appeared in: Proc. ESOP'92, Springer LNCS 582, pp. 254–268, 1992

Automatic Parallelization of Lazy Functional Programs

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

Abstract

We present a parallelizing compiler for lazy functional programs that uses strictness analysis to detect the implicit parallelism within programs. It generates an intermediate functional program, where a special syntactic construct '*letpar*', which is semantically equivalent to the well-known *let*-construct, is used to indicate subexpressions for which a parallel execution is allowed. Only for sufficiently complex expressions a parallelization will be worthwhile. For small expressions the communication overhead may outweigh the benefits of the parallel execution. Therefore, the parallelizing compiler uses some heuristics to estimate the complexity of expressions.

The distributed implementation of parallelized functional programs described in [Loogen et al. 89] enabled us to investigate the impact of various parallelization strategies on the runtimes and speedups. The strategy, which only allows the parallel execution of non-predefined function calls in strict positions, shows the best runtimes and reasonable speedup results.

1 Introduction

Due to their side effect free nature, functional programs contain implicit parallelism, which consists of independent subexpressions that can be evaluated in parallel. In lazy functional languages, one has to take into account that only subexpressions, whose result is necessary for the overall computation, may be evaluated. So, one has to use some analysis to determine the demanded subexpressions of a program. In general, strictness analysis is proposed for that purpose, because demanded subexpressions are always strict arguments of their context.

During the last decade, a lot of foundational work has been done on strictness analysis methods, see e.g. [Mycroft 81], [Maurer 85], [Burn et al. 86], [Burn 87a,b], [Hudak, Young 86], [Wadler 87], [Wadler, Hughes 87]. The implementation of strictness analysers has been considered in [Clack, Peyton-Jones 85] — with a generalization in [Martin, Hankin 87] — and in [Young, Hudak 86] and [Nöcker 90].

The aim of our work has been to implement a strictness analyser and use it as the kernel of a parallelizing compiler for lazy functional programs. The compiler generates an intermediate functional program, where a special syntactic construct '*letpar*' is used to indicate subexpressions for which a parallel execution is allowed. Although the theoretical foundations of the parallelization process are well-understood, there is a lack of practical experience. Especially the question of how to decompose a program with strictness annotations into sufficiently complex parallel processes has not yet been investigated.

Recently, it has been questioned whether an automatic parallelization of functional programs is at all possible [Vrancken 90]. Of course, one cannot expect that a parallelizing compiler automatically transforms a sequential algorithm into a parallel one. It will always remain the task of the programmer to design the parallel algorithm. The advantage of using a functional programming language instead of a 'parallel' conventional language like OCCAM or Parallel C is that the programmer needs not to think about distribution, communication and synchronization of parallel processes. He "only" has to *specify* the process system, the *management* of the dynamic processes is done by the runtime system. The automatic parallelization — as we understand it — simplifies the portability of programs. The *same* functional program can be executed on either a sequential or, after parallelization, on a parallel system.

*email: {ghogen,kindler,rita}@zeus.informatik.rwth-aachen.de

Most projects on the implementation of functional programming languages on parallel architectures directly use a functional language with explicit parallelism, given by special annotations or syntactic constructs, as source language, where it is either claimed that the programmer is responsible for the specification of parallelism and its correctness with respect to the semantics of programs, or that some parallelizing compiler should transform functional programs into their 'parallel' source language, see e.g. [Hudak 86], [Raber et al. 88], [van Eekelen et al. 88], [Hammond, Peyton Jones 90], [McBurney, Sleep 90]. These and several other projects concentrate on the efficient implementation of functional languages plus explicit parallelism on parallel distributed or shared memory architectures rather than on the detection and exposition of parallelism in conventional functional programs.

The work on strictness analysis has often been motivated by the detection of implicit parallelism in functional programs, but up to now only few approaches exist on the use of strictness analysers for that purpose. The first papers that describe a parallelizing compiler, whose structure is similar to ours, are [Hudak, Goldberg 85a,b]. They use some first order strictness analysis over flat domains to detect parallelizable subexpressions and then also use some, not further specified, heuristics to decide which subexpressions should really be executed in parallel.

Our approach can be seen as a continuation of Hudak's and Goldberg's work. We use an advanced strictness analyser that is based on the evaluation transformer approach of [Burn 87a,b], which takes into account context information to handle data structures in an appropriate way. Higher order functions are also supported. It is implemented by the frontier's algorithm, that was introduced in [Clack, Peyton-Jones 85] and extended to include higher-order functions and general lattices in [Martin, Hankin 87].

The main new aspects of our work lie in the discussion and comparison of strategies for the decomposition of analysed programs into parallel processes. Experimental results show that these strategies may have a great impact on the runtimes of parallelized programs and the achievable speedup values.

In this paper we concentrate on three natural decomposition strategies which are compared with respect to runtime experiments using the distributed implementation of [Loogen et al. 89]. It turns out that the strategy, which only allows the parallel execution of non-predefined function applications, gives the best results for our example programs.

The treatment of functions with functional parameters reveals an important property of our parallel implementation, namely the facility to perform a *dynamic parallelization*, which exploits parallelism that cannot be detected by our parallelizing compiler. As it is only possible to parallelize applications where the function is definitely strict in some arguments and as functional parameters may be instantiated by non strict functions, the parallelization of applications of functional parameters is not at all possible. For this reason, a *static parallelization* of higher order functions is not feasible, in general. This is the situation where the dynamic parallelization applies. It allows the parallel evaluation of "complex" arguments of dynamically created applications, i.e. applications of functional parameters that are instantiated by some strict function at runtime, or applications, whose evaluation has been delayed. Consequently, it might be the case that a functional program is evaluated in parallel, although it does not contain any *letpar*-construct, i.e. explicit (static) parallelism.

[Burn 88] and [Lester, Burn 89] also present a realization of the evaluation transformer model of parallel evaluation of functional programs. The main difference to our approach lies in the fact that they allow parallel evaluation wherever possible. This may lead to the generation of too small processes, for which the overhead of the parallel execution is higher than the benefits.

The paper is organized as follows. Section 2 describes the parallelizing compiler. In Section 3 we review our implementation of parallelized functional programs on a distributed memory system (transputers). A comparison of different parallelization strategies with runtime results is contained in Section 4. The dynamic parallelization of higher order functions is discussed in Section 5. Section 6 finally contains some conclusions and hints at future work.

2 Structure of the Parallelizing Compiler

The parallelizing compiler translates a lazy functional program into a system of *parallelized combinators*, i.e. a set of global function definitions, in which the places, where a parallel evaluation of subexpressions may take place, are indicated by a special syntactic construct. This transformation needs three phases as

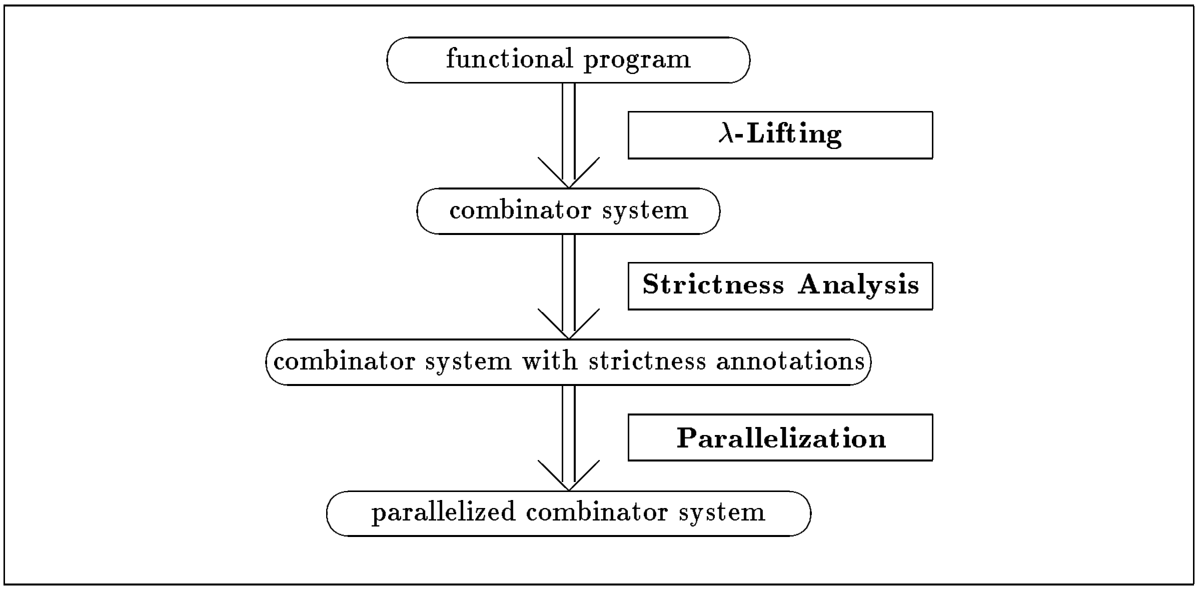

Figure 1: Structure of the Parallelizing Compiler

$$
\begin{array}{l}
\mathit{quicksort}\ l^{\mathrm{intlist}} = \textbf{case}\ l\ \textbf{of}\ \ \textsc{nil} : \textsc{nil};\\
\qquad\qquad\qquad\qquad \textsc{cons}\ h^{\mathrm{int}}\ t^{\mathrm{intlist}} : (\mathit{append}\ (\mathit{quicksort}\ (\mathit{filter}\ \mathit{less}\ t)),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textsc{cons}\ h\ (\mathit{quicksort}\ (\mathit{filter}\ \mathit{geq}\ t))))\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ \mathit{less}\ x^{\mathrm{int}} = (<\ x\ h)\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{and}\quad \mathit{geq}\ x^{\mathrm{int}} = (\geq\ x\ h)\\
\mathit{filter}\ \mathit{test}^{\mathrm{int}\rightarrow\mathrm{bool}}\ l^{\mathrm{intlist}} = \textbf{case}\ l\ \textbf{of}\ \ \textsc{nil} : \textsc{nil};\\
\qquad\qquad\qquad\qquad\qquad \textsc{cons}\ h^{\mathrm{int}}\ t^{\mathrm{intlist}} : \textbf{if}\ (\mathit{test}\ h)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (\textsc{cons}\ h\ (\mathit{filter}\ \mathit{test}\ t))\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (\mathit{filter}\ \mathit{test}\ t)\\
\mathit{append}\ l_1^{\mathrm{intlist}}\ l_2^{\mathrm{intlist}} = \textbf{case}\ l_1\ \textbf{of}\ \ \textsc{nil} : l_2;\\
\qquad\qquad\qquad\qquad \textsc{cons}\ h^{\mathrm{int}}\ t^{\mathrm{intlist}} : (\textsc{cons}\ h\ (\mathit{append}\ t\ l_2))
\end{array}
$$

Figure 2: Example Program

indicated in Figure 1. We explain the main ideas of these phases using the quicksort program given in Figure 2.

2.1 λ-Lifting

The translation of functional programs with nested function definitions into a system of global function definitions (combinators) is called λ-lifting and was introduced in [Johnsson 85]. λ-lifting has several advantages, especially with respect to the parallel execution of functional programs. For the evaluation of functional programs with nested function definitions, one has to use a central environment structure which contains the bindings of global variables during the evaluation of local function applications. Such a central structure is of course an obstacle for a parallel implementation, especially when the target architecture is a distributed memory system. λ-lifting lifts local function definitions to the upper level by extending the parameter list of such functions by the global variables that occur in their body. By explicitly passing the values of global variables to the function, a central environment structure becomes superfluous. The value of a combinator application only depends on the arguments. Thus, it causes no problems to pass combinator applications to other processors for evaluation.

λ-lifting of the example program in Figure 2 simply lifts the local non-recursive functions *less* and *geq*, by extending their parameter list by the global variable h. Each occurrence of the local functions is replaced by an application of the new global function to the global variable. Thus the function *quicksort* is replaced by

$$
\begin{array}{|l|}
\hline
\textit{quicksort}\ l^{\text{intlist}} = \textbf{case}\ l\ \textbf{of}\ \ \text{NIL} : \text{NIL}; \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{CONS}\ h^{\text{int}}\ t^{\text{intlist}} : (\textit{append}\ (\textit{quicksort}\ (\textit{filter}\ (\textit{global\_less}\ h)\ t)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{CONS}\ h\ (\textit{quicksort}\ (\textit{filter}\ (\textit{global\_geq}\ h)\ t)))) \\
\textit{global\_less}\ h^{\text{int}}\ x^{\text{int}} = (<\ x\ h) \\
\textit{global\_geq}\ h^{\text{int}}\ x^{\text{int}} = (\geq\ x\ h) \\
\hline
\end{array}
$$

Figure 3: λ-Lifted Example Program

the system given in Figure 3. The functions *filter* and *append* are omitted, because they are not changed.

In general, λ-lifting is more involved, especially for mutually recursive local function definitions. A complete algorithm is described in [Johnsson 85].

2.2 Strictness Analysis

Strictness analysis is used to detect the maximal source of parallelism within a program. Here, we use the technique of abstract interpretation to determine the strictness properties of combinator systems [Abramsky, Hankin 87]. The analysis applies only to monomorphic programs, but in [Abramsky 85] it has been shown that the result of the analysis of any monomorphic instance of a polymorphic function is valid for all instances.

Monomorphic programs are interpreted over the abstract base domains $\{0,1\}$ and $\{0,1,2,3\}$ following the approach of [Mycroft 81, Burn et al. 86, Wadler 87]. The abstract domains have the following intuitive meaning:

- $\{0,1\}$ with $0 < 1$ is the abstraction of non-structured base domains.

  0 represents the bottom value $\perp$.
  1 represents all values.

- $\{0,1,2,3\}$ with $0 < 1 < 2 < 3$ is the abstraction of the domains of algebraic data structures.

  0 represents the bottom value $\perp$, i.e. the non-defined structure.
  1 represents in addition, partial and infinite structures.
  2 represents in addition, finite structures (with possibly undefined entries).
  3 represents all structures, i.e. in addition, finite structures with defined entries.

The abstract interpretation is defined in such a way that:

> The standard interpretation of a function definition is strict in an argument i, if its abstract interpretation is strict in this argument.

Thus, strictness in the abstract interpretation is a decidable approximation of strictness in the standard interpretation.

Example: The abstract interpretation of the combinators of our small example program is as follows. $F^\sharp$ denotes the abstract interpretation of a combinator F.

$quicksort^\sharp$	
$l^\sharp$	
0	$\boxed{0}$
1	0
2	2
3	3

$filter^\sharp$				
$test^\sharp \backslash l^\sharp$	0	1	2	3
$b^\sharp \to 0$	0	0	0	$\boxed{3}$
$b^\sharp \to b^\sharp$	0	1	1	3
$b^\sharp \to 1$	$\boxed{0}$	1	3	3

$append^\sharp$				
$l_1^\sharp \backslash l_2^\sharp$	0	1	2	3
0	0	0	0	$\boxed{0}$
1	1	1	1	1
2	1	1	2	2
3	$\boxed{1}$	1	2	3

$global^{less^\sharp}_{geq^\sharp}$		
$h^\sharp \backslash x^\sharp$	0	1
0	0	$\boxed{0}$
1	$\boxed{0}$	1

The boxed entries show the strictness properties of the combinators. We can determine the strictness property by applying $F^\sharp$ to $\perp$ in the tested argument position and to the top values of the abstract domain in all other positions. E.g., *append* is strict in its first argument, because $append^\sharp(0,3) = 0$, but not in its second argument, since $append^\sharp(3,0) = 1 > 0$. The combinators *global_less/geq* are strict in both arguments.

To compute the abstract interpretation of combinator systems, we apply the *frontier method* of [Clack, Peyton-Jones 85], which has been extended in [Martin, Hankin 87] to handle higher order functions and data structures.

In order to handle data structures in an appropriate way, we use the *evaluation transformer model of computation*, which has been introduced in [Burn 87a]. In this model, expressions are annotated by *evaluators*, which indicate the amount of evaluation that can be done on these expressions. The following evaluators are distinguished:

- ξ_0 — means no evaluation,
- ξ_1 — evaluates expressions to weak head normal form,
- ξ_2 — evaluates the structure of data types, i.e. all constructor nodes, but no entries. For example, the structure of a list is its spine.
- 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. Evaluation to *weak head normal form* means that data structures are evaluated up to their top level constructor and functional values are evaluated, until a partial application[1] of a base function, constructor or combinator is reached. Expressions of a base type are completely evaluated. The additional evaluators allow a more accurate treatment of data structures, because they enable an earlier and more extensive evaluation of components and thus may reveal more implicit parallelism.

The evaluators are ordered by $\xi_0 < \xi_1 < \xi_2 < \xi_3$. There is a close correspondence of evaluators and abstract domain values:

> The evaluation of an expression with evaluator ξ_i terminates if and only if the abstract value of the expression is greater than or equal to i.

An *evaluation transformer* is a function that maps evaluators of a function application to suitable evaluators for the argument expressions. So it can be used to determine for each application, which amount of evaluation can be done on the arguments, when the evaluator for the whole application is known.

Thus, the *evaluation transformer* of an n-ary function F consists of n functions, $ET_i(F)$, $1 \leq i \leq n$, which map the evaluator of an application of F to the evaluator for the ith argument. It can simply be determined using the abstract interpretation of F (for details see [Burn 87a]).

Example: The evaluation transformers of the combinators in our example program are as follows:

$ET(quicksort)$	
ξ	$\mathrm{ET}_1(\xi)$
ξ_0	ξ_0
ξ_1	ξ_2
ξ_2	ξ_2
ξ_3	ξ_3

$ET(filter)$		
ξ	$\mathrm{ET}_1(\xi)$	$\mathrm{ET}_2(\xi)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_0	ξ_1
ξ_2	ξ_0	ξ_2
ξ_3	ξ_0	ξ_2

$ET(append)$		
ξ	$\mathrm{ET}_1(\xi)$	$\mathrm{ET}_2(\xi)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_1	ξ_0
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_3

$ET(global\ldots)$		
ξ	$\mathrm{ET}_1(\xi)$	$\mathrm{ET}_2(\xi)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_1	ξ_1

If an application of *quicksort* is evaluated with evaluator ξ_1, it is e.g. safe to evaluate the argument list with ξ_2. Intuitively this is true, because it is only possible to determine the top level constructor of the result list, if it is empty, contains only one element or if the minimum element has been determined. As sorting of the list (CONS $\perp$ NIL) yields the list itself, without that the non defined entry is accessed, the maximal possible evaluator for the argument list is ξ_2.

Evaluation transformers can also be introduced for arbitrary expressions. One simply views the expression as a function of its direct subexpressions. In this case, one even gets more specific information, because it is possible to use the abstract interpretation of the other subexpressions instead of assuming the top value of the abstract domains for these, when determining the strictness properties of argument subexpressions. In general, this approach is called *context sensitive* strictness analysis. We therefore call the evaluation transformer of an arbitrary expression e *context sensitive* and denote it by $ET^{cs}(e)$.

[1] Partial applications correspond to λ-abstractions.

2.3 Parallelized Combinator Systems

In the parallelized combinator systems, explicit parallelism is indicated by the following *letpar*-construct:

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

where the subexpressions of an expression e, for which it is decided that a parallel execution may be worthwhile, are abstracted out of the expression and represented by combinator applications. Thereby, new combinators are introduced, if a subexpression is not yet a combinator application. Thus, in the *letpar*-construct,

- y_i are local variables, which replace the abstracted subexpressions in e,
- F_i denote (possibly new) combinators with n_i parameters in the defining equations,
- e and e_{ij} are applicative expressions built from variables, predefined constants and functions, constructors and combinators by application, *if_then_else*, *case*, *let* and *letpar*-constructs,
- ev_i are evaluators.

The meaning of a *letpar*-expression is, that the combinator applications $F_i(e_{i1}, \ldots, e_{in_i})$ *may be* evaluated in parallel with the main stream of evaluation, e, if the evaluator of the whole expression is stronger, i.e. greater, than or equal to ev_i. 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 combinator applications, for which the evaluator allows a parallel execution. Then the evaluation of the body e proceeds until the result of a parallel subexpression, represented by y_j, is needed and not yet available. In this case, the evaluation of e is suspended until the parallel subprocess yields its result. In the meantime, the processor is free to evaluate some other process.

The resulting process system is hierarchical. The execution of a parallelized combinator system starts with the evaluation of the 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.

2.4 Parallelization

Principally, each expression of a program, for which an evaluator different from ξ_0 can be determined, can be evaluated in parallel. Such a maximal parallelization may however not be optimal, because a parallel evaluation always causes an overhead that might be greater than its gain. The expression, that is to be evaluated in parallel, must be encoded in a message that is passed to another processor element and there must be decoded. In the same way, the result[2] of the parallel evaluation must be transferred back to the original processor element. In order to guarantee that the parallel execution of an expression is profitable, one has to ensure that the expression has enough evaluation complexity (whatever this means). In general, it will not be possible to determine the complexity of expressions exactly, even if one focusses on a special implementation technique and target architecture. So, one has to use some heuristics.

In order to translate combinator systems into parallelized combinator systems, we use, in addition to the evaluator information given by the evaluation transformers of the combinators, constructors and base functions, an oracle function

$$oracle : Expressions \rightarrow \{ \text{true}, \text{false} \}$$

that determines whether a parallelization is allowed or not. The oracle functions, that we used in our experiments, are presented in Section 4. In the remainder of this section, we describe the parallelization algorithm and finish with a possible parallelization of our example program.

The parallelization algorithm transforms expressions into expressions, hopefully containing the *letpar*-construct. On the top level it is applied to the bodies of the combinator definitions. The most important

[2]in the case of data structures at least the top-level constructor

case is the treatment of general applications $(e_0\ e_1 \dots e_m)$. First, one has to determine the sets of parameters, for which a parallelization is allowed,

- $O := \{i \mid 0 \le i \le m, \mathit{oracle}(e_i) = \text{true}\}$,

and the set of parameters, for which an evaluator stronger than ξ_0 can be expected,

- $P := \{i \mid 0 \le i \le m, \exists \xi :\ ET_i^{cs}((e_0\ e_1 \dots e_m))(\xi) > \xi_0\}$.

The subexpressions, whose indices are in $O \cap P$, can be evaluated in parallel. For the subexpressions in $O \setminus P$, the oracle allows a parallel execution, but the non-strict context forbids an evaluation. As it is possible that such expressions are dynamically passed to a strict context, we providently replace these expressions, if necessary, by a combinator application in order to enable a parallel evaluation. Our implementation does not only support the explicit parallelism introduced by the *letpar*-construct, but also the parallel execution of strict, not yet evaluated, arguments, which themselves are combinator applications, as only combinator applications can be distributed in the system. This is an additional source of parallelism, which may lead to the generation of parallel processes where a static parallelization during compile time is not possible. Consequently, our approach ensures that a delayed activation of the subexpressions, whose indices are in $O \setminus P$, may nevertheless lead to a parallel evaluation. The effect of this *dynamic parallelization* in the case of higher order functions is shown in Section 5.

In order to avoid unnecessary process switches, the parallelization should ensure that enough work is kept on the processor element that evaluates the *letpar*-expression and does the parallel activation of the subprocesses. Thus, two cases are distinguished in the parallelization algorithm. If the oracle says that the whole expression can be evaluated in parallel, all parallelizable subexpressions are abstracted out. Otherwise, the leftmost parallelizable subexpression is kept for execution on the same processor element.

In order to keep as much locality as possible and to avoid flooding the system with parallel processes whose results are not yet needed, we abstract expressions only one level out. Thus, we parallelize e.g. an expression $(F \dots (G \dots (H \dots) \dots) \dots)$,
where we assume that the shown subexpressions are in strict parameter positions and allowed to be parallelized by the oracle, in the following way

$$\begin{array}{l} \textbf{letpar}\ y = (F_{p1} \dots)\ \textbf{if} \dots \\ \textbf{in}\ (F \dots y \dots) \end{array}$$

where the subsequent combinator is introduced:

$$F_{p1} \dots = \begin{array}{l} \textbf{letpar}\ y = (H \dots)\ \textbf{if} \dots \\ \textbf{in}\ (G \dots y \dots) \end{array}$$

Another possibility would be to generate the expression

$$\begin{array}{l} \textbf{letpar}\ y_1 = (H \dots)\ \textbf{if} \dots \\ \textbf{in}\ \textbf{letpar}\ y_2 = (G \dots y_1 \dots)\ \textbf{if} \dots \quad , \\ \quad \textbf{in}\ (F \dots y_2 \dots) \end{array}$$

which may however lead to unfavourable situations, because the result of the subprocess H of G will not be directly communicated to the place where G is executed, but via the processor element, where H has been activated. Thus, locality is lost. Furthermore, it may be disadvantageous to start subprocesses before the processes that consume their result, especially when the subprocesses need a lot of resources and produce a lot of output. The advantage of the second approach is a broader and faster parallelization with an early activation of subprocesses. Note that there is always a trade-off between parallelism and locality.

Let us come back to the problem of parallelizing an application $(e_0\ e_1 \dots e_m)$. Let $O \cap P = \{i_1, \dots, i_p\}$ and $\mathit{vars}(e_j) = \{v_{j1}, \dots, v_{jh_j}\}$ be the set of all (free) variables in e_j $(j \in O)$.

- If $\mathit{oracle}((e_0\ e_1 \dots e_m)) = \text{true}$, the parallelization algorithm produces the expression

$$\begin{array}{llcll} \textbf{letpar} & y_1 & = & (F_{i_1}^{\text{new}} v_{i_1 1} \dots v_{i_1 h_{i_1}}) & \textbf{if}\ \ ev_{i_1} \\ \textbf{and} & & \dots & & \\ \textbf{and} & y_p & = & (F_{i_p}^{\text{new}} v_{i_p 1} \dots v_{i_p h_{i_p}}) & \textbf{if}\ \ ev_{i_p} \\ \multicolumn{5}{l}{\textbf{in}\ (\mathit{exp}_0\ \mathit{exp}_1 \dots \mathit{exp}_m),} \end{array}$$

where

– new combinators F_j^{new} $(j \in O)$ with defining equations "$F_j^{\text{new}} v_{j1} \ldots v_{jh_j} = \mathit{parallelize}(e_j)$" are introduced,

– $ev_{i_j} = \min\{\xi \mid ET_{i_j}^{\text{cs}}((e_0\ e_1 \ldots e_m))(\xi) > \xi_0\}$ and

– $\mathit{exp}_k = \begin{cases} y_j & \mathit{if}\ k = i_j \in O \cap P \\ (F_k^{\text{new}} v_{k1} \ldots v_{kh_k}) & \mathit{if}\ k \in (O \setminus P) \\ \mathit{parallelize}(e_k) & \mathit{otherwise} \end{cases} \qquad (0 \leq k \leq m).$

- If $\mathit{oracle}((e_0\ e_1 \ldots e_m)) =$ false, a similar parallelization takes place, except that the expression, whose index is the minimal element of $O \cap P$, is not abstracted out for parallel activation, but kept in the body of the *letpar*-expression for local execution.

The parallelization of *if-then-else* and *case* expressions consists of an independent parallelization of their subexpressions. The treatment of *let*-expressions can be reduced to the parallelization of applications, because a *let*-expression can be viewed as a special application.

The parallelization of our *quicksort* example, assuming an oracle that only returns true for (complete) combinator applications, leads to the parallelized combinators given in Figure 4. The other combinators of the example remain unchanged.

$$\begin{array}{lll}
\mathit{quicksort}\ l^{\text{intlist}} & = & \textbf{case}\ l\ \textbf{of}\ \ \text{NIL} : \text{NIL}; \\
& & \qquad \text{CONS}\ h^{\text{int}}\ t^{\text{intlist}} : \\
& & \qquad\quad \textbf{letpar}\ y_1 = (F^{\text{new}}\ h\ t)\ \textbf{if}\ \xi_1 \\
& & \qquad\quad \textbf{in}\ (\mathit{append}\ y_1\ (\text{CONS}\ h\ \textbf{letpar}\ y_2 = (\mathit{filter}\ (\mathit{global_geq}\ h)\ t)\ \textbf{if}\ \xi_1 \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (\mathit{quicksort}\ y_2))\ ) \\
F^{\text{new}}\ h^{\text{int}}\ t^{\text{intlist}} & = & \textbf{letpar}\ y = (\mathit{filter}\ (\mathit{global_less}\ h)\ t)\ \textbf{if}\ \xi_1 \\
& & \textbf{in}\ (\mathit{quicksort}\ y)
\end{array}$$

Figure 4: Parallelized Example Program

When the expression, that is abstracted out for parallel execution, is already a combinator application, we avoid the introduction of new combinators during the parallelization process. In our example in Figure 4, we have not introduced new combinators for the 'parallel' call of *filter*. In fact, our current implementation always avoids the generation of new combinators when the subexpression, for which a parallel execution is decided, is already a combinator application. This implies that the parallel execution of arguments of the combinator application is not enabled.

3 The Underlying Parallel Implementation

The parallel implementation of the combinator systems is based on a parallel abstract machine (PAM) [Loogen et al. 89], that consists of a finite number of identical processor elements with local storage. The processor elements communicate by exchanging messages via an interconnection network.

PAM has a modular structure. Each processor element consists of two independent processing units (see Figure 5):

- a *communication unit* that is responsible for the organizational aspects of the parallelization of the reduction process, and
- a *reduction unit* that executes the parallelized functional program by code-directed graph reduction.

The two processing units communicate by exchanging messages in a local shared memory.

The modularization permits a decentralization of the parallel program execution by separating the overhead of parallelism — message handling, work distribution, workload balancing — from the reduction process. This leads to a better exploitation of parallelism.

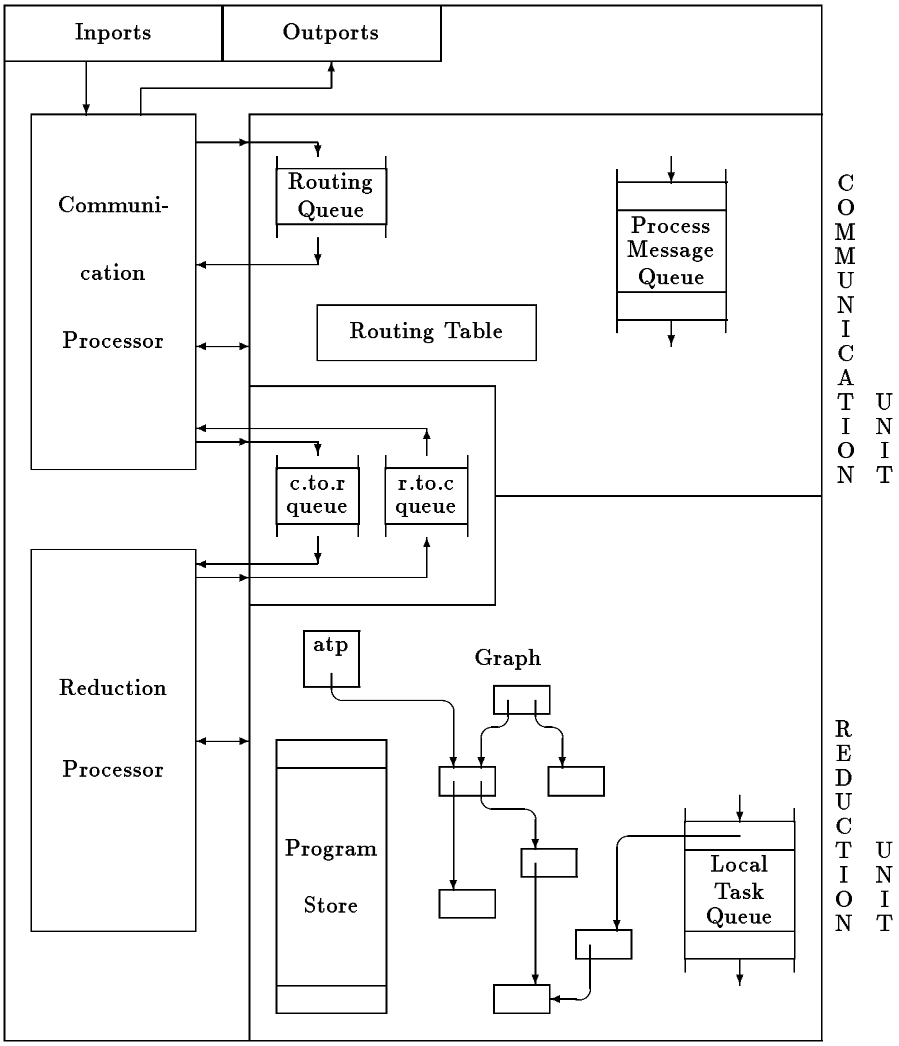

Figure 5: Structure of a Processing Unit

3.1 Program Execution

In the parallel abstract machine, program execution is done within the reduction units by *distributed code-directed graph reduction.*

The parallelized combinator program is translated into abstract machine code, that is copied into the program store of each reduction unit. For each potential ⟨combinator, evaluator⟩-pair, a machine code sequence is generated that controls the execution of combinator calls with this evaluator. The evaluation transformer information is integrated into this code sequence. So, the (first) evaluator of each subexpression can be determined at compile time. In particular, it is fixed which *letpar*-expressions yield parallel processes. At run time, one only has to choose the appropriate code sequence for a combinator and its evaluator.

Each code sequence for a combinator has two different entry points which correspond to the two different activation modes for applications. The first entry point leads to the potentially parallel activation of the arguments of the combinator using the evaluator given by its evaluation transformer, before the second entry point is passed which immediately leads to the evaluation of the combinator body. The second entry point can be used when it is known that the evaluation of the arguments of the combinator has already been initiated. This is the case for combinator applications within the program, which are *directly activated.* The evaluation of the arguments is initiated just before the combinator call, using the evaluator given by the context sensitive evaluation transformer of the application. The first entry point is used for *indirectly activated* applications, where we distinguish two cases. On the one hand, combinator applications can be dynamically created by some "higher order" applications. On the other hand, the evaluation of a combinator application may be delayed, because it appears in a non-strict context. When such applications are activated, one cannot assume that the arguments have already been evaluated.

TASK	function id	argument list	evaluator	ip	ds	fq	flags
DATA	value						
CONSTRUCTOR	constructor id	component list	evaluator				
FUNCTION	function id	argument list	number of missing arguments				
SUSPENSION	argument list	code address (triple)					
INDIRECTION	address / ?	evaluator	fq				

Figure 6: Graph Nodes

The main runtime structure is a *graph* which contains the following types of nodes (see also Figure 6):

- the activation blocks for the combinator calls, called *task nodes*, which are not, as usual, allocated on a stack, but, in order to support the parallelism, in the decentral graph structure,

  — Each task node contains the whole status information of the combinator call evaluation, i.e. a local instruction pointer (ip), a local data stack[3] (ds), a list of tasks waiting for the result (fq, father queue), the evaluator etc. This decentral organisation makes task switches very simple and cheap. —

- *terminal nodes*, where we distinguish data, constructor and function nodes, which represent expressions in weak head normal form,

- *suspension nodes*, which represent expressions whose evaluation is not yet demanded, due to the lazy evaluation strategy, and finally

- *indirection nodes*, which reserve place for the result of parallel processes, which are executed on other processor elements, or for terminal nodes, which have to be transferred from other processor elements.

At each time, a reduction unit executes at most one task, whose task node is indicated by the so called *active task pointer* (atp). A task is executed until it is finished or suspended. A suspension occurs, when a task has to wait for some information that is locally not yet available.

During a task execution, new local tasks or even parallel processes may be generated. The local tasks are kept in a *local task queue*, while the parallel processes are encoded into a process message and passed to the communication processor.

A task switch, triggered by a task termination or suspension, means that the reduction unit inspects the local task queue for a new task. If this queue is empty, the communication processor is asked to pass a new parallel process to the reduction unit. When a parallel process is passed to the reduction unit, it will be executed in this unit until it finishes.

3.2 Message Handling

Each reduction unit can be seen as a sequential reduction machine that has been extended for multitasking and message handling. Four types of messages are necessary for the parallelization of the reduction process:

- *process messages* to distribute the parallel processes among the processor elements,

- *answer messages* to communicate the result of distributed parallel processes and the contents of shared subgraphs,

- *request messages* to ask for graph nodes that are allocated on other processor elements and

- *activation messages* to activate the evaluation of subgraphs on other processor elements.

[3] The maximum storage requirements can be computed at compile time.

A parallel process always corresponds to a combinator application. Thus, it is completely specified by the combinatorname, the list of arguments, the evaluator with which the parallel process has to be evaluated, and finally the return-address (the address where the result of the task has to be sent to). These are exactly the contents of a process message. When a parallel process is initiated, an indirection node for the result of the process is generated and an appropriate process message is passed to the communication processor.

Access to an indirection or task node leads to the suspension of a task, i.e. its execution is stopped and its address is written into the accessed indirection or task node. For that purpose, indirection nodes and task nodes incorporate lists fq containing addresses of task nodes, which are waiting for the result of these nodes, i.e. for the indirection node to be overwritten by the result of a parallel process or the contents of a graph node, that lies on another processor element (both communicated by an answer message), and for the task to finish, respectively.

If a task needs access to a global node, a new indirection node is generated, a request message containing the address of this indirection node is sent to the global address and the address of the task is written into the new indirection node. When a task finishes or an indirection node is overwritten by the contents of an answer message, the tasks whose addresses are noted in the task or indirection node are reactivated.

Each communication processor maintains in its local store a process queue, that contains process messages that have been passed from other processor elements or from the reduction unit. When a process message from the reduction unit is passed to the communication processor, it will decide whether to keep it in the own process queue or to distribute it to a neighbour processor[4] depending on the load balancing strategy.

3.3 The Concrete Implementation

The abstract machine has been implemented on an OCCAM/Transputersystem [Loogen et al. 89, Hogen 91], where one processor element of the abstract machine runs on one transputer. Asynchronous message passing is simulated by buffering messages to ensure that no processor is ever blocked because it has to wait for a communication.

As the interconnection network does not provide a full interconnection of all transputers, messages must be routed through the network. The current implementation realizes a simple *static routing scheme*, where a routing table determines for each processor element, via which neighbour a message to that processor must be passed.

Experiments with our implementation showed that a very simple *passive load balancing strategy* is in general sufficient [Kuchen, Wagener 90]: Processors, which have no work and run idle, 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.*

4 Investigation of Parallelization Strategies

In Section 2, we described the parallelizing compiler, up to the function $oracle : Expressions \rightarrow \{\text{true, false}\}$ used in the final phase to decide whether a parallelization is allowed or not. As the resulting parallelized combinator program strongly depends on the choice of this oracle function, we investigated four different oracle functions, which realize different levels of parallelization, with our implementation.

1. $seq : e \mapsto \text{false}$ does not allow any parallelization.

2. $comb : e \mapsto \begin{cases} \text{true} & \textit{if } e \text{ is a complete combinator application} \\ \text{false} & \textit{otherwise} \end{cases}$

   only allows the parallel execution of combinator applications. This strategy, in general, realizes a very natural parallelization, that very often resembles the parallelization that one would intuitively produce by hand.

   Assuming that an atomic expression is a variable, base function, constructor or combinator, we define

3. $nested : e \mapsto \begin{cases} \text{true} & \textit{if } e \text{ is not an atomic expression and has not only atomic subexpressions} \\ \text{false} & \textit{otherwise} \end{cases}$

[4] A neighbour processor is a physically connected processor.

nested allows more parallelism than *comb*, but parallelizes still less than

4. $max : e \mapsto \left\{ \begin{array}{ll} \text{true} & \textit{if } e \text{ is not an atomic expression} \\ \text{false} & \textit{otherwise} \end{array} \right\}$ which produces the *maximal parallelization.*

Note that these oracle functions are completely independent from the underlying implementation. They consider only the structure of expressions.

The parallelizing compiler has been implemented in Turbo-Pascal[5] under MS-DOS[6] [Kindler 91]. It translates lazy functional programs into parallelized combinator systems, which are translated into abstract machine code for the PAM by an additional compiler, also written in Turbo-Pascal [Hogen 91]. The abstract machine code must finally be transferred to the transputer system, where it is, up to now, interpreted by an implementation of the PAM on one, four and twelve transputers, respectively [Hogen 91].

Problem	*Oracle*	*Runtime (sec)*			*Speedup*				
		1 PE	*4 PE's*	*12 PE's*	*1 PE*	*4 PE's*		*12 PE's*	
					abs.	*rel.*	*abs.*	*rel.*	*abs.*
genfib(21)	*seq*	44.269	—	—	—	—	—	—	—
	comb	60.390	15.254	5.581	0.73	3.96	*2.90*	11.18	*7.93*
	nested	71.538	17.921	6.093	0.62	3.99	2.47	11.74	7.27
	max	71.564	17.921	6.087	0.62	3.99	2.47	11.76	7.27
genfib(22)	*seq*	71.618	—	—	—	—	—	—	—
	comb	97.707	24.569	8.739	0.73	3.98	*2.91*	11.57	*8.20*
	nested	115.748	28.950	9.749	0.62	4.00	2.47	11.87	7.35
	max	115.775	28.931	9.760	0.62	4.00	2.48	11.86	7.34
hanoi(12)	*seq*	7.938	—	—	—	—	—	—	—
	comb	8.460	2.151	0.795	0.94	3.93	*3.69*	10.64	*9.98*
	nested	13.788	3.475	1.224	0.58	3.97	2.28	11.27	6.49
	max	18.307	4.628	1.611	0.43	3.96	1.72	11.36	4.93
hanoi(13)	*seq*	15.876	—	—	—	—	—	—	—
	comb	16.920	4.266	1.496	0.94	3.97	*3.72*	11.31	*10.61*
	nested	27.578	6.917	2.369	0.58	3.99	2.30	11.64	6.70
	max	36.617	9.197	3.140	0.43	3.98	1.73	11.66	5.06
queens(6)	*seq*	31.169	—	—	—	—	—	—	—
	comb	33.373	8.775	3.299	0.93	3.80	*3.55*	10.12	*9.45*
	nested	49.940	13.062	4.834	0.62	3.82	2.39	10.33	6.45
	max	56.804	14.824	5.459	0.55	3.83	2.10	10.41	5.71
queens(7)	*seq*	147.219	—	—	—	—	—	—	—
	comb	157.518	40.833	14.520	0.93	3.86	*3.61*	10.85	*10.14*
	nested	238.108	61.298	21.713	0.62	3.88	2.40	10.97	6.78
	max	271.721	69.750	24.573	0.54	3.90	2.11	11.04	5.99

Table 1: Measurements on 1, 4 and 12 Transputers

In Table 1 the runtimes and speedups, that have been measured for the different parallelizations of three example programs on 1, 4 and 12 transputers, are reported. The example programs are the following:

- *genfib* computes for a given number i a list containing the first i fibonacci numbers, where each number is computed by a separate call of the fibonacci function.
- *hanoi* solves for a given number n the 'Towers of Hanoi' problem starting with n slices.
- *queens* computes for a given number n all solutions of the queens problem on an $n \times n$ chessboard.

[5]Turbo Pascal is a trademark of Borland International.
[6]MS-DOS is a trademark of Microsoft Corporation.

We distinguish two speedup values. The *relative speedup* is the ratio of the runtime of a parallelized program on one processor element (PE) to the runtime of the same program on 4 or 12 processors, respectively. The *absolute speedup* is the ratio of the runtime of the sequential version of the program on 1 processor to the runtime of the parallelized version on 4 or 12 processors, respectively.

The runtimes for the different parallelized versions on 1 processor element show that already the introduction of the *letpar*-construct causes a non-neglectable overhead, which is due to the additional combinator calls and the abstraction of the parallel expression in the *letpar*-expressions. The maximal parallelization of the *hanoi* program runs more than twice as long on one transputer as the sequential version.

Although the nested and maximal parallelizations show the best relative speedups, their performance with respect to the sequential programs, indicated by the absolute speedup, is very poor. The relative speedups can only be used to get an impression of the quality of the parallel implementation and the distributed process management. To rate the quality of the parallelization, one has to consider the absolute speedups. It turns out that the parallelization using the oracle *comb* shows the best absolute speedups and accordingly the best runtimes on 4 and 12 processors.

This is a consequence of the fact, that the strategy *comb* is especially suited to our implementation, because of the presupposition that parallel processes correspond to combinator applications. The other strategies have to introduce new combinators to meet this requirement, but additional combinator calls cause additional overhead. It becomes clear that the overhead of the additional parallelization is not compensated by the profits of the parallel execution. This might, however, be different for more complex problems.

5 Dynamic Parallelization of Higher Order Functions

An important advantage of functional programming languages is the modularity and abstraction that can be achieved by using higher order functions, which allow to define general recursion schemes that can then be adapted to special situations. Consider the higher order function *foldint*, given in Figure 7, that can e.g. be used to compute the factorial function: "$fac\ n^{\text{int}} = (foldint\ 1\ n\ id\ *)$", where "$id\ n = n$" is the identity combinator, or the list of the first n square numbers: "$squares\ n^{\text{int}} = (foldint\ 1\ n\ sqr\ append)$", where "$sqr\ i = (\textsc{cons}\ (*\ i\ i)\ \textsc{nil})$".

$$\begin{aligned} foldint\ i^{\text{int}}\ j^{\text{int}}\ f^{\text{int}\to *}\ g^{*\times *\to *} = &\ \textbf{if}\ (=\ i\ j)\ \textbf{then}\ (f\ i)\ \textbf{else let}\ med = (/\ (+\ i\ j)\ 2) \\ &\ \textbf{in}\ (g\ (foldint\ i\ med\ f\ g)\ (foldint\ (+\ med\ 1)\ j\ f\ g)) \end{aligned}$$

$*$ is a type variable that can be instantiated by an arbitrary type.

Figure 7: Example of a Higher Order Function

These examples show that the higher order function *foldint* will often be called with strict functional parameters, so that it would be possible and advantageous to evaluate the recursive calls in the body of this function in parallel. Our parallelizing compiler is, however, not able to detect this implicit parallelism, because the parallelization is done independently from the actual parameters of the function in the program. As the parallel evaluation of the recursive calls is not correct with respect to the underlying non-strict semantics (laziness) if the higher order function is called with non-strict functional parameters, a parallelization is not possible.

Consequently, programs may contain implicit parallelism that cannot be expressed explicitly using the *letpar*-construct of the intermediate language. To exploit this kind of parallelism, our implementation allows the parallel activation of arguments of indirectly activated combinator or base function applications. In order to avoid the generation of too small processes, a parallel activation takes only place if the argument expression is a combinator application. Note that the parallelizing compiler always transforms expressions, for which the *oracle* function allows a parallel execution, into combinator applications, no matter in which context (strict or non-strict) they occur.

In Table 2 it is shown, that the dynamic parallelization applies to the *foldint*-function when its functional argument g is instantiated by the strict base function $+$. Although no static parallelization takes place, reasonable speedups are obtained, because the indirect activation of the dynamically created application of

Problem	*Runtime (sec)*			*Speedup*	
	1 PE	*4 PE's*	*12 PE's*	*4 PE's*	*12 PE's*
(foldint 1 1000 id +)	9.124	2.659	1.053	3.43	8.66
(foldint 1 2000 id +)	18.252	5.244	1.951	3.48	9.36
(foldint 1 5000 id +)	45.638	13.032	4.719	3.50	9.67
(foldint 1 10000 id +)	91.273	25.990	9.321	3.51	9.79

Table 2: Measurements of Dynamic Parallelism

the base function + leads to the generation of parallel processes for the recursive calls of *foldint*. Note that the relative and absolute speedup values collapse, because the parallelized version of *foldint* is identical to the sequential version.

6 Conclusions and Future Work

We presented a technique for the automatic parallelization of lazy functional programs, which uses a strictness analyser, based on the technique of abstract interpretation, for the detection of implicit parallelism, and an oracle function to restrict the parallelization to expressions, for which a parallel execution seems to be profitable. In order to keep locality, we enable the activation of parallel subprocesses only immediately before the evaluation of the expression, that consumes the result of the subprocess, is initiated. The most natural and, with respect to our runtime experiments, best parallelization could be achieved by the oracle, that only allows the parallel evaluation of combinator applications.

A natural generalization of our parallelization algorithm will be to consider not only the direct subexpressions of an expression for the generation of the *letpar*-construct, but the maximal subexpressions for which the oracle allows a parallel evaluation. Then, the *comb* oracle will e.g. lift the recursive calls of the quicksort combinator to the same level, which is the parallelization of this algorithm, one usually has in mind.

A disadvantage of our parallelizing compiler is its time complexity, where most of the time is needed to compute the abstract interpretation of combinators. Therefore, it will be important to investigate whether alternative, more efficient analysis methods, as they have e.g. been proposed in [Hughes 90], [Nöcker 90], [Hartel et al. 91], can be incorporated in our parallelizing compiler.

Acknowledgements

The authors are grateful to the anonymous referees for their valuable comments.

References

[Abramsky 85] S.Abramsky: *Strictness Analysis and Polymorphic Invariance*, Workshop on Programs as Data Objects, LNCS 217, Springer Verlag 1985.

[Abramsky, Hankin 87] S.Abramsky, C.Hankin (eds.): *Abstract Interpretation of Declarative Languages*, Ellis Horwood Limited 1987.

[Burn et al. 86] G.Burn, C.L.Hankin, S.Abramsky: *Strictness Analysis for Higher-Order Functions*, Science of Computer Programming, Vol.7, November 1986.

[Burn 87a] G.Burn: *Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages*, Conf. on Functional Progr. Lang. and Computer Architecture, LNCS 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 88] G.Burn: *A shared memory parallel G-machine based on the evaluation transformer model of computation*, Workshop on the Impl. of Lazy Functional Languages, Report 53, Programming Methodology Group, Chalmers University of Technology, Göteborg 1988.

[Clack, Peyton-Jones 85] C.Clack, S.Peyton-Jones: *Strictness Analysis — A Practical Approach*, Conf. on Functional Progr. Languages and Computer Architecture, LNCS 201, Springer Verlag 1985.

[van Eekelen et al. 88] M.C.J.D.van Eekelen, M.J.Plasmeijer, J.E.W.Smeetsers: *Parallel Graph Rewriting on Loosely Coupled Machine Architectures*, Technical Report 88-9, University of Nijmegen 1988.

[Hammond, Peyton Jones 90] K.Hammond, S.Peyton-Jones: *Some Early Experiments on the GRIP Parallel Reducer*, Workshop on the Impl. of Functional Languages on Parallel Architectures, Technical Report no. 90–16, University of Nijmegen 1990.

[Hartel et al. 91] P.Hartel, H.Glaser, J.Wild: *On the Benefits of Different Analyses in the Compilation of Lazy Functional Languages*, Workshop on the Parallel Impl. of Functional Languages, Technical Report CSTR 91–7, University of Southampton 1991.

[Hogen 91] G.Hogen: *Integration of the Evaluator Technique in the Parallel Abstract Machine PAM*, Diploma Thesis, RWTH Aachen 1991 (in german).

[Hudak 86] P.Hudak: *Para-functional programming, a paradigm for programming multiprocessor systems*, ACM Symp. on Principles of Progr. Languages 1986.

[Hudak, Goldberg 85a] P.Hudak, B.Goldberg: *Serial Combinators: Optimal Grains of Parallelism*, Conf. on Functional Progr. Languages and Computer Architecture, LNCS 201, Springer Verlag 1985.

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

[Hudak, Young 86] P.Hudak, J.Young: *Higher Order Strictness Analysis in Untyped Lambda Calculus*, ACM Symp. on Principles of Progr. Languages, POPL 1986.

[Hughes 90] J.Hughes: *Compile-time analysis of functional programs*, in Turner (ed.): Research Topics in Functional Programming, Addison-Wesley 1990.

[Johnsson 85] Th.Johnsson: *Lambda Lifting*, Conf. on Functional Progr. Languages and Computer Architectures 1985, LNCS 201, Springer Verlag 1985.

[Kindler 91] A.Kindler: *Automatic Parallelization of Functional Programs*, Diploma Thesis, RWTH Aachen 1991 (in german).

[Kuchen, Wagener 90] H.Kuchen, A.Wagener: *Comparison of Dynamic Load Balancing Strategies*, Workshop on Distributed and Parallel Processing, Sofia 1990, Elsevier Publishers 1990.

[Lester, Burn 89] D.Lester, G.Burn: *An Executable Specification of the HDG-Machine*, Technical Report, GEC Research Centre, East Lane, Wembley, Middlesex HA9 7PP, United Kingdom.

[Loogen et al. 89] R.Loogen, H.Kuchen, K.Indermark, W.Damm: *Distributed Implementation of Programmed Graph Reduction*, Conf. on Parallel Architectures and Languages Europe 1989, LNCS 365, Springer Verlag 1989.

[Martin, Hankin 87] C.Martin, C.Hankin: *Finding Fixed Points in Finite Lattices*, Conf. on Functional Progr. Languages and Computer Architecture, LNCS 274, Springer Verlag 1987.

[Maurer 85] D.Maurer: *Strictness Computations Using Special λ-expressions*, Workshop on Programs as Data Objects, LNCS 217, Springer Verlag 1985.

[McBurney, Sleep 90] D.McBurney, R.Sleep: *Concurrent Clean on ZAPP*, Workshop on the Impl. of Functional Lang. on Parallel Architectures, Technical Report no. 90–16, University of Nijmegen 1990.

[Mycroft 81] A.Mycroft: *Abstract Interpretation and Optimizing Transformations for Applicative Programs*, Ph.D. Thesis, University of Edinburgh, 1981.

[Nöcker 90] E.Nöcker: *Strictness Analysis Using Abstract Reduction*, Workshop on the Impl. of Functional Languages on Parallel Architectures, Technical Report no. 90–16, University of Nijmegen 1990.

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

[Vrancken 90] J.Vrancken: *Reflections on Parallel Functional Languages*, Workshop on the Impl. of Functional Lang. on Parallel Architectures, Technical Report no. 90–16, University of Nijmegen 1990.

[Wadler 87] P.Wadler: *Strictness analysis on non-flat domains (by Abstract interpretation over finite domains)*, in: [Abramsky, Hankin 87].

[Wadler, Hughes 87] P.Wadler, J.Hughes: *Projections for Strictness Analysis*, Conf. on Functional Progr. Languages and Computer Architecture, LNCS 274, Springer Verlag 1987.

[Young, Hudak 86] J.Young, P.Hudak: *Finding Fixpoints on Function Spaces*, Research Report YALEU/DCS/RR-505, Yale University, December 1986.