A New Stack Technique for the Management of Runtime Structures in Distributed Implementations

Guido Hogen and Rita Loogen[*]
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany

Abstract

A new technique for the management of runtime structures in distributed implementations is presented. Instead of storing the runtime stacks of parallel processes as linked lists of function blocks in a heap structure, the local stacks of several parallel processes, which are executed on the same processor element, are stored in an interleaved manner on a single physical stack, called the *meshed stack*. The technique originated from the stack organisation in the well known Warren Abstract Machine (WAM) for logic languages, where the runtime information of several alternative computation branches is stored on a single stack. In principle, the meshed stack technique is independent of the implemented language. We will explain it for the parallel implementation of functional languages. The new technique supports sequential and parallel evaluations equally well.

1 Introduction

In purely sequential implementations of most programming languages a *runtime stack* is used for the control of recursive function or procedure calls. Such calls lead to the allocation of control information (activation blocks) on top of the stack. When the call terminates, this information is removed and control is given back to the calling environment. A graph or heap structure is only used for the representation of structured data.

For parallel implementations, which support dynamic processes, as it is the case in parallel implementations of functional or logic languages, the efficient realisation of the runtime stack is one of the most important problems. The use of a global contiguous array of activation blocks is no longer advisable, because each parallel task or process needs its own stack structure. Furthermore a suspension mechanism must be realized to keep the state of tasks which are waiting for the result of some parallel subtask. This led to the idea of allocating runtime structures in a graph or heap instead of on a stack. The runtime stack of each process is realized as a linked list of activation blocks in the heap [Goldberg, Hudak 86], [Loogen et al. 89], [Augustsson, Johnsson 89], [Kingdon et al. 91]. Spawning of parallel processes yields a tree structure of such linked lists, which represents a so-called *cactus stack*.

A main advantage of this *graph organisation* is the *decentralisation* which is especially suited for distributed systems, as the bottleneck of a global central stack is avoided. Context switches can simply be realized by redirecting the pointer to the currently active process. Note that heap *allocation* can be done as efficiently as stack allocation by the incrementation of a heap pointer [Appel 87]. A disadvantage is however the increased organisational overhead for the *deallocation* (garbage collection) of runtime information. Another disadvantage of the "linked list"-realization of the cactus stack is the overhead that is imposed on the purely sequential evaluations within processes.

[*]email: {ghogen, rita}@zeus.informatik.rwth-aachen.de, Telefax: ++49/241/80 21 229

In this paper we present an alternative realization of the runtime structures which tends to support parallel and sequential evaluations equally well. Instead of representing the cactus stack as a linked tree of activation blocks, we allocate the activation blocks, within each processor element of the parallel system, on a single *meshed stack*, which is organized as a contiguous array, but shared by several processes. The heap is only used for the representation of structured data, closures, suspensions and placeholders for the result of parallel processes. Evaluation is controlled by the stack. During the sequential evaluation of processes the meshed stack is used as in conventional implementations. Activation blocks are allocated on top of it, when function or procedure calls are executed, and removed, when the calls terminate. Central issues of the new stack technique are the generation and suspension of parallel processes.
The meshed stack technique is independent from the implemented language. In this paper we explain it for the distributed implementation of a functional language.

The paper is organized as follows. In the next section, we introduce a simple first order functional language which is sufficient for the explanation of our approach. Section 3 explains a simple sequential reduction machine. Section 4 introduces parallelism information into the functional language and explains the problems that must be solved to provide a distributed implementation of the parallel functional language. Section 5 is devoted to the explanation of the meshed stack technique. In Section 6 we develop a parallel abstract reduction machine. First implementation results are given in Section 7. In Section 8 related work is discussed and finally some conclusions and hints at future work are given.

2 FUN — A Simple FUNctional Language

FUN is a first order monomorphically typed functional language with algebraic data structures. We do not consider higher order functions and polymorphism, as the treatment of these features is independent from our implementation technique.

2.1 FUN-Syntax

Let $\Sigma = \langle S, \Omega \rangle$ be the *base signature* with

- set S of base types (containing the type *bool*) and
- set $\Omega = \bigcup_{(w,s)\in S^* \times S} \Omega^{(w,s)}$ of typed base operation symbols.

$op \in \Omega^{(s_1 \ldots s_n, s)}$ is an operation of type $s_1 \times \ldots \times s_n \to s$. If $n = 0$, op is a constant of type s.
Let $\Delta = \langle D, \Gamma \rangle$ be the *data structure signature* with

- set D of data structure types and
- set $\Gamma = \bigcup_{(w,d)\in (S\cup D)^* \times D} \Omega^{(w,d)}$ of constructor symbols.

We assume that $S \cap D = \emptyset$ and $\Omega \cap \Gamma = \emptyset$. Let $\Gamma^{(d)} = \bigcup_{w\in (S\cup D)^*} \Gamma^{(w,d)}$ denote the set of constructors with destination type $d \in D$.
Let $Var = \bigcup_{s\in (S\cup D)} Var^s$ be a set of typed *variables* and $FS = \bigcup_{(w,s)\in (S\cup D)^* \times (S\cup D)} FS^{(w,s)}$ be a set of typed *function symbols*.
The definition of *Fun-Expressions* $e \in Exp^s$ $(s \in S \cup D)$ is given in Figure 1. A *FUN-program* consists of a finite set of defining rules for the function symbols in FS. Each *defining rule for* $f \in FS^{(s_1 \ldots s_n, s)}$ must have the form:

$$f(x_1, \ldots, x_n) := e,$$

where $x_i \in Var^{s_i}, e \in Exp^s$ and $vars(e) \subseteq \{x_1, \ldots, x_n\}$.

$$
\begin{array}{llll}
e & ::= & x & \%\ x \in \mathit{Var}^s \\
 & | & \phi(e_1, \ldots, e_n) & \%\ \phi \in (\Omega \cup \Gamma \cup FS)^{(s_1 \ldots s_n, s)}, e_i \in \mathit{Exp}^{s_i} \\
 & | & \textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2 & \%\ b \in \mathit{Exp}^{bool}, e_1, e_2 \in \mathit{Exp}^s \\
 & & & \%\ \text{conditional expression} \\
 & | & \textbf{switch } \mathrm{e} \textbf{ of} & \%\ e \in \mathit{Exp}^d (d \in D), \\
 & & \quad c_1\colon e_1; \ldots c_k\colon e_k & \%\ c_i \in \Gamma^{(d)}, e_i \in \mathit{Exp}^s \\
 & & \textbf{end} & \%\ \text{switch expression} \\
 & | & \textbf{proj}(c, j, e) & \%\ c \in \Gamma^{(s_1 \ldots s_n, d)}, e \in \mathit{Exp}^d,\ s_j = s \\
 & & & \%\ \text{projection to } j\text{th component} \\
 & | & \textbf{let} \quad y_1 = e_1 & \%\ y_i \in \mathit{Var}^{s_i}, e_i \in \mathit{Exp}^{s_i}\ (s_i \in S \cup D) \\
 & & \quad\quad \textbf{and} \ldots \textbf{and} & \\
 & & \quad\quad y_k = e_k & \\
 & & \textbf{in } e & \%\ e \in \mathit{Exp}^s \\
 & & & \%\ \text{local definitions}
\end{array}
$$

Figure 1: FUN-Expressions

2.2 Reduction Semantics

Let $\mathcal{A} = \langle A_\perp, \alpha \rangle$ be a strict interpretation of the base signature, i.e. $A_\perp = \bigcup_{s \in S} A^s_\perp$ with flat cpo's $A^s_\perp = A^s \cup \{\perp^s\}$, with $A^{bool} = \{\mathit{true}, \mathit{false}\}$ and $\alpha : \Omega \to \mathit{Ops}(A_\perp)$ is a type respecting mapping of operation symbols to strict operations over $A_\perp$. Let $A = \bigcup_{s \in S} A^s$.
The data structure signature is freely interpreted by the term algebra $\mathcal{T}_\Gamma(A) = \langle T_\Gamma(A), \gamma \rangle$, where $T_\Gamma(A)$ is the set of constructor terms over A and $\gamma : \Gamma \to \mathit{Ops}(T_\Gamma(A))$ maps constructors to their free interpretation.
Computation expressions are variable-free expressions, in which values from A^s may occur as constant symbols of type s. The reduction relation is based on the following reduction rules, which describe local reduction steps:

1. *constant reduction:* $\quad op(a_1, \ldots, a_n) \to a$
   if $op \in \Omega^{(s_1 \ldots s_n, s)}, a_i \in A^{s_i}$ and $a = \alpha(op)(a_1, \ldots, a_n) \neq \perp^s$
2. *conditional reduction:* $\quad \textbf{if } \mathit{true} \textbf{ then } e_1 \textbf{ else } e_2 \to e_1$
   $\textbf{if } \mathit{false} \textbf{ then } e_1 \textbf{ else } e_2 \to e_2$
3. *branching reduction:* $\quad \textbf{switch } c_j(\ldots) \textbf{ of} \ldots c_j : e_j \ldots \textbf{end} \to e_j$
4. *projection reduction:* $\quad \textbf{proj}(c, j, c(e_1, \ldots, e_m)) \to e_j$
5. *function application:* $\quad f(e_1, \ldots, e_n) \to e[x_1/e_1, \ldots, x_n/e_n]$
   if $f(x_1, \ldots, x_n) := e$ is the defining rule for f.

The reduction relation defines the embedding of local computation steps within expressions. Reduction sequences are given by the reflexive, transitive closure of the reduction relation.

2.3 Examples

The following two example programs use the base signature $\Sigma_0 = \langle \{\mathit{int}, \mathit{bool}\}, \Omega_0 \rangle$, where Ω_0 is assumed to contain the usual arithmetic and boolean operation symbols, for which we allow infix notation.
The definition of the function symbol *tower* assumes the data structure *tree* with constructors *leaf* of type $\mathit{int} \to \mathit{tree}$ and *node* of type $\mathit{tree} \times \mathit{int} \times \mathit{tree} \to \mathit{tree}$.

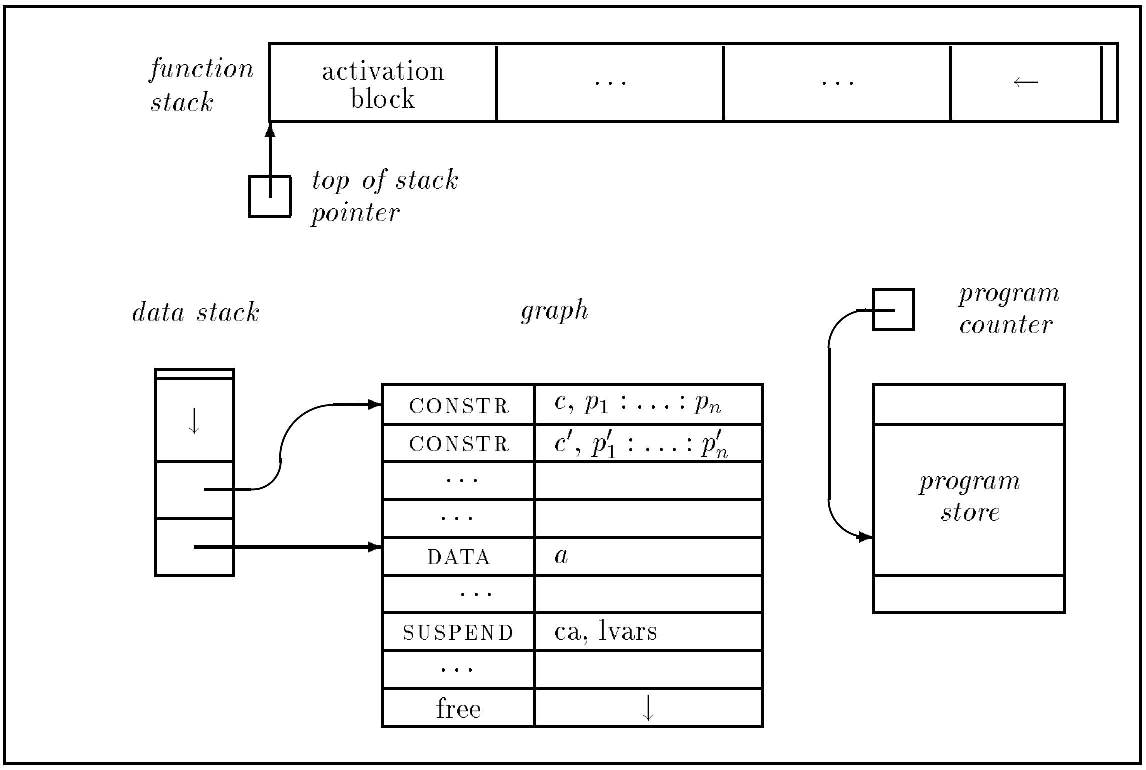

Figure 2: Structure of a Reduction Machine

$$
\begin{array}{lll}
psum(l,h) := & \textbf{if } l = h \textbf{ then } l \textbf{ else} & \textbf{if } l+1 = h \textbf{ then } l+h \\
& & \textbf{else let } m = (l+h)/2 \textbf{ in } psum(l,m) + psum(m+1,h) \\
tower(n, source, help, dest) := & \textbf{let } move = 10 * source + dest & \\
& \textbf{in if } \ n = 1 \textbf{ then } \ leaf(move) & \\
& \qquad\qquad \textbf{else } \ node(\ tower(n-1, source, dest, help), & \\
& \qquad\qquad\qquad\qquad move, & \\
& \qquad\qquad\qquad\qquad tower(n-1, help, source, dest)) &
\end{array}
$$

3 Implementation of Functional Languages

In this section we shortly describe the structure and the behaviour of a simple reduction machine for the implementation of functional languages.

3.1 Structure of the Reduction Machine

The store of the reduction machine is illustrated in Figure 2. Its central component is the function stack which contains the activation blocks of recursive function calls.

Terms, i.e. structured data, basic values, and arguments, whose evaluation is postponed in case of a lazy evaluation strategy, are represented in the graph. Consequently, we distinguish between *constructor nodes*, *data nodes*, and *suspension nodes* (see Figure 2).

Constructor nodes contain the constructor name c and pointers to the graph representation of the components. Data nodes simply contain a value $a \in A$. Suspension nodes contain a code address ca, which is the address, at which the code for the evaluation of the delayed or suspended argument starts, and a list *lvars* of parameters to which some instructions in the code may refer.

3.2 Behaviour of the Reduction Machine

In order to evaluate a function application, the addresses of the graph representations of the arguments are written on top of the data stack and a function call is executed. The instruction CALL (ca, n, k) creates a new function block on top of the function stack, moves n pointers to the function arguments from the data stack to the function stack, reserves k cells for local variables, writes the value of the instruction pointer as return address and the dynamic link to the previously active activation block into the new activation block and finally jumps to the program address ca.
The code for each function f consists of the code for the body expression e of f's defining rule and a RETURN instruction:

$$\textit{Code for } f : \textit{exptrans}(e);\ \text{RETURN}.$$

The return instruction terminates a function call by removing the top level activation block from the stack and jumping to the saved return address. The result of the function call is given on top of the data stack.
The compilation of expressions is shown in Figure 3.
The reduction machine is capable of realizing an innermost and a lazy strategy. Which strategy is chosen to execute a program, depends on the compilation of argument expressions. In case of an innermost strategy, arguments are translated by the *exptrans*-scheme. This implies that expressions are evaluated to normal form and INITIATE instructions can be omitted. In case of a lazy evaluation strategy, arguments are translated by the *delaytrans*-scheme, which generates suspension nodes for the representation of arguments not evaluated yet using the instruction SUSPEND. Furthermore, code for the evaluation of the suspension is generated using the *exptrans*-scheme. This code is concluded by the instruction UPDATE, which writes the result of the evaluation into the suspension node and gives control back to the calling environment. A lazy evaluation strategy evaluates expressions to head normal form, i.e. until the top level constructor of the result is determined. In order to force the complete evaluation of the goal expression, a special mechanism must be incorporated into the machine.
This gives an impression of the principal behaviour of the reduction machine. For simplicity, we have not considered any optimization techniques as e.g. tail recursive function calls.

4 Parallelism in Functional Languages

The *implicit parallelism* within FUN-programs consists of the reduction of independent subexpressions which is possible due to the side effect free nature of the reduction semantics. The eager evaluation strategy allows the parallel evaluation of arbitrary subexpressions. In case of the lazy reduction strategy, strictness analysis can be used to find out subexpressions which can be safely evaluated in parallel. Usually annotations or special syntactic constructs are used to show the implicit parallelism within programs [Hudak, Goldberg 86], [Augustsson, Johnson 89]. An automatic parallelizing compiler for lazy functional languages is described in [Hogen et al. 92].
In the following, we will use the construct:

$$\begin{array}{llcl} \textbf{letpar} & y_1 & = & f_1(e_{11}, \ldots, e_{1n_1}) \\ \textbf{and} & & \cdots & \\ \textbf{and} & y_p & = & f_p(e_{p1}, \ldots, e_{pn_p}) \\ \textbf{in}\ e & & & \end{array}$$

to indicate parallelism in FUN-programs. Such a *letpar*-expression means that the function applications $f_i(e_{i1}, \ldots, e_{in_i})$ *may be* evaluated in parallel with the main stream of evaluation, e. Whether a parallel evaluation really takes place at runtime, depends on the workload of the distributed system. Evaluation of a *letpar*-expression first leads to the parallel activation of the function applications. Then the evaluation of the body e proceeds until the result of a parallel

$$
\begin{array}{ll}
\multicolumn{2}{l}{exptrans(x_i)} \\
:= & \mathrm{LOAD}\ i \\
 & \mathrm{INITIATE} \\
\multicolumn{2}{l}{exptrans(op(e_1, ..., e_n))} \\
:= & exptrans(e_1) \\
 & \vdots \\
 & exptrans(e_n) \\
 & \mathrm{EXEC}\ (op, n) \\
\multicolumn{2}{l}{exptrans(c(e_1, ..., e_n))} \\
:= & argtrans(e_1) \\
 & \vdots \\
 & argtrans(e_n) \\
 & \mathrm{CNODE}\ (c, n) \\
\multicolumn{2}{l}{exptrans(f(e_1, ..., e_n))} \\
:= & argtrans(e_1) \\
 & \vdots \\
 & argtrans(e_n) \\
 & \mathrm{CALL}\ (ca(f), n, loc(f)) \\
\multicolumn{2}{l}{exptrans(\ \mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2)} \\
:= & exptrans(b) \\
 & \mathrm{JMF}\ l_2 \\
 & exptrans(e_1) \\
 & \mathrm{JMP}\ l_3 \\
l_2: & exptrans(e_2) \\
l_3: &
\end{array}
$$

$$
\begin{array}{rl}
\multicolumn{2}{l}{exptrans(\ \mathbf{switch}\ e\ \mathbf{of}\ \ldots c_j : e_j \ldots\ \mathbf{end}\ )} \\
:= & exptrans(e) \\
 & \mathrm{SWITCH}\ (\ldots c_j : l_j \ldots) \\
l_1: & \ldots \\
 & \ldots \\
l_j: & exptrans(e_j) \\
 & \mathrm{JMP}\ l \\
l_{j+1}: & \ldots \\
 & \ldots \\
l: & \ldots \\
\multicolumn{2}{l}{exptrans(\ \mathbf{proj}\ (c, j, e))} \\
:= & exptrans(e) \\
 & \mathrm{PROJ}\ j \\
\multicolumn{2}{l}{exptrans(\ \mathbf{let}\ \ldots y_k = e_k \ldots\ \mathbf{in}\ e)} \\
:= & \ldots \\
 & argtrans(e_k) \\
 & \mathrm{STORE}\ k \\
 & \ldots \\
 & exptrans\ (e)
\end{array}
$$

~~~~~~~

$$
\begin{array}{ll}
\multicolumn{2}{l}{delaytrans(x_i)} \\
:= & \mathrm{LOAD}\ i \\
\multicolumn{2}{l}{delaytrans(c(e_1, ..., e_n))} \\
:= & delaytrans(e_1) \\
 & \vdots \\
 & delaytrans(e_n) \\
 & \mathrm{CNODE}\ (c, n)
\end{array}
$$

otherwise:

$$
\begin{array}{rl}
\multicolumn{2}{l}{delaytrans(e)} \\
:= & \mathrm{SUSPEND}\ l \\
 & \mathrm{JUMP}\ l' \\
l: & exptrans(e) \\
 & \mathrm{UPDATE} \\
l': & \ldots
\end{array}
$$

~~~~~~~

$$
argtrans(M) := \begin{cases} exptrans(M) & \text{in case of an “eager” evaluation strategy} \\ delaytrans(M) & \text{in case of a “lazy” evaluation strategy} \end{cases}
$$

Figure 3: Code for Expressions

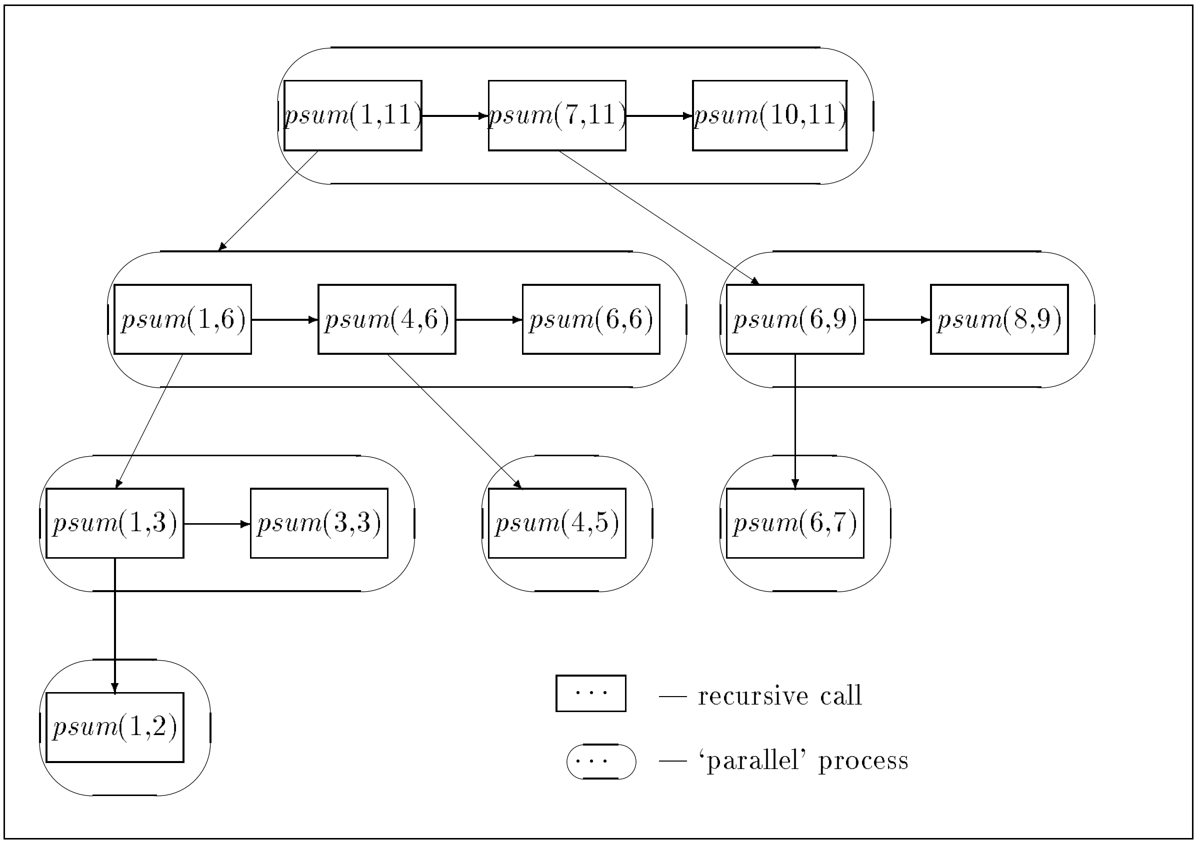

Figure 4: Recursive Calls and Parallel Processes of the *psum*-Program

subexpression, represented by y_j, is needed and not available yet. In this case, the evaluation of e will be suspended until the parallel subprocess yields its result. In the meantime, the processor should be free to evaluate some other process.

The resulting process system is hierarchical. The execution of a parallelized functional program starts with the evaluation of a main expression. *letpar*-expressions generate parallel subprocesses, that can be executed on other processor elements. By the execution of parallel processes, further processes may be generated. When a process terminates, its result is communicated to the father process, i.e. the process that generated the subprocess, and all other processes waiting for the result.

A possible parallelization of the example programs given in Section 2 is:

$$
\begin{array}{l}
psum(l,h) := \ \textbf{if } l = h \textbf{ then } l \textbf{ else } \ \textbf{if } l+1 = h \textbf{ then } l+h \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else let } m = (l+h)/2 \textbf{ in } \textbf{letpar } y_1 = psum(l,m) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } psum(m+1,h) + y_1 \\
tower(n, source, help, dest) := \ \textbf{let } move = 10 * source + dest \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if } n = 1 \textbf{ then } leaf(move) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else letpar } y_1 = tower(n-1, help, source, dest) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } node(\ tower(n-1, source, dest, help), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad move, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y_1)
\end{array}
$$

The process system that will be created for the expression $psum(1,11)$ is shown in Figure 4. It shows already the shape of the cactus stack the parallel system has to cope with. Most of the existing parallel implementations of functional languages generate for each recursive function call a new activation block, called task node or VAP (variable sized application) node, in the

heap [Augustsson, Johnson 89], [Loogen et al. 89], [Kingdon et al 91]. The activation blocks in the heap contain all the information and substructures that are necessary for the evaluation of the corresponding function call. A parallel evaluation can simply be achieved by the migration of a block to another processor element.
The dynamic extension of activation blocks in case of tail recursive function calls can be avoided by the use of the stacklessness analysis of [Lester 89], which allows to create activation blocks large enough to evaluate all tail recursive calls that may occur in a reduction sequence.
In this paper we claim that this "linked list"-realization of the cactus stack is not natural, because it does not support the sequential evaluations within processes in an optimal way. The knowledge about the stack behaviour of the various local stacks is lost and general garbage collection techniques must be used to collect the activation blocks of terminated tasks.
[Goldberg, Hudak 86] already pointed out that the graph allocation of activation blocks should be avoided for purely sequential functions which do not spawn parallel subtasks and may not be suspended due to parallel subtasks. They proposed a hybrid system which assumes an analysis and decomposition of the program into sequential and non-sequential functions. For sequential functions an ordinary stack-based evaluation is arranged. For non-sequential function calls the more expensive allocation of runtime information in the heap must be chosen to support parallel evaluations.

In the following, we will present a new stack technique that is a combination of the graph organisation of parallel processes and the stack management of sequential evaluation sequences. It works for arbitrary parallel programs.

5 The Meshed Stack Technique

From an abstract view point, each parallel process will be executed on a sequential reduction machine, which is capable of the generation of new parallel processes. Thus, each process works on its own function stack area. For sequential recursive function calls, new activation blocks are generated on top of this private function stack. Parallel processes are spawned by generating a special node in the graph, which serves as a placeholder for the result of the process. The characteristic information of the process is embedded into a so-called *process message*, which contains also the address of the placeholder node. This message will be sent to another reduction machine. The placeholder for the result of the parallel process can be seen as a graph representation of the migrated evaluation. For this reason we call it an *indirection node*.
In a concrete parallel system there will, of course, be only a finite number of processor elements. As processes are created dynamically, i.e. there is no restriction on the number of processes, this implies that each reduction machine within a parallel environment must be capable of *multitasking*. It must be possible to evaluate several parallel processes in a single reduction unit. In particular, a machine must be able to organize the interleaved evaluation of several active processes.
The main problem is how to organize the runtime information of several parallel processes in a space- and time-efficient way. One possibility is the pure graph organization which leads to a linked-list realization of local function stacks, which we already rejected. Another possibility is the allocation of private contiguous stack areas within a reserved memory area for the function stack. This approach is very time-efficient, but it leads to a fragmentation of the function stack area and there will be a restriction on the number of parallel processes which can be executed in a single reduction unit or there must be some mechanism to split stack areas if more processes than available stack areas are created. Furthermore there must be some mechanism for the dynamic extension of stack areas, if processes run out of stack space and there is still free space.

Because of these difficulties, we decided to try another approach which originated from the organisation of runtime information in the Warren Abstract Machine (WAM), which has been developed for the implementation of logic languages [Warren 83]. In the WAM the runtime information of several alternative computation sequences is kept on a single stack. This stack is not a stack in the conventional sense. Due to the information that is kept for alternative computation branches, the active activation block (environment) is not always on top of the stack and on termination of "procedure" calls the activation blocks can only be removed from the stack if they are on top of it.

Our meshed stack has similar properties. It keeps the runtime information of several parallel processes on a single stack. Its main property is that the current activation block need not be on top of the stack. The generation of new activation blocks will always be done on top of the stack, but if a function call terminates and its activation block is not on top of the stack, it will not be removed, but simply marked as garbage. If a parallel process is suspended, because it has to wait for some information that is not locally available, its activation blocks are kept on the stack until its evaluation can continue. In the meantime, the reduction machine can proceed with the evaluation of another parallel process, whose activation blocks will simply be allocated on top of the activation blocks of the suspended process. If the new process will eventually be suspended and the process is reactivated, newly created activation blocks of this evaluation will simply be allocated on top of the activation blocks of the previous evaluation. Thus we will get a mixture of activation blocks belonging to different parallel processes on the stack, which is the reason for calling it a meshed stack. This explains the main idea of the meshed stack technique.

In Figure 5 a cactus stack and its implementation with a meshed stack is shown. In the cactus stack each node represents one parallel task including sequential function calls. So each node represents a number of function blocks. Process 1 and 3 are executed in the same reduction machine. If we look at the meshed stack of this reduction machine, we find the function blocks of these processes intermixed.

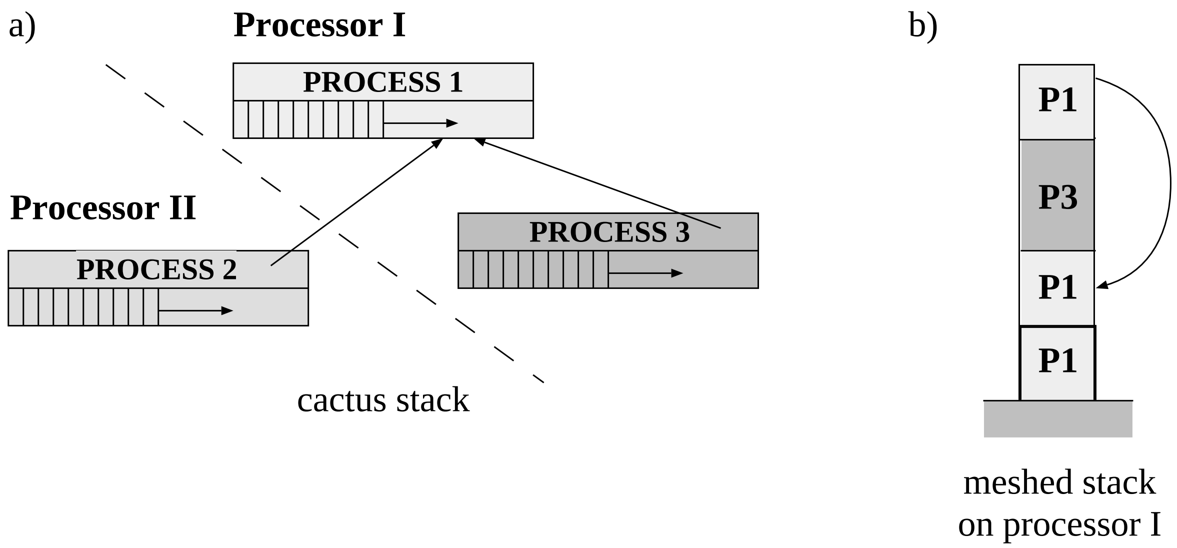

Figure 5: Implementation of a Cactus Stack on a Meshed Stack

But, what happens when function calls terminate and the corresponding blocks can be deallocated. The deallocation of blocks on top of the stack is of course no problem. Inner activation blocks which can be removed are marked as garbage. They are collected, i.e. removed, as soon as there is no valid, i.e. active or suspended function block on top of them. If the stack runs out of space, a special garbage collector is invoked. This garbage collector simply crunches the stack *in place* in one pass. Details will be given in the next section.

6 Development of a Parallel Reduction Machine

In this section we embed the extended sequential reduction machines in a parallel environment and develop a parallel reduction machine.

6.1 General structure

The parallel reduction machine consists of a finite number of identical processor elements, each with a local storage. An interconnection network enables the processor elements to communicate by exchanging messages. Each processor element consists of two independent processing units, which communicate via the exchange of messages in a local shared memory (see Figure 6).

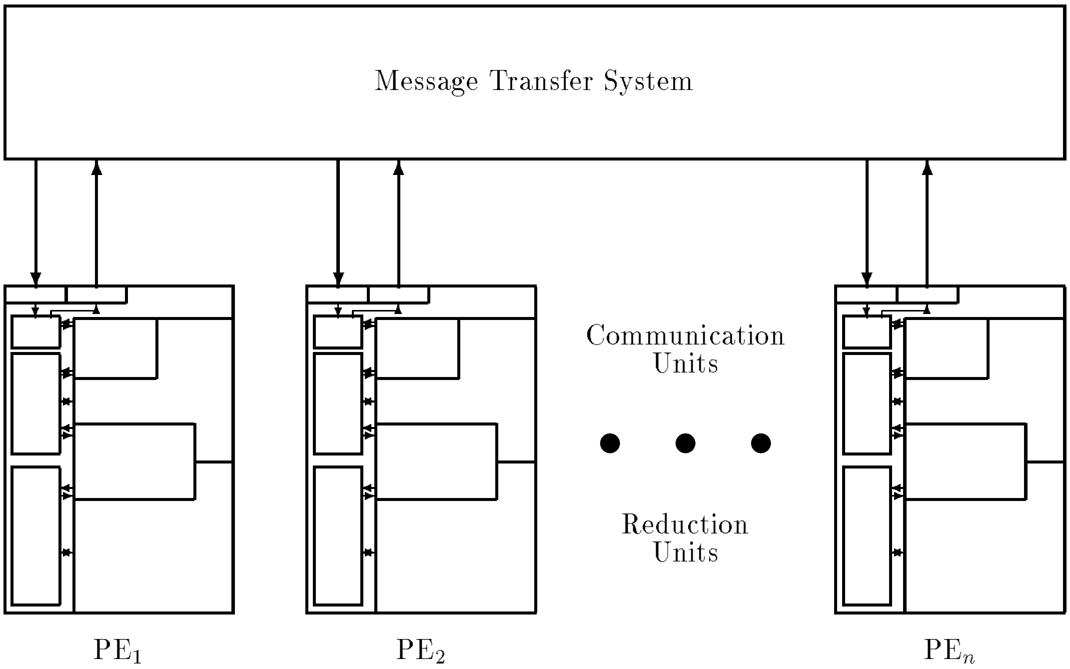

Figure 6: Global Structure of the machine

- The *communication unit* handles the organization of the parallel aspects of the reduction process.
- The *reduction unit* is a sequential reduction machine which has been extended by capabilities for multitasking and for the processing of messages.

The main task of the communication unit is the distribution of parallel processes and the organisation of the message traffic. We will not go into the details of the communication processor, as task distribution and routing strategies are out of the scope of this paper. We assume a simple static routing scheme and a passive process distribution strategy, i.e. processor elements without work ask their neighbouring processor elements for processes. In its local store the communication processor maintains a process queue which contains parallel processes which have been generated by the reduction unit and a static table for the routing of messages from and to other processor elements.

6.2 Process Management in the Reduction Units

The store of the sequential reduction machine is extended by a *ready queue*, which holds the local suspended processes which are ready to be re-activated (see Figure 7). The function stack

is now organized as a meshed stack.
The data stack will be divided into local data stack areas for each activation block. This is necessary to avoid the saving of data stack entries on process switches. Each function call gets an activation block on the function stack and its own "private" data stack. For allocation purposes one has to know the maximum storage demands of each local data stack. These can be estimated during compile time and passed as a parameter in the instructions which generate new activation blocks, as e.g. CALL.
The heap component is extended by placeholder nodes for the results of parallel processes (indirection nodes).

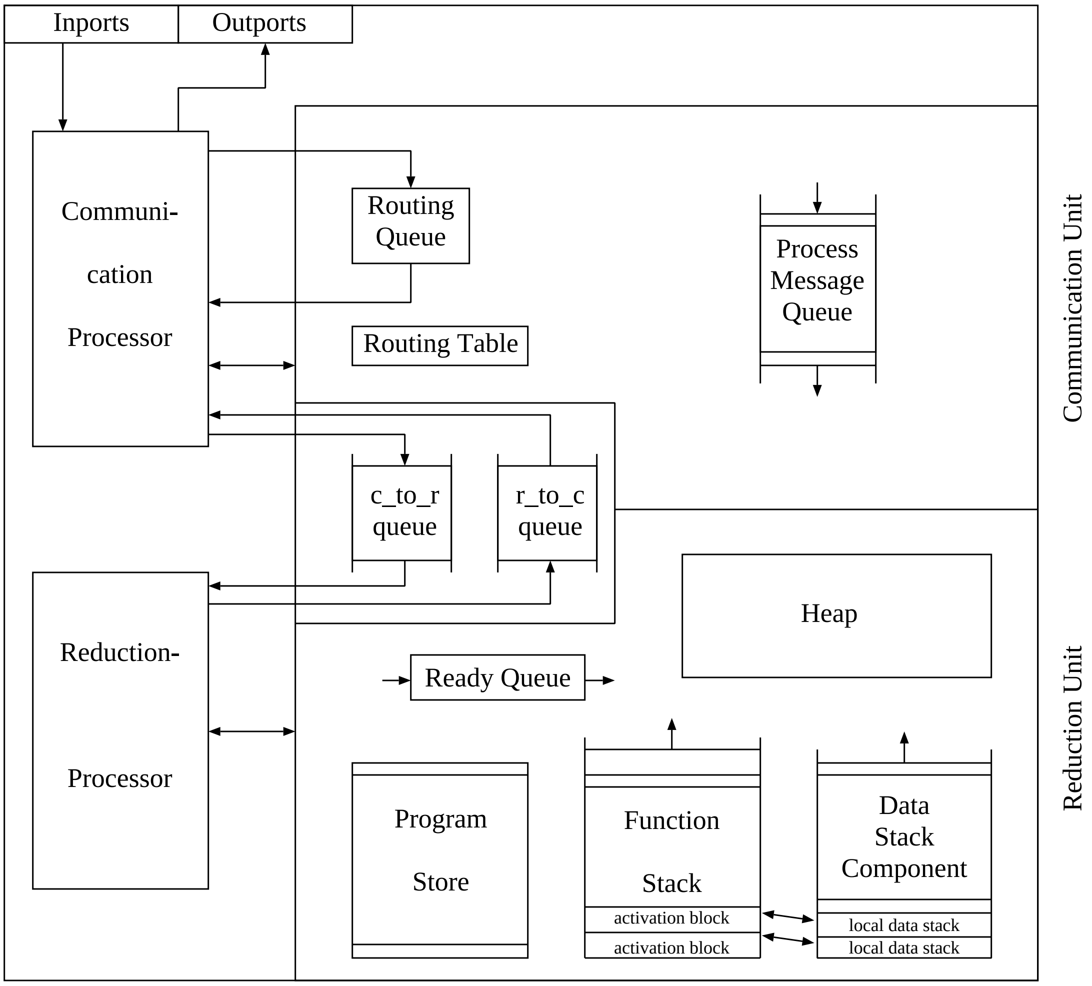

Figure 7: Global Structure of a Processor Unit

Process creation

For spawning a parallel function call a new instruction, similar to the CALL-instruction, is introduced:

TASK (ca, n, k, $size$) produces a process that can be executed in parallel. ca is the code address, n the number of arguments, k the number of local variables in the definition of the corresponding function and $size$ the maximal data stack size that will be necessary during the execution of the function.

In the heap component, an indirection node for the result is created.

A process message containing the code address and parameters of the parallel function call and the address of the indirection node is sent to the communication unit.

The computation of the actual task proceeds, until there is a demand for the result of a parallel subtask.

The communication unit stores the process message in its *process queue*, as processes are only distributed to other processor elements on demand. If the reduction unit runs idle, the communication unit can satisfy its workrequest by an entry from its own process queue before asking other processor elements. In this case, a parallel process is executed on its creation processor.
Let us consider now program compilation:
When reaching a **letpar**-construct, parallel tasks for all function calls on the right hand side are created by the TASK-instruction. The addresses of the placeholder nodes are stored into the corresponding local variable positions within the function block.

$$
\begin{array}{llll}
\mathit{exptrans}(& \mathbf{letpar} & y_{m+1} = \tilde{f}_1(e_{11},\ldots,e_{1l_1}) & \\
 & \mathbf{and} & \ldots & \\
 & \mathbf{and} & y_{m+p} = \tilde{f}_p(e_{p1},\ldots,e_{pl_p}) \ \mathbf{in}\ e) := & \mathit{pcode}(\tilde{f}_1(e_{11},\ldots,e_{1l_1})) \\
 & & & \text{STORE } m+1; \\
 & & & \quad\vdots \\
 & & & \mathit{pcode}(\tilde{f}_p(e_{p1},\ldots,e_{pl_p})) \\
 & & & \text{STORE } m+p; \\
 & & & \mathit{exptrans}(e)
\end{array}
$$

$$
\text{where } \mathit{pcode}(\tilde{f}_j(e_{j1},\ldots,e_{jl_j})) := \begin{cases} \mathit{argtrans}(e_{j1}) \\ \quad\vdots \\ \mathit{argtrans}(e_{jl_j}) \\ \text{TASK } (\mathit{ca}(\tilde{f}_j), l_j, \mathit{loc}(\tilde{f}_j), \mathit{size}(\tilde{f}_j)); \end{cases}
$$

Process suspension

The suspension of processes has to enable a context switch to another process.
When a process encounters a situation where a value is needed but not available yet, i.e. when a process accesses an indirection node, it has to wait until the indirection node is overwritten by the result. In the meantime the reduction unit should proceed with another process, which can be taken from the ready queue of reactivated processes or from the process queue in the communication unit. The following instruction performs the suspension of a process.

WAIT checks if the top entry of the data stack is evaluated. If this is not the case, the active process is suspended until the result is available. Therefore, the stack address of the current activation block is added to a *list of waiting tasks* within the placeholder node of the argument. When the result is delivered, the waiting list of an indirection node is moved to the ready queue.

The WAIT-instruction is used for the compilation of variables in a context where the result of the variable is needed.

$$
\mathit{exptrans}(y) := \begin{cases} \text{LOAD } i; & \\ \text{WAIT}; & \text{if } y \text{ is a variable introduced using } \mathbf{letpar} \\ & \\ \text{LOAD } i; & \\ \text{INITIATE}; & \text{else} \end{cases}
$$

We have to distinguish between variables introduced by the **letpar**-construct and others (formal parameters and local variables introduced by **let**). The latter are always locally available and can be accessed directly. The result of parallel processes may not be available, so one has to wait and suspend the actual evaluation.

Process termination

As in the sequential machine, the RETURN instruction terminates a function call. It returns the result to the calling process and marks the actual activation block and data stack as garbage. Due to parallel execution, the calling task can be local (so one can put the pointer onto the calling data stack and continue computing the calling task) or global (here the result is sent to the placeholder node on the other processor element).

6.3 Message Handling in the Reduction Units

During the execution of processes, the reduction units will produce

- *process messages* when parallel processes are spawned,
- *memory request messages* when graph nodes which are allocated in the heap of another reduction unit are needed,
- *answer messages* in order to reply to process and memory request messages.
- Finally, *work requests* are sent to the communication unit to ask for new processes.

In detail, the reduction units behave as follows on incoming messages:

- A *process message* represents a parallel ready-to-run process. It will be received after a work request message has been sent to the communication processor.

  A new activation block is created and its stack pointer is added to the ready queue. One cannot start the process directly (although demanded by the work request) because in the time between the work request and the task message, a result could have been delivered (which triggered the re-activation of some waiting tasks) or a memory request arrived, which could have triggered the activation of expressions (suspensions) not evaluated yet. Also, due to different scheduling strategies, several processes could be delivered on demand.
- A *memory request message* forces the reduction unit to look up a value for another processor element and to send the value in form of an answer message back to the inquiring processor. If the requested graph node is a suspension, the reduction unit will first initiate its local evaluation. The request message will be answered as soon as this evaluation finishes.
- Finally, *answer messages* overwrite the placeholder nodes and activate the waiting tasks. If the result needs more space than the placeholder offers, a local indirection node will be used.

6.4 Garbage-collecting the stack

In general, activation blocks can only be released immediately from the top-of-stack pointer until one finds a non-garbage block. Inbetween the stack, there may be still some "holes" left (see Figure 8). But a major advantage of our machine is, that this special structure can be subject to a special garbage collection technique.
Each block contains an individual block id, all references to the function block are made through this id. The actual address of the block can be found in a table which maps id's to stack addresses. If the garbage collector is invoked, the stack is simply crunched *in place*, deleting the holes. Only

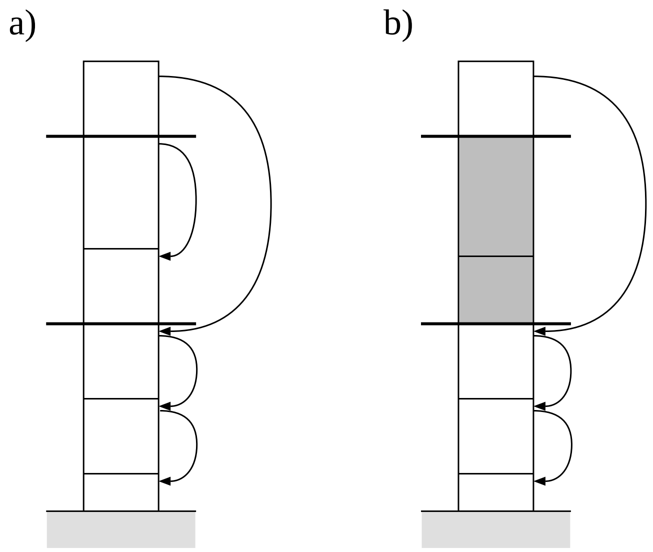

Figure 8: Function Stack Structure

the address entry in the table has to be redirected. Also, the entries for garbage id's can be deleted.
References to the stack occur *only* locally in the *waiting task nodes*, attached to the placeholder nodes. So, at the moment a waiting task is activated, it looks up its actual stack position in the table. So the garbage collection process is absolutely transparent to the rest of the environment. Due to the distinction between *control information* (in the stack component) and *data information* (stored in the heap component), one can handle each component with an appropriate garbage collection mechanism. E.g., the problem of data sharing needs only to be handled by the garbage collector of the heap component and causes no unnecessary administration overhead in the stack garbage collector. Details of the stack garbage collector will be given in [Schmidt 93].

7 First Implementation Results

The implementation of an interpreter version of the machine on a transputer system under Helios C is in progress. We will present some runtime results for the example programs given in the previous sections. But first we want to draw the attention to one efficient implementational optimization.
In a real system the length of the task queue, which is managed by the communication unit is limited. First of all the question arises, what to do in that case. Obviously, too much tasks are generated. So we introduce a flag *tqfull* which is set by the communication unit and can be read by the reduction unit. If the reduction process encounters a TASK-instruction, it first checks the boolean flag. If the task queue is full, instead of sparking a new task, a CALL-instruction is simulated, causing the generation of a *local* activation block. The introduction of this flag has two interesting side-effects:

1. If only one processor is involved (the total amount of processors in the system is known by every processor), the *tqfull*-flag can simply be set to *true*, thus suppressing any task generation.

2. The maximum number of task queue entries can be changed during runtime according to statistics about the system behaviour. So the system can adjust itself dynamically and the generation of too much tasks flooding the system can be prevented.

In our system we have implemented the first feature, which results in a runtime behaviour of a parallel program on one processor that is very similar to the sequential version of the program. In Figure 9 runtime measurements (in seconds) of the parallel example programs of Section 4 on 1, 2 and 4 processor elements (PE) are presented.

Program	n	1 PE	2 PE	4 PE
psum	5,000	2.42	1.31	0.83
	10,000	4.85	2.56	1.45
	20,000	9.69	4.98	2.79
	50,000	26.25	13.35	7.18
towers	11	0.87	0.49	0.39
	12	1.72	0.93	0.57
	13	3.43	1.81	1.16

Figure 9: Runtimes for example programs in seconds on the parallel machine

To get a hint on the proportions, we have listed the runtimes of the wellknown *nfib*-program

$$\begin{array}{ll} \mathit{nfib}(x) := \textbf{if } x < 2 \textbf{ then } 1 \textbf{ else} & \textbf{letpar } y_1 = \mathit{nfib}(x-2) \\ & \textbf{in } \mathit{nfib}(x-1) + y_1 + 1 \end{array}$$

in our parallel machine and in the graph-based PAM ([Loogen et al. 89], [Langendoen, Vree 92]) in Figure 10. Note that these runtimes are not directly comparable. The PAM was implemented in OCCAM under the Multitool programming environment and already has a garbage collection mechanism incorporated.

n	1 PE	2 PE	4 PE
20	6.04	3.14	1.79
21	9.77	5.02	2.99
22	15.80	8.28	4.23
knfib/s:	3.6	7.1	13.5

Runtimes for the *nfib*-program in seconds on the parallel machine

n	1 PE	2 PE	4 PE
20	19.78	–	5.01
21	32.00	–	8.05
22	51.79	–	13.00
knfib/s:	1.1	–	4.4

Runtimes for the *nfib*-program in seconds on the PAM

Figure 10

8 Related Work

Most parallel implementations of functional languages rely on the graph-based model of reduction ($\langle \nu, G \rangle$-machine of [Augustsson, Johnsson 89], the HDG-machine of [Kingdon et al. 91], ALFALFA [Goldberg, Hudak 86], PAM [Loogen et al. 89]). There have been attempts to incorporate stack techniques ([Koopman et al. 90], [Peyton-Jones 91]), but only on the sequential level. A similar approach to our technique has been taken in [Hofman et al. 92], but here only activation blocks on top of the stack are allowed to be executed. So inner activation blocks may be ready to be reactivated, but are blocked by the top-of-stack task. Note that they consider a fork/join task model. In our environment, where sharing of subtasks is allowed, this approach could result in deadlock situations. Nevertheless, their approach yields a good cache behaviour, which should be also observed in our machine.
The meshed stack technique is related to the stack technique used in the WAM [Warren 83]. In the WAM the runtime information of alternative computation branches are allocated on a single stack, while the meshed stack keeps the information of several parallel processes.

9 Conclusions and Future Work

We presented a basic framework of the machine which gives some opportunities for future work. In Section 6.4 we have presented an idea for garbage collecting the stack. For garbage collection of the heap, the approach of [Lester 89b], a combined scheme of a (local) copying and a (global) reference counting garbage collector, seems appropriate and will be implemented in the future. Furthermore we want to compare this parallel implementation to the sequential machine, to a graph-based version and to an implementation on a shared-memory multi-processor system.

References

[Appel 87] A.W. Appel: *Garbage Collection can be faster than stack allocation*, Information Processing Letters, 21(4):275–279, 1987.

[Augustsson, Johnsson 89] L. Augustsson, Th. Johnsson: *Parallel Graph Reduction with the* $\langle \nu, G \rangle$*-Machine*, Conference on Functional Programming Languages and Computer Architecture (FPCA), Addison-Wesley 1989.

[Goldberg, Hudak 86] B. Goldberg, P. Hudak: *ALFALFA — Distributed Graph Reduction on a Hypercube Multiprocessor*, Workshop on Graph Reduction, Lecture Notes in Computer Science 279, Springer Verlag 1986.

[Hogen et al. 92] G. Hogen, A. Kindler, R. Loogen: *Automatic Parallelization of Lazy Functional Programs*, 4th European Symposium on Programming (ESOP '92), LNCS 582, Springer Verlag 1992.

[Hofman et al. 92] R. Hofmann, K. Langendoen, W.G. Vree: *Scheduling Consequences of Keeping Parents at Home*, to appear in: Proceedings of ICPADS'92, December 1992.

[Hudak 89] P. Hudak: *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys, Vol. 21, No. 2, September 1989, 359–411.

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

[Kingdon et al. 91] H. Kingdon, D. Lester, G. Burn: *The HDG-Machine: a highly distributed graph-reducer for a transputer network*, The Computer Journal, Vol. 34, No. 4, 1991.

[Koopman et al. 90] P. Koopman, M. van Eekelen, E. Nöcker, J. Smeetsers, R. Plasmeijer: *The ABC-machine: A Sequential Stack-based Machine for Graph Rewriting*, Internal Report, University of Nijmegen 1990.

[Langendoen, Vree 92] K. Langendoen, W.G. Vree: *Parallel Implementations of Lazy Functional Languages*, to appear in Koen Langendoen's Ph.D. thesis, April 1993.

[Lester 89a] D. Lester: *Stacklessness: Compiling Recursion for a Distributed Architecture*, Conference on Functional Programming Languages and Computer Architecture (FPCA), Addison-Wesley 1989.

[Lester 89b] D. Lester: *An Efficient Distributed Garbage Collection Algorithm*, Conference on Parallel Architectures and Languages Europe (PARLE) 1989, Lecture Notes in Computer Science 365, Springer Verlag 1989.

[Loogen et al. 89] R. Loogen, H. Kuchen, K. Indermark, W. Damm: *Distributed Implementation of Programmed Graph Reduction*, Conference on Parallel Architectures and Languages Europe (PARLE) 1989, Lecture Notes in Computer Science 365, Springer Verlag 1989.

[Peyton-Jones 87] S.L. Peyton-Jones: *The Implementation of Functional Programming Languages*, Prentice Hall 1987.

[Peyton-Jones 91] S.L. Peyton-Jones: *The Spineless tagless G-machine: Second Attempt*, Proceedings of the Workshop on Parallel Implementation of Functional Languages, Technical Report CSTR 91–07, University of Southhampton 1991.

[Schmidt 93] A. Schmidt: *Garbage Collection of Meshed Stacks*, Diploma Thesis, RWTH Aachen 1993 (in preparation).

[Warren 83] D.H.D. Warren: *An abstract Prolog instruction set*, Technical Report 309, SRI International 1983.