(1) \). Besides, \( v \in E_L(1) \), which means that location L also knows v. Moreover, location M only knows the variables it owns, i.e., x and v. These results are consistent with the source code.

4 Local Elimination

Now that we have computed for each state \( q \) the sets of variables known by each computing location at the beginning of \( q \), there remains to eliminate the redundant \texttt{dep} statements in each state of the automaton.

In each state \( q \) of the automaton, for each location \( \text{loc} \) we compute the set \( \text{Known}_{\text{loc}} \) of variables currently known, i.e., whose values have been previously sent to \( \text{loc} \) and which have not been modified by their owning location since then. For each location \( \text{loc} \), the algorithm involves initialising \( \text{Known}_{\text{loc}} \) to the set \( \Psi[\text{loc}] \) computed in Section 3.5, starting at the root of the DAG of state \( p \), and propagating the set \( \text{Known}_{\text{loc}} \) forward in the DAG in the following way:

1. When reaching a dependence \( \texttt{dep}(\text{src}, \text{var}, \text{dst}) \), if \( \text{var} \in \text{Known}_{\text{dst}} \), then suppress this dependence \( \texttt{dep}(\text{src}, \text{var}, \text{dst}) \); otherwise add \( \text{var} \) to \( \text{Known}_{\text{dst}} \).

2. When reaching an assignment \( \text{var} := \text{exp} \), a read input(\text{var}), or an external procedure call proc(...) on ..., then for each location \( \text{loc} \) that does not own \( \text{var} \), remove \( \text{var} \) from the set \( \text{Known}_{\text{loc}} \).

3. When reaching a branching if(\text{exp}), duplicate the sets \( \text{Known}_{\text{loc}} \), and proceed in each branch then and else.

4. When reaching a branching closure, for each location \( \text{loc} \), proceed with \( \text{Known}_{\text{loc}}^{\text{then}} \cap \text{Known}_{\text{loc}}^{\text{else}} \).

5. When reaching a leaf goto \( q \), the algorithm terminates.

Applying this algorithm to the program of Figure 3 gives the following result for state 0. Recall that y belongs to location L while x and v belong to location M.
Table 5: Local elimination for state 0.

At the point of the \texttt{dep(M,v,L)} statement, \( v \not\in \text{Known}_L \), so this dependence is not suppressed.

For state 1 we have:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
location & state 1 & Known_L & Known_M \\
\hline
L,M & \texttt{dep(M,v,L);} & \{y,v\} & \{x,v\} \\
L,M & \texttt{if (v<0) then} \texttt{y:=-v;} & \{y,v\} & \{x,v\} \\
L,M & \texttt{else} \texttt{y:=v;} & \{y,v\} & \{x,v\} \\
L,M & \texttt{endif} & \{y,v\} & \{x,v\} \\
L & \texttt{output(y);} & \{y,v\} & \{x,v\} \\
L,M & \texttt{goto 1;} & & \\
\hline
\end{tabular}
\end{table}

Table 6: Local elimination for state 1.

This time, at the point of the \texttt{dep(M,v,L)} statement, \( v \in \text{Known}_L \), so the dependence is suppressed! The resulting distributed program, obtained by inserting the \texttt{send} and \texttt{receive} actions according to the \texttt{dep} statements, is shown in Figure 4:
5 An Example: The Wristwatch Controller

We run our elimination algorithm on a more realistic example, a digital wristwatch controller [6]. This example is interesting because of its relative complexity. Numerous commands are folded into four buttons, which are the inputs of the program. This watch has a time-keeper, a stopwatch, and an alarm. It is inspired by a Casio watch. The statistics of the centralised wristwatch controller are:

<table>
<thead>
<tr>
<th>Number</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of states</td>
<td>42</td>
</tr>
<tr>
<td>number of actions</td>
<td>1857</td>
</tr>
<tr>
<td>number of transitions</td>
<td>298</td>
</tr>
<tr>
<td>number of variables</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 7: Statistics of the centralised wristwatch controller.

We choose to distribute it over five computing locations, according to the following specifications:
Table 8: Distribution specifications for the wristwatch controller.

The obtained distributed controller contains a total of 1,920 sends/receives between the five computing locations. The following table shows how they are located:

<table>
<thead>
<tr>
<th>location</th>
<th>sends/receives</th>
<th>actions</th>
<th>local variables</th>
<th>distant variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>653</td>
<td>1690</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>712</td>
<td>1205</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>235</td>
<td>1934</td>
<td>7</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>128</td>
<td>1077</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>192</td>
<td>1488</td>
<td>4</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 9: Send/receive statistics for the distributed wristwatch controller.

Our redundant message elimination removes a total of 236 messages. We therefore achieve a 12% gain in the number of sends/receives. The following table shows the new statistics:

<table>
<thead>
<tr>
<th>location</th>
<th>sends/receives</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>653</td>
<td>1690</td>
</tr>
<tr>
<td>2</td>
<td>688</td>
<td>1181</td>
</tr>
<tr>
<td>3</td>
<td>105</td>
<td>1804</td>
</tr>
<tr>
<td>4</td>
<td>104</td>
<td>1053</td>
</tr>
<tr>
<td>5</td>
<td>136</td>
<td>1416</td>
</tr>
</tbody>
</table>

Table 10: Send/receive statistics for the optimised wristwatch controller.

This gain results in an improvement in performance of the wristwatch controller. Generally, a reactive system must react continuously to its environment, at a speed determined by the latter (see Section 1.3). For this kind of applications, one important step is to check, at the end of the design, that the object code obtained meets the real-time constraints, i.e., does not miss a single input event from the environment. This check is easily performed on the sequential
automaton obtained after compiling. For the wristwatch example, the environment sends the inputs $S$ and $HS$, respectively emitted every second and $1/100$ of a second. This means that any transition of the automaton must complete faster than $1/100$ of a second. When the program is distributed, the problem is the same, but one has also to take into account the communication time induced by the sends/receives. For this purpose, eliminating redundant messages is particularly useful, because an improvement in performance resulting from redundant message elimination could cause the optimised distributed program to meet its real-time constraints, while the non optimised one would fail to do so.

6 Complexity Analysis

Now let us compute formally the time and memory requirements for our algorithm. We adopt the following notations:

- For any procedure $P$, $T(P)$ and $M(P)$ denote respectively its time and memory requirements.
- $|Q|$ is the number of states of the automaton.
- $nb_{act}(q)$ is the number of actions of state $q$.
- $nb_{loc}$, $nb_{var}$ and $nb_{act}$ are respectively the number of locations, of variables and of actions of the distributed automaton.
- $av_{var} = \frac{nb_{var}}{nb_{loc}}$ is the average number of variables belonging to a given location.

Our redundant messages elimination algorithm consists of four steps:

1. coefficients computation (Section 3.3)
2. linear system substitution (Section 3.4)
3. linear system solving (Section 3.5)
4. local elimination (Section 4)

We assume that the implementation allows any set operation to be performed in $O(av_{var})$, for instance with bit-streams. Hence, the time requirement for the coefficients computation is $av_{var}$ times the cost of the action graph traversal, which is $\sum_{q \in Q} nb_{act}(q)$. Thus:

$$T(\text{coefficients computation}) = \mathcal{O}(nb_{act} \times av_{var})$$

The memory requirement for the coefficients computation is the cost of the \texttt{Assigned}_{loc} and \texttt{Received}_{loc} sets, for each location \texttt{loc}. Thus:

$$M(\text{coefficients computation}) = \mathcal{O}(2 \times nb_{var} \times nb_{loc})$$

Since the data structures used during the substitution of the linear system already exist, the corresponding memory requirement is negligible. Concerning the time requirement, it is given by:
\[ T(\text{substitution}) = nb_{loc} \times \sum_{i=0}^{|Q|-1} \sum_{j=1}^{|Q|-1} \left( T(\text{computation (12)}) + T(\text{computation (13)}) \right) \]

where “computation (12)” stands for the computation given by equation (12). Then \( T(\text{computation (12)}) = \sum_{j=1}^{|Q|-1} O(nb_{var}) = \mathcal{O}((|Q|-i)\times nb_{var}) \).

Moreover, \( T(\text{computation (13)}) = \mathcal{O}(nb_{var}) \). Therefore:

\[ T(\text{substitution}) = \mathcal{O}(nb_{loc} \times |Q|^3 \times nb_{var}) \]

\[ M(\text{substitution}) = \mathcal{O}(1) \]

Exactly like the substitution, the data structures used during the solving of the linear system already exist, so the corresponding memory requirement is negligible. Concerning the time requirement, it is given by:

\[ T(\text{solving}) = nb_{loc} \times \sum_{j=|Q|-1}^{j=0} \sum_{k=j}^{k=j} T(\text{computation (15)}) \]

Computation (15) is a basic set operation, thus \( T(\text{computation (15)}) = \mathcal{O}(nb_{var}) \). Therefore:

\[ T(\text{solving}) = \mathcal{O}(nb_{loc} \times nb_{var} \times |Q|) \]

\[ M(\text{solving}) = \mathcal{O}(1) \]

The last step is the local elimination. Its memory requirement is negligible since the data structures already exist. Its time requirement is the cost of a set operation times the cost of the action graph traversal. Thus:

\[ T(\text{local elimination}) = \mathcal{O}(nb_{act} \times av_{var}) \]

\[ M(\text{local elimination}) = \mathcal{O}(1) \]

As a result, the time and memory requirements for our algorithm are:

\[ T(\text{global elimination}) = \mathcal{O}(nb_{act} \times av_{var}) + \mathcal{O}(nb_{loc} \times |Q|^3 \times nb_{var}) \]
\[ + \mathcal{O}(nb_{loc} \times nb_{var} \times |Q|^2) + \mathcal{O}(nb_{act} \times av_{var}) \]
\[ = \mathcal{O}(nb_{act} \times av_{var}) + \mathcal{O}(nb_{loc} \times |Q|^3 \times nb_{var}) \]

\[ M(\text{global elimination}) = \mathcal{O}(2 \times nb_{var} \times nb_{loc}) + \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) \]
\[ = \mathcal{O}(nb_{var} \times nb_{loc}) \]
7 Conclusion

We have presented an efficient optimisation for parallel programs that have been distributed automatically. This optimisation concerns redundant messages. The starting point is a distributed program, where the dependence analysis has already been performed. The program model is a finite deterministic automaton with a DAG of actions in each state. This is the most general model since it can represent programs with gotos.

Our elimination method works in two passes: First a global data-flow analysis, and then a local elimination. The goal of the first pass is to compute the set of known distant variables, for each state of the automaton. These sets are computed by solving a system of fix-point equations. The second pass is achieved by a traversal of the DAG of each state. We have presented the algorithms involved during both passes as well as the associated time and memory costs.

Finally, since the internal format we are working with is finite state automata with DAGs of actions, it is easy to apply our elimination algorithm to many automatic parallelisers. In particular, it is implemented within our ocrep paralleliser [12, 10] and works satisfactorily.

Acknowledgement

Many thanks to Paul Caspi and Jean-Claude Fernandez of VERIMAG and LSR lab, Grenoble, France, as well as the anonymous reviewers, for their helpful suggestions!

References


of Software Technology and Theoretical Computer Science, FST&TCS’95, volume
1026 of LNCS, Bangalore, India, December 1995. Springer-Verlag.

www.inrialpes.fr/bip/people/girault/Ocrep.

K. Ali, and P. Magnusson, editors, 1st International Conference on Parallel Pro-
cessing, EURO-PAR’95, volume 966 of LNCS, pages 15–26, Stockholm, Sweden,

for asynchronous networks of processors. IEEE Trans. on Software Engineering,

flow problems unig edge placement. ACM Trans. on Programming Languages and


fication. Technical Report CRPC-TR92225, Rice University, Houston, USA, May
1993.

[16] C. Gong, R. Gupta, and R. Melhem. Compiling techniques for optimizing commu-
nication on distributed-memory systems. In International Conference on Parallel
Processing, IPPS’93, St. Charles, USA, August 1993.


Ching, and T. Ngo. An HPF compiler for the IBM SP2. In Supercomputing’95,
San Diego, USA, December 1995. ACM.

communication in data-parallel programs. IEEE Trans. on Parallel and Dis-


[21] N. Hallbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-
flow programming language LUSTRE. Proceedings of the IEEE, 79(9):1305–1320,
September 1991.

for FORTRAN D on MIMD distributed-memory machines. In ACM International

to optimize communication. In 9th International Parallel Processing Symposium,
IPPS’95, Santa Barbara, USA, April 1995.


framework. In Conference on Program Language Design and Implementation,