Proc. PARLE 1989, LNCS 365, Springer Verlag 1989, pp. 136–157

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:

$$\mathcal{P} = \langle \mathbf{main} = e, \mathcal{R} \rangle \text{ with } \mathcal{R} = \left\{ \begin{array}{rcl} F_1(x_1, \ldots, x_{n_1}) & = & e_1, \\ & \vdots & \\ F_r(x_1, \ldots, x_{n_r}) & = & e_r, \end{array} \right\}.$$

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

Let Ω denote the set of basic functions and Γ 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*: $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \mathbf{fi} \in PExp$ where $e \in PExp$ with type *bool* and $e_1, e_2 \in PExp$,
3. *complete case analysis on the possible decompositions of a data structure*

$$\mathbf{case}\ e\ \mathbf{of}\ c_1(y_{11}, \ldots, y_{1n_1})\ :\ e_1; \cdots;\ c_k(y_{k1}, \ldots, y_{kn_k})\ :\ e_k\ \mathbf{esac} \in PExp$$

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,

4. *first order applications* of constants and combinators where we admit partial applications of basic functions and combinators up to the arity of the function symbol:

$$\Phi(e_1, \ldots, e_m) \in PExp,$$

where $\Phi \in \Omega \cup \Gamma \cup Com$ and $arity(\Phi) = m$ if $\Phi \in \Gamma$ and $arity(\Phi) \geq m$ otherwise,

5. *higher order applications*: $\mathrm{ap}(e, e_1, \ldots, e_m) \in PExp$ where $e \in PExp$ is an arbritrary functional expression.

6. *explicit parallelism:*

$$\begin{array}{lll} \textbf{letpar} & y_1 = F_1(e_{11}, \ldots, e_{1n_1}) & \\ \textbf{and} & \quad\quad\cdots & \\ \textbf{and} & y_p = F_p(e_{p1}, \ldots, e_{pn_p}) & \\ \textbf{in } e & & \in PExp. \end{array}$$

where $p \geq 1, y_i \in V (1 \leq i \leq p)$ and $F_i(e_{i1}, \ldots, e_{in_i}) \in PExp$ are (complete) combinator applications with $F_i \in C$ $(1 \leq i \leq p)$.

The letpar-construct is used to make parallelism in the intermediate program explicit. Subexpressions that should be evaluated in parallel are abstracted out from combinator bodies and represented by applications of 'serial' combinators. During the parallelization process new combinators may be defined to represent such parallel subexpressions. The remaining body e which contains — via the local variables y_i — references to the parallel subtasks is called the *sequential thread* of the computation.

Parallel subtasks always correspond to combinator applications. There are two sources of parallelism within serial combinator systems. On the one hand parallel processes will result from the execution of **letpar**-constructs. On the other hand a parallel process will also be generated for the delayed execution of a combinator application in a non-strict argument position.

Expressions and combinator definitions are monomorphically typed because we use a strictness analysis that is only adaptable for monomorphically typed expressions.

We omit a formal description of the reduction semantics as this is straightforward assuming a strict interpretation of the base functions. The data constructors are handled as non-strict non-interpreted functions. The combinator definitions of the serial combinator system are used as rewrite rules. Additionally there are special rules for the treatment of conditional expressions and higher-order applications. The rules for higher-order applications simply describe the collection of arguments for the completion of partial applications. An example program will be given in the next section.

3 Evaluation Transformers

In the evaluation of functional programs with structured data types several levels of evaluation may be distinguished. In [Burn 87a] a selection of evaluation levels is considered that seems to be natural for the treatment of structured data types. These levels are described by the following *evaluators*, which are used to evaluate expressions:

ξ_0 — does no evaluation,

ξ_1 — evaluates expressions to weak head normal form,

ξ_2 — evaluates the structure of data types,

ξ_3 — evaluates the structure of data types and each element of the data type to weak head normal form.

The first two evaluators are common for the lazy evaluation of functional programs. The new evaluators will be used to organize the evaluation of data structures in a more appropriate way. Evaluating the structure of a data type simply means evaluating the sequence of constructor nodes of that data type.

In the following, let the set of evaluators applicable to an expression e of type t be given by

$$Evset^t := \begin{cases} \{\xi_0, \ldots, \xi_3\} & \text{if } t \text{ is a structured type} \\ \{\xi_0, \xi_1\} & \text{otherwise} \end{cases}.$$

The evaluators $\xi_0, \ldots, \xi_3$ handle only top-level data structures. In the case of nested data structures, e.g. lists of integerlists, it is not possible to force the evaluation of each element of a structure with evaluator ξ_2 or ξ_3. To handle nested structures in a more sophisticated way one would have to extend the set of evaluators. As this set has to be finite it is only possible to specify the degree of evaluation of nested data structures up to a finite depth.

Using the technique of abstract interpretation one can derive *evaluation transformers* for basic functions, constructors and combinators of a serial combinator system [Burn 87b, Loogen 88]. An evaluation transformer tells us for a function which evaluator may be used for the evaluation of the arguments in an application when the evaluator for the application is given.

Thus, if we have a function ϕ of type $t_1 \times \cdots \times t_k \to t_0$, the corresponding evaluation transformer $\mathrm{ET}(\phi)$ is a mapping from the set of evaluators $Evset^{t_0}$ into the set of k-tuples of evaluators from $Evset^{t_1} \times \cdots \times Evset^{t_k}$. We denote the component functions of $\mathrm{ET}(\phi)$ by $\mathrm{ET}_i(\phi)$ for $1 \leq i \leq k$.

As we assumed a strict interpretation we get the following evaluation transformers for n-ary base functions $f \in \Omega$

$$\forall i \in \{1, \ldots, n\}\ \forall \xi \in \{\xi_0, \xi_1\}\ \mathrm{ET}_i(f)(\xi) = \xi.$$

For data constructors $c \in \Gamma$ of type d with arity $k \geq 0$ and argument types $t_1, \ldots t_k$ the evaluation transformer can be defined by:

$$\begin{aligned} \forall i \in \{1, \ldots, k\} \text{ with } t_i \neq d\ \forall \xi \in Evset^d\ \mathrm{ET}_i(c)(\xi) &:= \text{ if } \xi = \xi_3 \text{ then } \xi_1 \text{ else } \xi_0 \text{ fi} \\ \text{and } \forall i \in \{1, \ldots, k\} \text{ with } t_i = d\ \forall \xi \in Evset^d\ \mathrm{ET}_i(c)(\xi) &:= \text{ if } \xi = \xi_1 \text{ then } \xi_0 \text{ else } \xi \text{ fi.} \end{aligned}$$

To derive the evaluation transformers of combinators one has to use the techniques described in [Burn 87b, Loogen 88].

The activation of parallel subtasks will naturally depend on the given evaluator. For this reason the syntax of the letpar-construct will be slightly changed to become

$$\begin{array}{lll} \textbf{letpar}\ y_1 & = F_1(e_{11}, \ldots, e_{1n_1})\ \textbf{if}\ ev_1 \\ \textbf{and} & \quad \cdots \\ \textbf{and} \quad y_p & = F_p(e_{p1}, \ldots, e_{pn_p})\ \textbf{if}\ ev_p \\ \textbf{in}\ e & \quad \in PExp. \end{array}$$

under the assumptions given in section 2 and $ev_1, \ldots, ev_k \in Evset$.

The parallel activation of the combinator applications depends on the evaluator of the **letpar**-construct. The combinator application $F_i(e_{i1}, \ldots, e_{in_i})$ will be evaluated in parallel only if the evaluator of the letpar-expression is stronger than or equal to ev_i.

The evaluators for the combinator applications will be determined by the evaluation transformer of the sequential thread of the letpar-expression. For this reason we will additionally assume that each letpar-expression is annotated by the evaluation transformer of its sequential thread.

One could also note the evaluation transformers of the different alternatives of case-expressions and use this information to speed up the evaluation of the components of data structures. We omit this optimization here for simplicity.

Example: The following first order example program computes all solutions of the well-known queens problem. It has been adapted from [Hudak/Goldberg 85]. The types of parameters and combinator applications are noted in italic font style.

The following data structures are used in this example:
intlist with constructors NIL $\in \Gamma^{intlist}$ and CONS $\in \Gamma^{int \times intlist \rightarrow intlist}$,
and *listofintlist* with constructors LNIL $\in \Gamma^{listofintlist}$ and LCONS $\in \Gamma^{intlist \times listofintlist \rightarrow listofintlist}$.

main: *listofintlist*
= print(allsolutions(NIL, 0))
allsolutions (psol: *intlist*, col: *int*): *listofintlist*
= **if** col=8 **then** LCONS(psol, LNIL) **else** findsolutions(psol, col, 0) **fi**
findsolutions (psol: *intlist*, col: *int*, row: *int*): *listofintlist*
= **if** row=8 **then** LNIL **else letpar** y_1 = findsolutions(psol, col, row+1) **if** ξ_2
and y_2 = complete(psol, col, row) **if** ξ_2
in append(y_1, y_2) **fi**
complete (psol: *intlist*, col: *int*, row: *int*): *listofintlist*
= **if** safe(col, row, psol, col$-$1)
then allsolutions (CONS(row, psol), col +1) **else** LNIL **fi**
safe (col: *int*, row: *int*, ps: *intlist*, col1: *int*): *bool*
= **case** ps **of** NIL: true;
CONS(h,t): (h$\neq$row and col$\neq$col1) and
(| h$-$row |$\neq$|col$-$col1| and safe(col, row, t, col1 $-$1))
esac

The evaluation transformers for the combinators of this example are given by the following table where $\xi \neq \xi_0$:

combinators	$ET_1(.)(\xi)$	$ET_2(.)(\xi)$	$ET_3(.)(\xi)$	$ET_4(.)(\xi)$
allsolutions	ξ_0	ξ_1		
findsolutions	ξ_0	ξ_0	ξ_1	
complete	ξ_3	ξ_0	ξ_0	
safe	ξ_0	ξ_0	ξ_3	ξ_0

These evaluation transformers are very simple as they do not depend on the given evaluator ξ. The letpar-expression in the body of the combinator findsolutions indicates the parallel evaluation of the arguments of the library function append only if the actual evaluator is stronger than ξ_2. This becomes clear if one considers the evaluation transformer of the function append defined by:

append (l_1 : *listofintlist*, l_2 : *listofintlist*) : *listofintlist*
:= **case** l_1 **of** LNIL : l_2; LCONS(h, t): LCONS(h,append(t, l_2)) **esac**

The evaluation transformer of append is given in the following table:

ξ	$ET_1(\text{append})(\xi)$	$ET_2(\text{append})(\xi)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_1	ξ_0
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_3

The second argument may only be evaluated if the evaluator is stronger than ξ_2. As the sequential thread of the letpar-expression equals the function append, the evaluation transformer of append is also used to annotate the evaluation transformer of the letpar-expression.

4 The Parallel Abstract Machine

In this section we give an overview of our parallel abstract machine concentrating on the organization of parallelism. The implementation of serial combinators and the realization of programmed

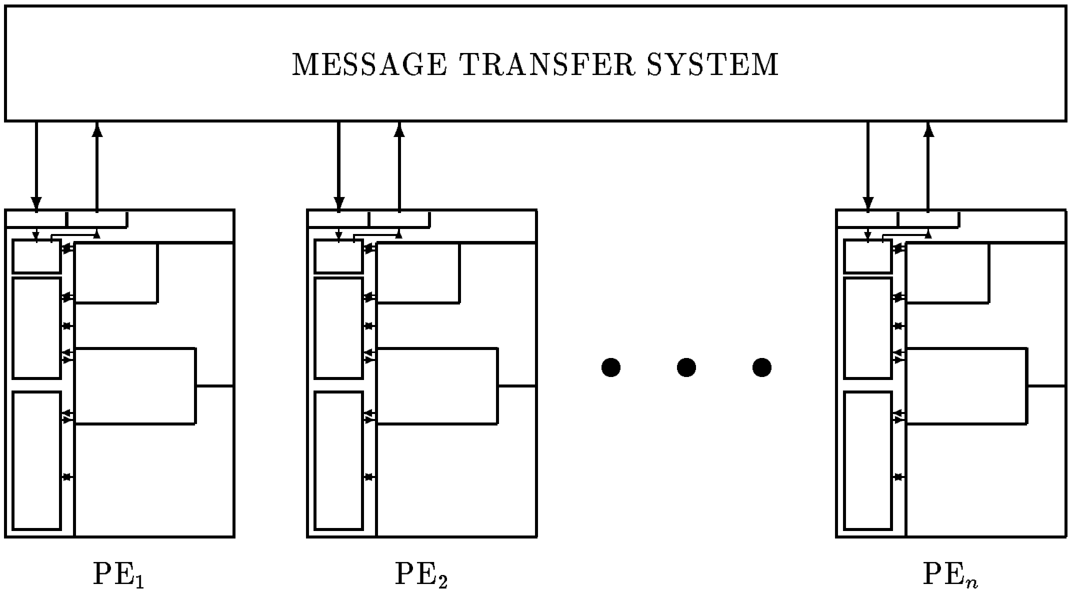

Figure 1: Global Structure of PAM

graph reduction in the parallel machine will be discussed in the next section. A complete formal specification of the machine can be found in [Loogen 87/88].

The parallel abstract reduction machine consists of a finite number of identical processor elements with local storage which may communicate with each other via a message transfer system (see figure 2). For the purpose of this paper we assume an interconnection network which provides a logically complete communication between processor elements.

4.1 Processor Elements

Each processor element consists of two autonomous processing units — the communication unit and the reduction unit — which may communicate via exchange of messages in local shared memories. The communication units manage the distribution of work. The reduction units perform the combinator reductions by programmed graph reduction. Figure 2 shows the structure of a single processor element.

The communication unit contains two independent processors:

- The communication processor is responsible for the organization of the parallelism. Its main task is the distribution of the parallel processes and the workload balancing.
- The network adapter collects the incoming messages from the inports and forwards them to the communication processor. Furthermore, it passes messages from the communication processor to the correct outports.

The shared memories within the local store of a processor element contain queues for the transfer of messages between the different processing units. The *next-task-flag* in the common store of reduction and communication unit can be set by the reduction unit to ask the communication processor for a new process. To avoid conflicts the flag can only be reset by the communication processor.

4.2 Reduction Messages

Four types of messages are essential for the parallelization of the graph reduction process:

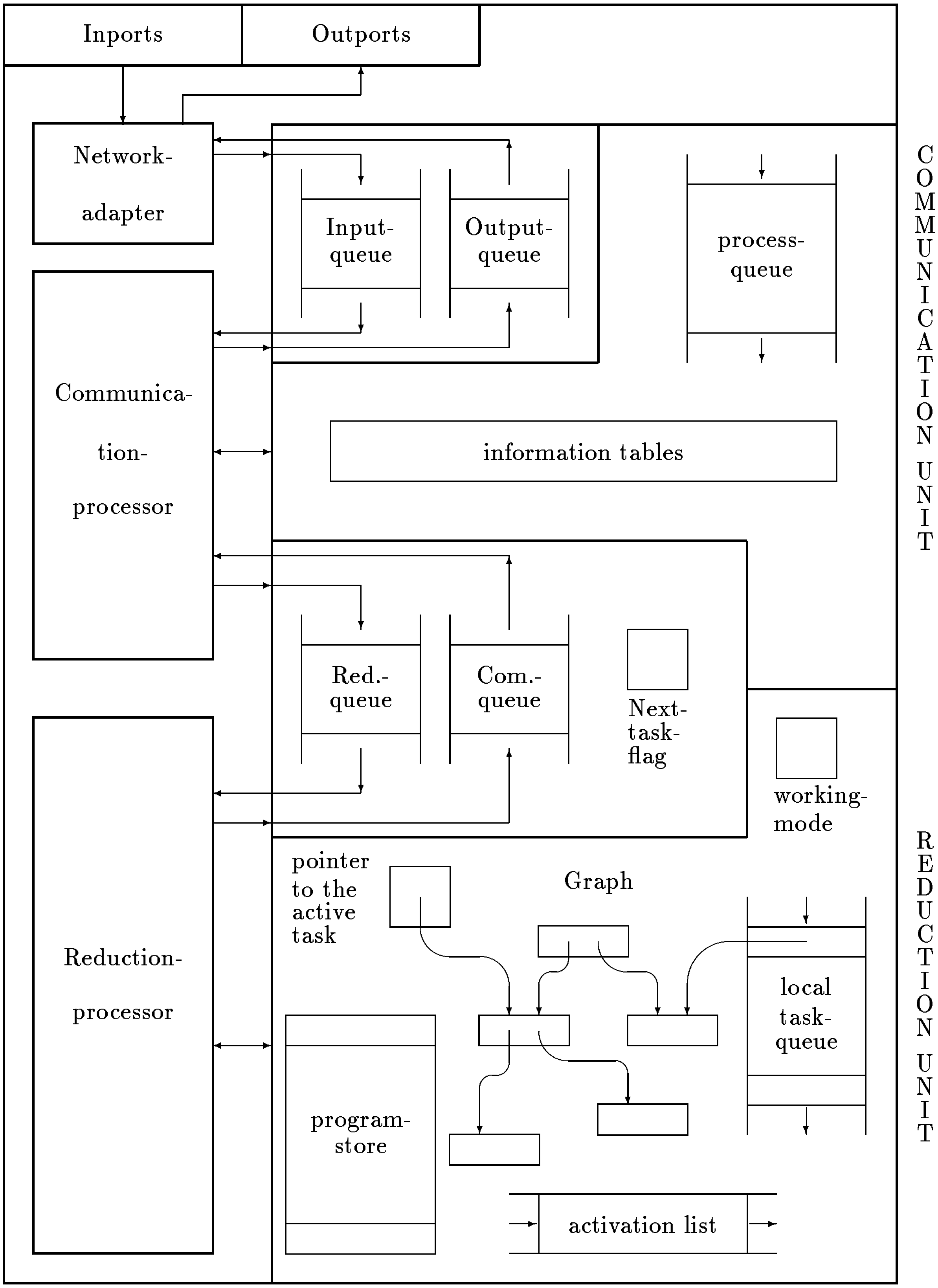

Figure 2: Structure of a Processor Element

- *process messages* to distribute the parallel processes among the processor elements,
- *answer messages* to communicate the result of distributed parallel processes and the "value" of global subgraphs,
- *request messages* to ask for nodes of the program graph that are allocated on other processor elements and
- *activation messages* to activate the evaluation of subgraphs on other processor elements.

A parallel process corresponds always to a combinator application. Thus it is completely specified by the combinatorname, the list of arguments — where basic values are directly given and complex arguments are represented by pointers —, the evaluator with which the parallel task has to be evaluated, the return-address (the address where the result of the task has to be sent to) and finally the kind of activation, which will be explained later:

[PROCESS, combinatorname, argument-list, evaluator, return-address, kind of activation]

Answer messages consist of the destination address where the answer has to be sent to and the result which is a terminal node of the program graph[1]. Graphs are transferred node-wise between processor elements.

[ANSWER, address, terminal graph node]

Request messages contain two addresses — the result-address where the answer to the request must be sent to and the address of the graph node whose content is needed. Additionally, the evaluator which can be used for the evaluation of the requested node (if it is not a terminal node) is given.

[REQUEST, address, returnaddress, evaluator]

Activation messages are similar to request messages except that they contain no return-address, as no result of the initiated evaluation needs to be returned.

[ACTIVATE, address, evaluator]

4.3 Process Management

The communication processor maintains in its local store a process queue, that contains process messages (from other processor elements and from the reduction unit) (see Figure 2). Process messages represent the parallel activated, but not yet started processes in the system.

A process is passed to the reduction unit only if this unit has set the nexttask flag. As soon as a reduction unit gets a process, execution of this process starts and the process is never migrated again.

Parallel subprocesses that are generated in the reduction unit are passed to the communication processor which will decide whether to keep them in the own process queue or to distribute them among the neighbour processors[2]. To make this decision the processor should take into consideration which processors are capable of performing the task, where the arguments of the task are allocated, the workload of the neighbours, as far as it is known, and the length of the own process-queue. To provide such information the communication processor contains certain information tables in its local store. In certain time intervals dynamically changing information as for example the workload of processors will be communicated in special administration messages that are exchanged between the communication processors.

On the level of the abstract machine we will not go further into these issues. We abstract from the algorithm that is used for work distribution and workload balancing, as appropriate algorithms will also depend on the special structure of the interconnection network.

[1] Note that only terminal nodes of the program graph are copied and sent in answer messages.

[2] Neighbour processors are physically connected processors.

5 Organization of Graph Reduction

In the parallel abstract machine program execution is done by the reduction units. The implementation technique is *distributed programmed graph reduction*, i.e.:

- The program is represented as a graph that will be transformed during execution. In fact, the nodes representing combinator applications correspond to the activation records of conventional implementations.
- For each $\langle$ serial combinator, evaluator $\rangle$-pair a machine code sequence is generated that controls the reduction process.
- The program graph will be distributed among several processor elements.

5.1 Graph Representation

The distinction between first-order and higher-order applications in the syntax of applicative expressions has been made to provide a much more efficient graph representation of these expressions. The standard technique in the literature is to work with curried functions and to represent them by binary graphs, as e.g. in the G-machine [Johnsson 84]. This requires *unwinding* the sequence of apply-nodes to determine the function symbol before any application can be reduced. Due to the special syntactic structure of our typed applicative expressions we get a more efficient graph representation of applications, such that a process like the unwinding in the G-machine is avoided whenever possible.

Complete applications $\Phi(e_1, \ldots, e_n)$ are represented by so-called *task nodes* of the following structure:

TASK	Φ	e_1 / $\cdots$ / e_n	status-information

Task nodes are similar to the vector-apply nodes, which are used in an optimized version of the G-machine, which has been described in [Johnsson 87]. Vector-apply nodes however do not contain status-information. The status-information will be explained later.

Partial applications $\Phi(e_1, \ldots, e_k)$ with $k < n$, where n denotes the arity of Φ, are represented by *incomplete-task-nodes*:

INCOMPLETE-TASK	Φ	e_1 / $\cdots$ / e_k / ? /$\cdots$ / ?	$n-k$

Because of the full laziness of serial combinator systems copying partial applications does not unshare computations. Thus incomplete-task nodes are treated as terminal nodes.

Higher-order applications $\text{ap}(e, e_1, \ldots, e_m)$ do not require an explicit graph representation. This is because of the special treatment of (non-strict) arguments whose evaluation must be suspended.

The function name in a task node may be a combinator name, a basic function or the special tag 'arg' that indicates tasks that are generated for the delayed evaluation of arguments. For each of these symbols machine code is contained in the program store of each reduction unit. The arguments e_i in task nodes are directly given, if they are basic values. Otherwise the task node contains a pointer to the graph representation of the argument which may be an argument node.

As the program graph will be distributed among several processor elements by the migration of tasks and by request- and answer-messages, a global address-space which is composed of disjoint local address-spaces is required. Each global address consists of two components:

$$\text{global address} = \langle \text{ processor number, local address } \rangle.$$

The organization of the parallel abstract machine is completely decentralized. For this reason all the status-information that is necessary for the reduction of a task is stored in the task node. If a task is not yet activated the status-information consists only of the evaluator ξ_0. The status of an activated tasks has the following form

$$\text{status} = \langle \text{ev}, \text{ip}, \text{d}, \text{lv}, \text{pc}, \text{lq}, \text{gq} \rangle,$$

where

ev is the evaluator of the task,

ip is the instruction pointer, which indicates the next instruction that must be executed,

d denotes the data stack, which is used for the execution of basic functions[3];

lv stands for storage for the local variables and pointers to parallel subtasks in the case of a combinator task; this storage is organized as a stack,

pc is a pending count, which indicates the number of results a suspended task has to wait for,

lq is a list of the addresses of local tasks that are waiting for the result of the task and

gq is a list of global addresses, where the result of the task must be sent to.

The entries in the argument list, on the data stack or on the stack of local variables are basic values, local addresses or global addresses. As usual, tags are used to distinguish the different kinds of values. For addresses the evaluator which can be assumed for the subgraph with this address is additionally noted. This information is especially important for global addresses as it can be used to avoid multiple activations with the same evaluator.

In the original G-machine [Johnsson 84] arguments whose evaluation must be delayed were represented by graphs which were interpreted in the case of evaluation. In our machine we follow the approach taken in [Fairbairn, Wray 87] and represent suspended expressions by special so-called *argument nodes* which correspond to closures:

ARGUMENT	environment	pointer to code

The environment within argument nodes consists of the argument list and the local variable stack of the task node that created the argument node. If the argument node represents a data structure, three different code addresses corresponding to the possible evaluators are given for the evaluation of the node. Otherwise only one address needs to be noted within the node.

In the case of evaluation the argument node will be overwritten with a task node with the function tag 'arg'. The argument list and the stack of local variables will be copied from the argument node. The instruction pointer will be initialized with the code address given for the actual evaluator.

Thus the evaluation of suspended arguments will still be controlled by code. Especially no graph representation for **case**- and **letpar**-expressions or higher order applications is needed.

Structured data objects are represented by *data nodes* that contain the data constructor, the components of this structure and the evaluator, which indicates the amount of evaluation that has already been done on the structure.

SDATA	constructor	components	evaluator

Basic values can be stored in basic data nodes:

[3] The length of the data stack is statically bounded because tasks themselves are not recursive. Recursive combinator calls result in the activation of new tasks.

BDATA	basic value

These are e.g. necessary if the result of a task is a basic value, as a task node is — after termination of the task — overwritten with its result.

Data nodes and incomplete task nodes are terminal graph nodes that may be contained in answer messages.

If a task is activated and sent to another processor element for execution, its original task node is overwritten by an *indirection node*. The indirection node indicates that the value of this subgraph will be computed on some other processor element, which is usually not known. It contains the new address of the subgraph, if this is known, the evaluator of the task and lists of local and global addresses to note the places where the result of this graph has to be communicated.

INDIRECTION	address	evaluator	lists of addresses

The address of the indirection node is the return-address of the distributed task and thus the result of the task will be sent back to the indirection node.

An indirection node is also generated, when a request to a global address is produced. The indirection node becomes the place-holder for the result that will be returned in an answer message.

If a task tries to access an indirection node, it is suspended — its pending count is incremented — and the address of the task node is noted in the list of local addresses within the indirection node. The pending count of the task will be decremented, when the indirection node is over-written by a terminal node.

If there is a request-message to an indirection node the address where the answer to the request should be sent to, is noted in the list of global addresses within the indirection node. The request message will be answered as soon as the indirection node is over-written.

For technical reasons we also need local indirection nodes which simply consist of the corresponding tag and a local address. They are sometimes used to overwrite task nodes after termination of the task by an indirection to the result.

5.2 State of a Reduction Unit

The local store of each reduction unit consists of six components:

$$\langle \mathrm{m}, \mathrm{atp}, \mathrm{G}, \mathrm{ltq}, \mathrm{al}, \mathrm{p} \rangle$$

m indicates the *working mode* of the reduction unit. We distinguish four different modes:

- In the *communication mode* (cm) messages from the communication unit are processed.
- In the *reduction mode* (rm) combinator reductions are performed by executing machine instructions.
- In the *activation mode* (am) the further evaluation of the components of data structures, whose evaluator has been incremented is initiated.
- In the *wait mode* (wm) the reduction unit waits for new processes to be passed by the communication processor.

atp denotes the active task pointer, which indicates the task that is presently executed. Each task is represented by a node of the program graph. Thus the active-task-pointer is just an element of the local-address-space pointing to a node of the local graph. In each local graph there is at most one active task. If the reduction unit is running idle, the active task pointer has the value 'nil'.

G denotes the *graph* component which is a mapping from the local addresses into the graph nodes, which have been explained in the previous section.

ltq denotes the *local task queue* which contains pointers to local active tasks. On the one hand, these are subtasks of local tasks that will be executed locally. These subtasks correspond to subroutines that are not distributed. On the other hand, these are tasks that were suspended because some argument value or the result of some subtask was not yet available, but that can now be reactivated because the missing result has arrived. A task will be reactivated if its pending count becomes zero.

al denotes the activation list, which contains pairs of pointers to local structured data nodes and evaluators. Each entry represents a data structure whose evaluator has been increased. In the activation mode the reduction unit processes this list and initiates the further evaluation of the components of the data structure.

p denotes the program-store that contains the translations of the combinators and sequences of machine instructions for the reduction of basic function applications and higher-order applications.

The state of a reduction unit is determined by the state of the local store and additionally by the state of the shared memory of the reduction unit and the communication unit. This shared memory contains three components:

$$\langle \text{ red-q, com-q, next } \rangle.$$

red-q is the queue of (reduction) messages that are passed from the communication processor to the reduction unit.

com-q is the queue of reduction messages that are generated by the reduction unit and passed to the communication unit.

next is the boolean nexttask-flag that can be set by the reduction unit to ask for new processes.

5.3 Machine Code

We distinguish four classes of machine instructions, namely

- data stack instructions:
  - EXEC f ($f \in \Omega$) — Execute base function f
  - NODE (c, ξ) ($c \in \Gamma$) — Construct a structured data node,
- control instructions:
  - JMP l — Jump to label l (l is a program-address)
  - JPFALSE l — Jump to l if “false” is on top of the data stack
  - CASE $\langle (c_1, l_1) \cdots (c_k, l_k) \rangle$ — Jump to l_i if the top of the data stack represents a structure with constructor c_i,
- graph instructions:
  - LOAD i — Load the i-th argument of the active task on the stack.
  - LOADLOC i — Load the i-th element of the local variable stack of the active task on the data stack
  - SPLIT — Load the components of the data structure represented by the top of the data stack on the local variable stack

- STORE m — Push the m elements on top of the data stack on the stack of local variables
- POP n — Delete n elements from the stack of local variables.
- ARGNODE(l_1, l_2, l_3) — Create an argument node
- MKNODE(ϕ,i) — Create a task node, where ϕ is a combinator or basic function and i $\in \mathcal{N}$ gives the number of arguments that are available on the stack; the node that is constructed has status dormant (evaluator ξ_0) if enough arguments are given, otherwise an incomplete task node is generated;
- APPLY — adds arguments to an incomplete task node

and

- process instructions.
  - EVALUATE ξ — Activate a *local* subtask with evaluator ξ
  - ACTIVATE ξ — Activate a *parallel* subtask with evaluator ξ
  - INITIATE ξ — Initiate the evaluation of the top element of the data stack
  - INITARG/LOC (i,ξ) — Initiate the evaluation of the i-th argument /the i-th element of the local variable stack of the active task using evaluator ξ
  - GETARG/LOC (i, ξ) — the same as INITIATE but it is ensured that the result of this evaluation will be locally available
  - WAIT m — Suspend the active task, if one of the m elements on top of the data stack is not yet completely evaluated or not locally available
  - RET — Finish the active task
  - PUSH (F,i) — Perform a tail-recursive combinator call

The first three classes of instructions work only on the local store of the reduction unit. The most important instructions are the *process instructions* which control the evaluation and the local process and task management. They may lead to the generation of reduction messages which will be written into the shared memory component com-q. We distinguish instructions for the activation, suspension and termination of computations.

There are four different mechanisms for the activation of tasks and processes. First of all, a task may be executed *locally* or *in parallel.* Local tasks are initiated by the instructions EVALUATE and INITIATE. Parallel processes are generated by the instructions ACTIVATE and INITARG/LOC and GETARG/LOC, respectively.

A local task activation leads to the extension of the task node by the status-information that is necessary for its execution. The address of the task node is added to the internal task queue. The active task is not necessarily suspended. In the case of a parallel activation a process message is generated and written into the shared memory of the reduction unit and the communication unit.

Additionally, we distinguish *direct* and *indirect* activations. A task will be activated directly (by the instructions EVALUATE or ACTIVATE) if the evaluation of its strict arguments — with respect to the context sensitive evaluation transformer of the corresponding application — has already been initiated. This is always the case when a task is activated immediately after the generation of its task node with the MKNODE-instruction.

A direct activation is not possible for dynamically (by higher order applications) generated task nodes or delayed evaluations. If a task is activated indirectly (by the instructions INITIATE or INITARG/LOC, GETARG/LOC) the execution of the task starts with a special code sequence for the evaluation of the strict arguments — with respect to the context-free evaluation transformer of the function corresponding to the task.

The GET-instructions and the INIT-instructions differ only for global addresses. While the INIT-instructions generate an *activation*-message, the GET-instructions send a *request*-message to a global address to ensure that the contents of the global graph node will be returned (after its evaluation to weak head normal form).

5.4 Compilation Rules

We distinguish the following compilation schemes whose complete formal definitions can be found in [Loogen 88] or [Loogen, Kuchen et al. 89].

- ComTrans $[\![F_i(x_1, ..., x_{n_i}) = e_i]\!]\xi$ gives the translation of a combinator definition for the evaluator ξ.

We generate special code for each combinator/evaluator-combination. The code starts with a sequence of INITARG-instructions that cause the evaluation of the strict arguments of the combinator before the evaluation of the combinator body is started. These instructions will not be executed when a task is directly activated, i.e. activated by an EVALUATE- or ACTIVATE-instruction immediately after the construction of the task node, because, in this case, the evaluation of the strict arguments has already been initiated.

- EvalRet $[\![e]\!]\xi$ lv m is used for the translation of combinator bodies. It generates code for the evaluation of an expression with evaluator ξ and the subsequent termination of the executed task. It is used to handle tail-recursive combinator calls in an optimized way. The parameters lv and m are used to control the local variable stack.

- Eval $[\![e]\!]\xi$ lv m and Init $[\![e]\!]\xi$ lv m give code for the evaluation of an expression e with evaluator ξ.

The scheme Eval is used for the translation of strict arguments of base functions, whose value is locally *needed*.

The scheme Init is used for the translation of arguments of combinators, constructors or other functional expressions. The generated code initiates computations, but does not ensure that the result of the computations will be locally available.

Thus values (graph nodes) are only transferred to processor elements, where they are really needed. The necessity of distinguishing locally needed and strict, but not (yet) locally needed global arguments has been pointed out to the authors by Geoffrey Burn [Burn 88b]. He detected that in a previous version of our machine arbitrarily long chains of indirection nodes could be generated and that consequently, answer messages following these chains could take a roundabout way through the system before they reached their destination. In the current version, there are at most chains of indirection nodes with length two.

The schemes Eval and Init differ only for variables for which the Eval generates a GETARG-instruction and Init generates an INITARG- instruction. Therefore these schemes are defined using the scheme

- ETrans $[\![e]\!]\xi$ lv m ν which has an extra parameter $\nu \in \{\text{eval, init}\}$ to indicate the mode of translation.

We give here only the translation of a **letpar**-expression: (Let $\xi \in \mathbf{Evset}\backslash\{\xi_0\}$ and $\nu \in \{\text{eval, init}\}$.)

$$
\begin{aligned}
&\text{ETrans } [\![\textbf{letpar } y_1 = \tilde{F}_1(e_{11}, \ldots, e_{1l_1}) \textbf{ if } ev_1 \textbf{ and } \ldots\ y_p = \tilde{F}_p(e_{p1}, \ldots, e_{pl_p}) \textbf{ if } ev_p \textbf{ in } e]\!]\ \xi \text{ lv } m\ \nu \\
&\quad := pcode(\tilde{F}_1(e_{11}, \ldots, e_{1l_1}), ET_1^{cs}(e)(\xi), \xi, ev_1, \text{lv}, m) \\
&\qquad\quad \ldots \\
&\qquad\quad pcode(\tilde{F}_p(e_{p1}, \ldots, e_{pl_p}), ET_p^{cs}(e)(\xi), \xi, ev_p, \text{lv}, m) \\
&\qquad\quad \text{STORE } p; \\
&\qquad\quad \text{ETrans } [\![e]\!]\xi \text{ lv}[y_1/m+p-1, \ldots, y_p/m]\ m+p\ \nu;
\end{aligned}
$$

POP p;

where $pcode(\tilde{F}_j(e_{j1},\ldots,e_{jl_j}),\xi^j,\xi,ev_j,\mathrm{lv},m)$

$$:= \begin{cases} \textsc{Init}[\![e_{j1}]\!]ET_1^{\mathrm{cs}}(\tilde{F}_j(e_{j1},\ldots,e_{jl_j}))(\xi^j)\ \mathrm{lv}\ m;\ldots\textsc{Init}[\![e_{jl_j}]\!]ET_{l_j}^{\mathrm{cs}}(\tilde{F}_j(e_{j1},\ldots,e_{jl_j}))(\xi^j)\ \mathrm{lv}\ m; & \\ \text{MKNODE }(\tilde{F}_j,l_j);\text{ ACTIVATE }\xi^j; & \text{falls } \xi \geq ev_j \\ & \\ \textsc{Init}[\![\tilde{F}_j(e_{j1},\ldots,e_{jl_j})]\!]\xi^j\ \mathrm{lv}\ m; & \text{falls } \xi < ev_j \end{cases}$$

and $ET^{\mathrm{cs}}(e)$ denotes the context-sensitive evaluation transformer of the sequential thread

- DELAY $[\![e]\!]$ lv m is used to handle "lazy evaluation". It generates a representation of expressions in non-strict argument positions whose evaluation must be delayed.

Below we show the machine code sequence that is generated for the combinator 'findsolutions' of our example program of section 3 with evaluator ξ_3.

$\mathrm{ca_{indir}}$: INITARG (3,ξ_1);
$\mathrm{ca_{dir}}$: GETARG (3,ξ_1); LOAD 3; LIT 8; WAIT 2; EXEC =;
JPFALSE l_F; LIT LNIL; JMP l_{END};
l_F: LOAD 1; LOAD 2; GETARG (3,ξ_1); LOAD 3; LIT 1; WAIT 2; EXEC +;
MKNODE(findsolutions,3); ACTIVATE ξ_3;
INITARG (1,ξ_3); LOAD 1; LOAD 2; LOAD 3;
MKNODE(complete,3); ACTIVATE ξ_3; STORE 2;
INITLOC (1, ξ_3); LOADLOC 1; INITLOC (0, ξ_3); LOADLOC 0; PUSH(append,2);

The schematical translation of combinator systems generates of course a lot of unnecessary instructions as e.g. the INITLOC-instructions in the code for findsolutions. These can be detected and eliminated. We will not go further into this issues.

6 Implementation Aspects

The parallel abstract machine has been implemented on an OCCAM-Transputer system, where one processor element of the abstract machine runs on one transputer. Before we will discuss our first experimental results in the next section, we want to give an overview on special aspects of this implementation, which have not been handled on the level of the abstract machine.

6.1 Asynchronous Communication

All the processing units of PAM work completely asynchronously. Unfortunately, the language OCCAM only supports synchronous hand-shake communications. Thus we have to simulate asynchronous message passing by always buffering messages to ensure that no processor is ever blocked because it has to wait for a communication.

The buffers have a highly decentralized structure. Each buffer consists of a finite number of one-element buffers that work in parallel. Access to the buffers is controlled by the processors using the buffers by simple counters indicating the buffer cell which can be read or written next. Thus it is always possible for two processors at the same time to write a message into the buffer and to read a message from the buffer,respectively.

The structure of the processor elements was slightly changed to optimize the message traffic within a processor element, because it is not possible to run the communication unit and the reduction unit on separate transputers. The different processors are implemented by independent OCCAM-processes which communicate via buffered channels.

The communication unit now contains three independent processors:

- The *process manager* is responsible for process distribution and workload balancing.
- The *network-adapter-in* collects the incoming messages from the inports and forwards them to the reduction unit, the task manager or to the network-adapter-out, if messages must be routed through the network.
- The *network-adapter-out* is responsible for the routing of messages. It passes messages from the reduction unit, the processmanager and the network-adapter-in to the correct outports.

We use two processors for the adaption to the interconnection network in order that receiving messages from the inports and sending messages via the outports can be done in parallel. In the realization of the abstract machine on transputers we cannot assume a complete interconnection of the different processor elements. Therefore messages must be routed through the system.

The reduction unit now sends answer, request and activation messages directly to the network-adapter-out which sends them to their destination. Process messages are always send to the process manager which is responsible for their distribution.

6.2 Routing of Messages

In general, the interconnection network does not provide a full interconnection of all transputers. Thus messages must be routed through the network. This work is done by the network-adapter-out. In the current implementation we use a very simple static routing scheme. Each network-adapter-out has a routing table in a local store that gives for each processor element the number of the outport (link) into which a message for this processor element has to be written. Of course, the routing table fixes a shortest way between each pair of processor elements. The advantages of this method are that routing can be done very efficiently and that the order of messages is never changed (although this is not necessary in our machine).

6.3 Process Distribution

The process manager in the OCCAM implementation corresponds to the communication processor of the parallel abstract machine. Its main task is the process organization. One can coarsely distinguish two different approaches to process distribution, namely passive or active distribution.

- *Active* distribution of processes means that the process manager may decide to send processes that have been generated by the reduction unit to neighbour processor elements without that these processor elements have asked for work.
- *Passive* process distribution means that processor elements which have no work have to ask the neighbour processor elements to send them processes.

Active process distribution schemes seem to provide a better balancing of the workload and a more appropriate distribution of the program graph, especially in the presence of complex data structures. We intend to test various process distribution schemes in our implementation. In fact we want to support some combination of active and passive process distribution, as it seems that an active scheme should be prefered when there are less processes than processors while an passive scheme is advantageous in the other case. Especially in the beginning and before the termination of the overall computation the workload will be low. The passive scheme produces an enormous communication overhead in these situations, because the idle processors keep on sending workrequests to their neighbours.

In our current implementation we have implemented a very simple passive scheme. Processors, which have no work and run idle, have the possibility to send *workrequest messages* to locally connected processor elements. These messages may then be answered by process messages if the addressed processor has enough work and decides to distribute some processes. Otherwise they are answered by *nowork messages.*

[WORKREQUEST, processor number] and [NOWORK, processor number]

In order to prevent the idle processors from sending workrequests when the workload is low and there is no more work to distribute, we use a clock. Whenever a processor only gets nowork-answers to its workrequests, it sets an alarm clock and sends no more workrequests until this clock has run down.

6.4 Garbage Collection

In the abstract machine we completely abstracted from garbage collection in the programm graph. In an implementation this is of course not possible. We use weighted reference counting [Bevan 87] for garbage collection in the distributed program graph. For its realization special decrement messages are necessary to decrement the reference count of graph nodes on other processor elements.

7 Applicative Data Structures

An important topic of our project is the development and comparison of different applicative (i.e. side-effect-free) data structures. In the following we concentrate on the structures *list* and *sequence*. Sequences are very similar to lists but allow to insert new elements in arbitrary positions and to delete and access arbitrary elements more efficiently. They are implemented using binary *inhomogeneous* trees, i.e. the elements of the structure are stored in the leaves. The internal nodes are marked by the length of the subsequence represented by the corresponding subtree. We use inhomogenous trees since they are easier to append and to traverse in a divide and conquer style, which is essential for a parallel implementation.

Consequently, a sequence can be divided into subsequences in O(1), an update (insertion or deletion) can be done in O(log n) in the average case. The updated structure and the original one are stored in an overlapping manner. For each update only a path from the new root to the changed leaf needs to be build. The rest of the structure can be shared. In order to search the i-th element of a sequence, i has to be compared with the length information of the left son for each node on the search path down to a leaf.

We consider unbalanced sequences and height balanced sequences based on AVL-trees, which have been proposed by Myers [Myers 84].

8 Experimental results

The machine code of the parallel abstract machine is interpreted, which might explain the absolute times needed by our simple example programs (see Figures 3 and 4). Until now, we have run experiments on 1, 2, 4, and 12 transputers. Two examples work on integer numbers (nfib and one) and the other examples use data structures (quicksort, towers of Hanoi, the queens problem, matrix multiplication, fold and map) (see appendix).

For the examples with integer numbers, we got good speedups as can be seen from Figure 3. In both examples the grain of parallelism was too fine. Very few of the generated processes were sent to another processor, most of them returned to their origin. Because of this, we modified the PAM in such a way that process messages were only generated when a workrequest from another processor arrived. A second task queue, containing only pointers to those processes, was added to the reduction unit. With this modification, it was possible to save about 75% of time.

For programs working on data structures, we get reasonable speedups up to 8 or 9 with 12 processors. The workload balancing for all problems except quicksort is good. Each processor executes approximately the same number of PAM-commands. After a short starting phase, all processors work continuously, until in a short final phase the results are collected.

				on 1 processor		on 12 processors		
problem	size	structure	#com	sec	# nodes	sec	# messages	speedup
one	13		238K	12.36	0.1K	1.21	0.1K	10.21
one	15		950K	49.41	0.1K	4.31	0.3K	11.46
one	17		3801K	197.62	0.2K	16.69	0.4K	11.84
nfib	20		317K	16.48	0.1K	1.55	0.2K	10.63
nfib	22		831K	43.14	0.1K	3.80	0.3K	11.35
nfib	24		2176K	112.94	0.1K	9.65	0.4K	11.70
map	500	L	44K	2.41	7.0K	2.30	3.0K	1.05
map	500	U	99K	5.35	4.2K	1.02	5.7K	5.25
map	500	B	135K	7.10	6.2K	1.54	9.8K	4.61
map	1000	L	88K	4.80	14.0K	4.58	6.0K	1.05
map	1000	U	199K	10.69	8.2K	1.93	11.9K	5.54
map	1000	B	270K	14.18	12.2K	2.35	22.4K	6.03
map	1500	L	132K	7.20	21.0K	6.81	9.0K	1.06
map	1500	U	298K	16.04	12.2K	2.44	19.2K	6.57
map	1500	B	407K	21.34	18.3K	3.63	31.6K	5.88
map	2000	L	176K	9.59	28.0K	9.09	12.1K	1.06
map	2000	U	398K	21.38	16.2K	3.01	23.9K	7.10
map	2000	B	541K	28.35	24.3K	4.17	40.0K	6.80
map	2500	L	220K	12.00	35.0K	11.36	15.1K	1.06
map	2500	U	497K	26.72	20.3K	3.61	28.3K	7.40
map	2500	B	678K	35.50	30.3K	5.29	47.2K	6.71
fold	2500	L	231K	12.82	20.0K	12.82	0	1.00
fold	2500	U	437K	25.28	30.2K	3.65	26.6K	6.93
towers	11	L	578K	35.09	15.6K	12.13	5.9K	2.89
towers	11	U	292K	15.00	8.4K	2.05	0.2K	7.32
qsort	700	L	889K	48.76	14.6K	24.04	36.0K	2.03
qsort	700	U	1727K	99.23	20.2K	24.69	76.3K	4.01
matmult	10	L	761K	47.07	1.8K	6.54	51.5K	7.20
matmult	10	U	1291K	71.05	2.4K	9.15	72.0K	7.76

size : problem size
structure : data structure used:
L : strict lists, U : unbalanced sequences, B : height-balanced sequences
com: total number of executed PAM-commands
sec: runtime in seconds,
nodes: number of graph nodes; each node needs 16 bytes
messages: total number of messages

Figure 3: Experimental results

#processors		1	2	4	12
problem	size	sec	speedup	speedup	speedup
map	2500	26.72	1.75	2.99	7.40
fold	2500	25.28	1.74	3.08	6.93
towers	12	30.07	2.03	3.95	8.92
qsort	700	99.23	1.94	2.88	4.01
queens	8	428.00	1.89	3.61	9.44
matmult	10	71.05	1.81	3.18	7.76

Figure 4: Speedup for problems on unbalanced sequences

Unbalanced sequences behave better than balanced ones in all our examples, since they are automatically balanced by the divide and conquer algorithms which produced them. Explicit balancing only causes overhead. This can be seen for the map-function in figure 3.

Lists are easier to traverse then sequences, since they consist of less nodes. Because of this, algorithms which are based on traversing like map and fold are more efficient on lists on one processor. But lists can only be handled sequentially. So, sequences are better on many processors, since they allow a divide and conquer style traversal. Sequences are also easier to append (in O(1)). Since this is the only operation needed in the towers of Hanoi example, sequences are better here even on one processor.

Lists are better, where small structures up to 10 or 15 elements are needed, since building an inhomogenous tree is more expensive than building a list. This overhead cannot be made up on such small structures by a divide and conquer style algorithm or by a more efficient access operation (see queens and matmult in figure 3).

Map and fold are recursive higher order functions. The same functional parameter is passed as an argument in each recursive call until the base case is reached. Then, the corresponding process accesses the representation of this parameter, in general via a message. Since the base case is reached n times for a tree with n leaves, this function is accessed n times. This causes an unnecessary communication overhead, especially for the processor element where the function representation is stored. In fact, this processor element sequentializes all these accesses. To avoid this problem, we have sent (the root of) each argument, in which a function is strict together with the corresponding task. Following recursive calls refer to these copies of the original arguments.

A similar problem occurs in the matmult example. The roots of the matrix representations are accessed n^2 times (n being the size of the matrices). The technique mentioned above doesn't help, since there remain $n^2/2$ accesses to each son of the roots, $n^2/4$ to their grandsons and so on. In the matmult example, the processing elements which store those critical nodes (and their neighbor processing elements) execute up to 30% less PAM-commands than the others. This problem occurs, since our matrix multiplication algorithm does not take into consideration the structure of the matrix (representations). Another algorithm could work better. Nevertheless, we are working on solutions of the problem.

Our quicksort implementations perform bad in parallel (see figure 3), since they are not structural recursive. So, more communication is needed, which prevents a good speedup. The algorithm on lists suffers from the additional problem, that dividing a list into two sublists can only be done sequentially.

Figure 4 shows, how the speedup for some programs using unbalanced sequences increases with the number of processors.

9 Related Work

There are a lot of projects underway with the goal of implementing functional languages on appropriate parallel architectures. We mention here only the two projects which are most similar to our work.

A distributed memory architecture for programmed graph reduction is also described in [Burn 88a]. The approach of Burn is based on the same principles as our work. The main differences can be found in the task and process management. Furthermore Burn does not consider serial combinator systems where parallelism has been made explicit using the **letpar**-construct. An implementation of his machine design is not yet available.

Another approach similar to our work has been described in [Raber, Remmel et. al. 88]. Raber and his coworkers developped a parallel version of the G-machine [Johnsson 84] which is much more closer to the original G-machine than our implementation of distributed programmed graph reduction. Implementation results of their machine are also not yet available.

10 Conclusion

Our experiments with the implementation of the parallel architecture are still in a preliminary stage. It is too early to draw final conclusions. Our first experience with the handling of data structures is that some optimizations should be done on the message traffic to reduce the total number of messages. This could be done by introducing an address translation table within each processor element that indicates which global addresses correspond to which local addresses, or by bundling request messages with the same destination that must be routed. We will report on these optimizations and further implementation results in forthcoming papers.

Acknowledgements

This research benefitted from lots of interesting discussions with Geoffrey Burn. In particular, Geoffrey pointed out that there could be arbitrarily long chains of indirection nodes in an earlier version of our parallel abstract machine design, and showed us how such chains can be avoided.

References

[Bevan 87] D.Bevan: *Distributed Garbage Collection Using Reference Counting*, in: Lecture Notes in Computer Science, Vol. 259, Springer Verlag 1987.

[Burn 87a] G.Burn: *Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages*, in: Lecture Notes in Computer Science 274, Springer Verlag 1987.

[Burn 87b] G.Burn: *Abstract Interpretation and the Parallel Evaluation of Functional Languages*, Ph.D. Thesis, Imperial College, London 1987.

[Burn 88a] G.Burn: *Developing a Distributed Memory Architecture for Parallel Graph Reduction*, CONPAR 88.

[Burn 88b] G.Burn : *personal communication* at the Aspenäs Workshop on Implementation of Lazy Functional Languages, Göteborg, Sept. 1988.

[Fairbairn/Wray 87] J.Fairbairn, S.Wray: *TIM — A Simple Lazy Abstract Machine to Execute Supercombinators*, in: Lecture Notes in Computer Science 274, Springer Verlag 1987.

[Hudak/Goldberg 85] P.Hudak, B.Goldberg: *Efficient Distributed Evaluation of Functional Programs Using Serial Combinators*, IEEE Transactions on Computers, Vol. C-34, No. 10, October 85.

[Johnsson 84] Th.Johnsson: *Efficient Compilation of Lazy Evaluation*, SIGPLAN Notices Vol. 19, No. 6, June 1984.

[Johnsson 87] Th.Johnsson: *Compiling Lazy Functional Languages*, Ph.D. Thesis, Chalmers University of Technology, Göteborg 1987.

[Loogen 87] R.Loogen: *Design of a parallel programmable graph reduction machine with distributed memory*, Aachener Informatik-Berichte Nr. 87-11.

[Loogen 88] R.Loogen: *Parallele Implementierung funktionaler Programmiersprachen*, Dissertation (in german), RWTH Aachen, 1988.

[Loogen,Kuchen et al. 89] R.Loogen, H.Kuchen, K.Indermark, W.Damm: *Implementation of Functional Languages on Multiprocessor Systems*, Aachener Informatik-Berichte 1989.

[Myers 84] E.Myers: *Efficient applicative data types*, ACM Symp. on Principles of Programming Languages 1984, 66-75.

[Raber,Remmel et al. 88] M.Raber, Th.Remmel, E.Hoffmann, D.Maurer, F.Müller, H.Oberhauser. R.Wilhelm: *Compiled Graph Reduction on a Processor Network*, GI/ITG Tagung Paderborn, 1988, Informatik-Fachberichte, Springer Verlag.

A Example Programs

We list some of the example programs which we used in our experiments, as serial combinator systems to give the reader an idea of which algorithms have been used for the different problems and which form of parallelization has been taken:

one (n : *int*) : *int* = **if** n = 0 **then** 1 **else letpar** v = one(n-1) **in** one(n-1) $*$ v **fi**
nfib (n : *int*) : *int* = **if** n $\leq$ 1 **then** 1 **else letpar** v = nfib(n-1) **in** v + nfib(n-2) + 1 **fi**

The programs map, fold, quicksort, towers of Hanoi and matrix multiplication will only be given for sequences. It is straightforward to get the corresponding programs for lists. The following base operations are available for sequences:

- seqsel, seqdel, sequpd — to select, delete, update an element of the sequence,
- seqappend — to append two sequences and additionally
- leftseq and rightseq — to decompose a given sequence into two subsequences such that

seqappend(leftseq(S),rightseq(S)) = S for every sequence S.

All basic operations are implemented in a straightforward way.

The example programs are not always the most efficient ones to solve the problem. They serve to show the properties of our abstract machine and the different applicative data structures. To compare lists and sequences, we use nearly the same algorithms for the corresponding example programs (except map and fold).

In our experiments, map has always been called with the successor function as argument, and fold has been called with the addition and the identity function.

```
mapseq (f: int → int, T: intseq): intseq =
      case T of EMPTYSEQ: EMPTYSEQ;
                SEQLEAF(x): SEQLEAF(ap(f,x));
                SEQNODE(L,R,lg): letpar v = mapseq(f,rightseq(T))
                                 in SEQNODE(mapseq(f,leftseq(T)),v,lg) esac
```

```
foldseq (f: int × int → int, g: int → int, T: intseq) : int =
      case T of EMPTYSEQ: error;
                SEQLEAF(x): ap(g,x);
                SEQNODE(L,R,lg): letpar v = foldseq(f,g,rightseq(T))
                                 in ap(f,foldseq(f,g,leftseq(T)),v) esac
```

```
tower (n: int, source,help,dest: int): intseq =
    let move = PAIR(source,dest)
    in if n=1 then SEQLEAF(move)
       else letpar v = tower(n-1, help, source, dest)
            in seqappend(tower(n-1,source,dest,help),seqins(v,1,move)) fi

quicksort (L: intseq): intseq =
      case L of EMPTYSEQ: EMPTYSEQ;
                SEQLEAF(x): L;
                SEQNODE(L,R,lg): letpar v0 = seqsel(L,1)
                                 and    PAIR(L1,R1) = divide(v0,seqdel(L,1))
                                 and    v1 = quicksort(R1)
                                 in seqappend(quicksort(L1),seqins(v1,1,v0)) esac
divide (x: int, S: intseq): intseqpair =
      case S of EMPTYSEQ: PAIR(EMPTYSEQ,EMPTYSEQ);
                SEQLEAF(y): if y ≤ x then PAIR(S,EMPTYSEQ) else PAIR(EMPTYSEQ,S) fi
                SEQNODE(L,R,lg): letpar PAIR(L1,R1) = divide(x,leftseq(S))
                                 and    PAIR(L2,R2) = divide(x,rightseq(S))
                                 and    Ln = seqappend(L1,L2)
                                 in PAIR(Ln,seqappend(R1,R2)) esac

matmult (n: int, A,B: seqofintseq): seqofintseq          = matrix(ap(sprod,n,A,B))
matrix (f: int → int → intseq, n: int): seqofintseq      = foldint(1,n,ap(row,f,n),seqappend)
row (f: int → int → intseq, n: int, i: int): intseq      = SEQLEAF(foldint(1,n,ap(f,i),seqappend))
sprod (n: int, A,B: seqofintseq, i,j: int): intseq       = SEQLEAF(foldint(1,n,ap(mult,A,B,i,j),+))
mult(A,B: seqofintseq, i,j,k: int): int                  = seqsel(seqsel(A,i),k) * seqsel(seqsel(B,k),j)
foldint(i,j: int, f: α → β, g: β × β → β) : β =
     if i=j then ap(f,i) else let med = (i+j)/2
                              in letpar v = foldint(i,med,f,g) in ap(g,foldint(med+1,j,f,g),v) fi
```