Distributed Implementation of Programmed Graph Reduction

Rita Loogen, Herbert Kuchen and Klaus Indermark
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, 5100 Aachen, West Germany

Werner Damm
FB Informatik, Universität Oldenburg
Postfach 2503, 2900 Oldenburg, West Germany

Abstract
Programmed graph reduction has been shown to be an efficient implementation technique for lazy functional languages on sequential machines. Considering programmed graph reduction as a generalization of conventional environment-based implementations where the activation records are allocated in a graph instead of on a stack it becomes very easy to use this technique for the execution of functional programs in a parallel machine with distributed memory. We describe in this paper the realization of programmed graph reduction in PAM — a parallel abstract machine with distributed memory. Results of our implementation of PAM on an OCCAM-Transputersystem are given.

1 Introduction
The most important properties of lazy functional programs are their expressiveness which is due to the possibility to work with higher order functions and infinite data structures, their semantic simplicity which is due to their mathematical foundation and finally their implicit parallelism which is due to the Church-Rosser-property of the reduction semantics. There is no à priori need to extend a functional language by special syntactic constructs to indicate parallelism that can be exploited in an implementation on a parallel machine. A parallelizing compiler may detect the implicit parallelism and decompose the functional program for parallel execution. Thus, the programmer need not think about the organisation of parallelism at all.

An approach to the automatic parallelization of functional programs has been described in [Hudak/Goldberg 85]: Each functional program can be translated into a system of serial combinators, i.e. a set of fully lazy global function definitions in which the places where parallel evaluation of subexpressions should take place are indicated by a special syntactic construct. Some form of strictness analysis is used to detect the maximal parallelism within a program. We will assume here that the technique of evaluation transformers, which has been introduced in [Burn 87a] and which handles structured data types in an appropriate way, is used for strictness analysis. For each predefined function and for each combinator an evaluation transformer tells us which amount of evaluation can be done on the arguments of the function or combinator if we know how much evaluation has to be done on the application. This information will be used in the parallel machine to control the evaluation. In section 3 we will give a short introduction to evaluation transformers as we will use them.

We consider higher order combinator systems with non-curried functions and structured data types. The combinators will be given in a flat syntactic form, which provides a clean distinction
between first-order and higher-order expressions. As we will point out later the special syntactic form enables us to treat first-order expressions in a more efficient way during the reduction process.

The parallel implementation of the combinator systems is based on a parallel abstract machine (PAM), that consists of a finite number of identical processor elements with local storage. The processor elements may communicate by exchanging messages via an interconnection network.

PAM has a very modular structure. Each processor element consists of two independent processing units:

- a communication unit that is responsible for the organizational aspects of the parallelization of the reduction process, and
- a reduction unit that executes the parallel processes that have been derived from the original functional program, by programmed graph reduction.

The modularization of the processor elements simplifies the formal specification of the parallel machine as it is e.g. given in [Loogen 88] by the use of state transition systems and in [Loogen 87] by the use of an axiomatic architecture description language.

On the other hand the modularization causes a decentralization of the parallel program execution by separating the overhead of parallelism — message handling, work distribution, workload balancing — from the reduction process. This promises a better exploitation of parallelism.

Due to space limitations we omit a complete formal specification of the abstract machine in this paper and describe the state transitions of the machine in sections 4 and 5 only in an informal way. A detailed description can be found in [Loogen, Kuchen et al. 89].

The abstract machine has been implemented on an OCCAM/Transputersystem. In the final sections we describe this implementation and present first simulation results. Furthermore we give an overview of and comparison with some related projects.

## 2 Serial Combinator Systems

A program \( \mathcal{P} \) in the parallel intermediate language consists of a serial combinator system \( \mathcal{R} \) and an applicative expression that defines the value of the program:

\[
\mathcal{P} = (\text{main} = e, \mathcal{R}) \text{ with } \mathcal{R} = \left\{ \begin{array}{l}
F_1(x_1, \ldots, x_{n_1}) = e_1 \\
\vdots \\
F_r(x_1, \ldots, x_{n_r}) = e_r
\end{array} \right\}.
\]

The serial combinator system \( \mathcal{R} \) contains a finite number of recursive combinator definitions whose bodies are parallelized applicative expressions.

Let \( \Omega \) denote the set of basic functions and \( \Gamma \) the set of data constructors. Let \( V \) be a set of typed argument and local variables and \( C \) be a set of combinator variables.

The set \( PExp \) of parallelized applicative expressions is the smallest set with

1. variables and constants: \( V \cup C \cup \Omega \cup \Gamma \subseteq PExp \),

2. simple conditional expressions: if \( e \) then \( e_1 \) else \( e_2 \) \( \mathsf{if} \in PExp \)

where \( e \in PExp \) with type \( \text{bool} \) and \( e_1, e_2 \in PExp \),

3. complete case analysis on the possible decompositions of a data structure

\[
\text{case } e \text{ of } c_1(y_{11}, \ldots, y_{1m_1}) : e_1; \cdots; c_k(y_{k1}, \ldots, y_{kn_k}) : e_k \mathsf{esac} \in PExp
\]

where \( e \in PExp \) is an element of a data structure \( d \) and \( \{c_1, \ldots, c_k\} \subseteq \Gamma \) are the constructors of type \( d \),