Distributed Applicative Arrays

Herbert Kuchen and Günther Geiler*

Lehrstuhl für Informatik II, RWTH Aachen

Abstract

Lists, the typical data structures of functional programming languages, force a strictly sequential computation of their elements and hence are badly suited for parallel implementations. Some other applicative data structures are proposed, which are appropriate for implementations of such languages on loosely coupled multiprocessor systems. Besides so called sequences, i.e. list-like structures internally represented by binary trees, we mainly consider a distributed applicative version of arrays. Some experimental results allow a comparison of those data structures.

1 Introduction

In recent years, several multiprocessor systems have been developed in order to offer high computation power (at a reasonable price). Such systems are either tightly coupled (e.g. the SEQUENT symmetry), i.e. all processors share the same memory, or loosely coupled, i.e. the processors have only local memory and communicate via message passing. The latter will be called ***multicomputer*** in the sequel, each processor together with its local store will be called processing element (PE). While the first approach is limited to about 30 processors, the second allows topologies of arbitrary size.

Since we are mainly interested in large multiprocessor systems, we shall only consider multicomputers in the following. Typical representatives of such systems are Transputer systems [IN87] and the Intel iPSC [Qu87].

Now the question arises, with which language such a system should be programmed. For very regularly structured problems like the numerical solution of partial differential equations, parallel imperative languages with explicit parallelism and explicit communication between processes work rather well.

For problems with a more complex structure, such languages lead to programs, which are very hard to debug. One reason is that errors and deadlocks in those typically nondeterministic programs are hardly reconstructable.

Here languages with implicit communication and (preferably) implicit parallelism are needed. The runtime system ensures that the communication is correct and deadlock free.

One possible choice is to use a parallely implemented functional language. A functional program consists of a sequence of mutually recursive function definitions (in the mathematical sense) and an expression (containing those functions), which is to be

*Ahornstr. 55, D-5100 Aachen, email: kuchen@informatik.rwth-aachen.de

evaluated. The right hand side of a function definition consists of an expression, e.g. $f\ e_1 \ldots e_n$[†]. Since the evaluation of an expression does not cause side effects, the subexpressions $e_1, \ldots, e_n$ may be evaluated in arbitrary order and especially in parallel. This implicit parallelism may be exploited by a parallel implementation.

Of course, not every subexpression is worth computing in parallel. If e.g. it only consists of adding two numbers, the effort of spawning an own process and possibly moving this process to another processor does not pay.

There are different opinions, as to whether the compiler can automatically decide, which subexpressions are "big enough", or whether the programmer needs to insert annotations indicating which expressions are expensive.

In either case, the communication is implicit and transparent to the programmer. Each process may spawn subprocesses representing subexpressions. Each subprocess eventually delivers the result to the father process.

Which data structures should by offered by a functional language implemented on a multicomputer? First of all, operations on those data structures have to work without side effects. Such data structures are called applicative in the following. Something like changing the ith element of an array A to x by $A[i] := x$ is not allowed. It is only possible to generate a new version B of a structure A, where B only differs from A in the ith element:

$$let\ B\ =\ (upd\ A\ i\ x)\ in\ \ldots\text{main expression}\ldots$$

A remains unchanged.

In certain situations a program analysis may find out that A is not needed any more [HB85]. In this case, an update in place should be performed, since this will save time and space, independently of the considered data structure.

Of course it is too expensive, if A is first copied and then changed. Hence, applicative data structures should allow to store different versions in an overlapping way.

On a multicomputer, a data structure needs to be distributed among the stores of the different processes. Storing it in one store leads to a bottleneck, since all accesses are sequentialized by the corresponding processor.

Furtheron, the data structures must be suitable for parallel computation, i.e. they should be divisible into substructures of nearly equal size to allow divide and conquer computations. Moreover, they should not contain a global component, which needs to be passed by every access, since this would also establish a bottleneck.

Also, the basic operations working on a data structure need to be efficient. Such operations are for example:

- $hd\ S$: accessing an outstanding element of a structure S, e.g. the head of a list
- $sel\ S\ i$, $upd\ S\ i\ x$, $ins\ S\ i\ x$, $del\ S\ i$: accessing (e.g. selecting, updating, inserting, or deleting) one element of a structure S
- $map\ f\ S$, $fold\ (\oplus)\ S$, $zip\ (\oplus)\ S_1\ S_2$: working with whole structures using higher order functions
- $left\ S$, $right\ S$: decomposing a structure S into parts of nearly equal size

[†]In our examples, we use Miranda syntax. We hope, it is quite understandable. If something remains unclear, refer to [Tu90].

- *append* S_1 S_2: concatenating two structures S_1 and S_2
- *gen* f n: generating a new structure with elements f $1, \ldots,$ f n

We will use the same operation names for all data structures. From the context, it will be clear which of them is meant. Not all mentioned operations are available for all considered data structures. Sometimes the number and order of arguments will differ. To enable an efficient parallel implementation, we demand that the first argument of ***fold*** is associative. For a detailed description of those operations for lists refer to the appendix. For other data structures, they are defined accordingly [Ku90].

The mentioned properties are essential for distributed applicative data structures. Other properties are also of interest. Hughes [Hu90] shows how infinite objects, in particular infinite lists, can be used to write modular and elegant programs. Hence, it is interesting whether a data structure allows to build infinite objects. Another important property of a data structure is, whether it allows to insert and delete elements. Futheron, the different data structures differ in their influence on work load distribution.

The rest of this paper is organized as follows. First, different applicative data structures are compared basing on the mentioned requirements. Then, we show some experimental results for a couple of example programs. These examples have been tested on a parallel implementation of a Miranda-like functional language on a system of Transputers [LK89]. The runtime system of this implementation is founded on the parallel abstract machine PAM, which is based on graphreduction.

2 Applicative Data Structures

2.1 Lists

The typical data structure of every functional language is the ***list***. It is usually predefined, but it can be easily introduced by the following algebraic type definition (or similar):

$$list\ * \ ::= \ Nil \mid Cons\ *\ (list\ *)$$

i.e. a list of arbitrary component type $*$ is either the empty list represented by Nil or constructed out of an element of type $*$ and another list of the same component type using a binary constructor $Cons$. A straightforward implementation will use an own storage cell for each constructor application (see figure 1). Each cell will point at the cells representing the components. Obviously, different versions of a list (here

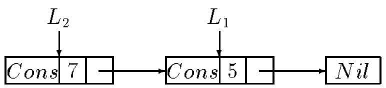


Figure 1: lists $L_1 = (Cons\ 5\ Nil)$ and $L_2 = (Cons\ 7\ (Cons\ 5\ Nil))$ (straightforward internal representations)

L_1 and L_2) can be stored in an overlapping way. Also, lists are dynamic, since they may be extended or shortened at the left hand side. If the lists are evaluated lazily, infinite lists can be handled as long as only a finite prefix of them is really needed. Lists

are efficient, if all elements need to be accessed, but they only allow a slow access to individual arbitrary elements. Table 1 shows the complexities $T_1(n)$ and $T_\infty(n)$ of the basic operations depending on the size n of the argument list(s). $T_1(n)$ is the complexity on 1 PE, while $T_\infty(n)$ is the complexity on as much PEs as tasks are available. Since we want to abstract from the network topology, the communication costs are ignored. For a concrete network, $T_\infty(n)$ needs to be multiplied by the diameter, e.g. by $log(p)$ for cube connected cycles, where p denotes the number of PEs used.

operation	$T_1(n)$	$T_\infty(n)$
hd, tl, cons	$O(1)$	$O(1)$
sel, upd, insert, del	$O(n)$	$O(n)$
map, fold, zip, gen	$O(n)$	$O(n)$
left, right	$O(n)$	$O(n)$
append	$O(n)$	$O(n)$

Table 1: complexities of list operations

Although each element may in principle be stored in another store, lists are obviously not suited for parallel computation, because elements can only be accessed sequentially.

2.2 Tree Structures

In order to solve this problem, one can try to find a data structure with operations similar to those of lists but with a different internal representation (transparent to the user).

2.2.1 Sequences

One possibility is to use binary trees as internal representations. The resulting data structures will be called ***sequences***. In Miranda they could be defined as:

$$sequence\ * \ ::= \ Emptyseq \mid Seqleaf\ * \mid Seqnode\ (sequence\ *)\ (sequence\ *)\ num$$

The number l stored in an internal node ($Seqnode$ t_1 t_2 l) (for some subsequences t_1 and t_2) specifies the length of the corresponding sequence (i.e. the number of leaves of the tree). The real elements are only stored in the leaves and not in the internal nodes. This is better for divide and conquer algorithms, since otherwise at each step the element stored in the root needs a specific treatment. The length of the left subsequence is used to decide, whether an access to an element has to be directed to the left or right subtree.

Sequences are easily divisible among the different stores, since each node may in principle be placed in another store. Also, they are suited to divide and conquer algorithms, because the subsequences represented by the left and right subtrees can be handled in parallel. Figure 2 shows the internal representation of a sequence A consisting of the elements 19, 14, 28, and 12 and a sequence $B = (ins\ A\ \ 2\ \ 39)$ (i.e. a sequence generated by inserting a new element 39 after the second element of A).

Obviously generating a new sequence B by inserting a new element into an existing sequence A only requires adding a path from the new root to the new leaf. The rest

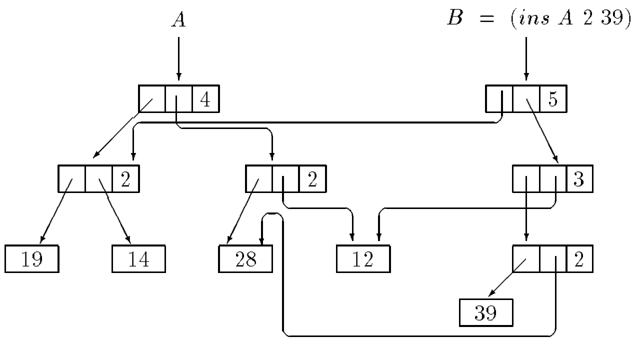


Figure 2: internal representation of sequences

of the structures A and B is shared. Deleting and updating elements can be handled analogously. Hence, sequences are also dynamic.

Compared to accessing the elements in arrays of imperative languages, sequences are still slow, especially when the internal representations degenerate to list-like structures (see table 2). To overcome the bad worst case behaviour, balanced trees could be considered. This would result in a structure like AVL-DAGs [My84]. Experiments show that this is not worthwhile, since in a typical and good functional program a sequence is not generated element by element but by some higher order function like *map*, *zip* and *gen*. Those functions are internally implemented in a divide and conquer style. Hence, the sequences are automatically balanced. Additional overhead does not pay. Table 2 shows the complexities of the basic operations on sequences.

operation	$T_1(n)$ balanced* case	$T_1(n)$ worst case	$T_\infty(n)$ balanced* case	$T_\infty(n)$ worst case
hd, sel, upd, insert, del	$O(log\ n)$	$O(n)$	$O(log\ n)$	$O(n)$
map, fold	$O(n)$		$O(log\ n)$	$O(n)$
zip	$O(n\ log\ n)$	$O(n^2)$	$O(log\ n)$	$O(n^2)$
gen	$O(n)$		$O(log\ n)$	
left, right	$O(1)$		$O(1)$	
append	$O(1)$		$O(1)$	

Table 2: complexities of the operations on sequences

left and ***right*** just select the left and right subtree respectively. If they are implemented in this way, they may only be used internally (e.g. to define *map*, *fold*, and so on) and

*We use the notion *balanced case* instead of the usual *average case*, since it is hard to say, what average might mean for a functional program.

not by the programmer. Otherwise the referential transparency is lost, since $S_1 = S_2$ does ***not*** imply $left\ S_1 = left\ S_2$ and $right\ S_1 = right\ S_2$.

When manipulating a sequence element by element all accesses to the elements have to pass the root of its internal representation. Hence, they are sequentialized by the PE storing the root. Infinte sequences are not possible, since the length of the sequence is needed to guide the search and has to be inserted into the root of its internal representation.

So, sequences only allow efficient parallel programs, if they are mainly treated monolithically e.g. by some higher order functions. Fortunately, this is exactly the recommanded style of functional programming.

Note that sequences need approximately twice as much storage as lists, because of the internal nodes.

2.2.2 Tables

In [Ku90] some data structures similar to sequences are considered, too. They are not explained in detail here, since they behave very similarly to sequences. They are called ***tables*** and ***sets***.

Tables can be defined as:

$$table\ * \ ::=\ Emptytab \mid Tableaf\ num\ * \mid Tabnode\ (table\ *)\ (table\ *)\ num$$

If a leaf of the internal representation of a table T is $(Tableaf\ i\ x)$, this means that the component of T with index i has value x. The operations working on tables ensure that the index j contained in an internal node $(Tabnode\ t_1\ t_2\ j)$ is greater or equal than that of each leaf in t_1 and less than that of each leaf in t_2. Consequently, an access to the ith element is guided to t_1, if $i \leq j$, and to t_2 otherwise. Tables are especially suited for sparse matrices, since only the non-zero elements need to be stored.

2.2.3 Sets

The internal representation of sets can be based on the following algebraic type:

$$set\ * \ ::=\ Emptyset \mid Setleaf\ * \mid Setnode\ (set\ *)\ (set\ *)\ *$$

The real elements are only stored in the leaves, too. Each internal node contains an element to guide the search. Since searching elements includes comparing the desired element with elements stored in the internal nodes, the elements cannot be evaluated lazily. Furtheron, only sets of such types can be built, which allow a comparison of elements. For instance sets of functions are forbidden.

2.2.4 Applicative Quad Trees

Applicative 2^d-ary trees called ***quad trees*** were introduced by Wise [Wi87] to represent sparse d-dimensional matrices†. Insertions and deletions are forbidden. An $n \times n$-matrix is embedded into an $m \times m$-matrix M, where $m\ :=\ 2^{\lceil log\ n \rceil}$. If $m \leq 2$, the matrix elements

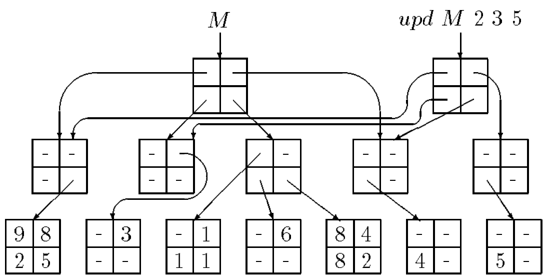
†In the following, only $d = 2$ is considered.

are directly inserted into a new leaf node. If $m > 2$, M is divided into four $(m/2) \times (m/2)$-matrices, which are handled analogously. Furtheron, a new root is generated pointing at the roots of the trees, which have been produced for the submatrices.

A pointer at a node containing only zeros is replaced by *nil*. The same happens to a node, which only contains *nil* pointers.

In [Wi87] quad trees are used for various matrix algorithms. The amount of storage needed for special matrices (triangular, tridiagonal,...) is shown to be linear in the number of non-zero elements.

Figure 3 shows the representation of a 6×6-matrix.



$$M = \begin{pmatrix} 9 & 8 & 0 & 0 & 0 & 0 \\ 2 & 5 & 4 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 6 & 8 & 4 \\ 0 & 0 & 0 & 0 & 8 & 2 \end{pmatrix}$$

Figure 3: quad tree representation of a tridiagonal 6×6matrix M (embedded into an 8×8-matrix) and its updated version (*upd* M 2 3 5)

The height of the tree is $O(\log\ n)$. The complexities for *sel*, *upd*, *map*, *fold*, *submatrix* (instead of *left* and *right*), and *gen* are the same as for sequences (see table 2). *zip* works in linear time on 1 processor and in logarithmic time on an arbitrary number of processors. Updates can be handled analogously to that of sequences (see figure 3).

2.3 Distributed Applicative Arrays

For all data structures shown so far accessing an element is slow compared to arrays of imperative languages. In this subsection an applicative variant of arrays called distributed applicative array (DAA) is presented, which allows to access the elements in constant time (ignoring communication costs).

2.3.1 Basic Ideas

While the previously mentioned data structures may in principle be based on algebraic data types, the abstract machine, on which the implementation is founded, needs to be extended by additional commands for the following basic DAA operations:

- *gen f n* generates a new DAA with components $f\ 1, \ldots, f\ n$
- *sel A i* selects the ith component of DAA A

The other operations like *map*, *fold*, and *zip* can be defined based on those two operations. For efficiency reasons, at least *fold* should be implemented directy as an additional

command of the abstract machine, too. DAAs are not dynamic, i.e. operations to insert, delete, or update elements are not provided. An extension of DAAs by such operations will be discussed later.

Consequently, DAAs can neither be generated element by element nor by repeatedly concatenating subarrays, but must be handled monolithically. This has of course some impact on the programming style.

For an implementation on a multicomputer, it is not reasonable to store a DAA in one contiguous block of memory, because then all accesses to elements would be sequentialized by the corresponding processor. Hence, a real distribution among all the stores is needed. Since the DAAs are generated at run time, the allocation of storage also must be done at run time. The problem is that a PE creating a DAA does not know, which storage is free on the other PEs. Therefore, each element is represented by a virtual address, and each PE maintains a translation table indicating which storage cell is allocated to each of the DAA elements assigned to it. A virtual address consists of

- an identification of the array (e.g. a pair consisting of the number of the creating PE and a running array number on this PE)
- an element index.

Given a virtual addresss, each PE can compute to which PE the corresponding DAA element is given. No global component (and hence no bottleneck) like the root of a tree needs to be passed in order to access an element.

When a PE creates a (part of a) DAA, it first sends messages to its immediate neighbours, indicating which parts of the DAA they shall create. Then, it generates tasks to compute those elements, which are bound to it, and inserts pointers at them into its translation table. Each neighbour proceeds analogously. The tasks are started, when the corresponding DAA elements are accessed, i.e. the elements are computed lazily. The distribution mechanism will ensure that each PE receives at most one creation order for each array.

An attempt to access a DAA element may occur before the creating message arrives. In this case a pointer at the place, to which the element needs to be sent, is stored in the translation table. When the element is eventually ready, it will be sent to all places, where it is needed. This behaviour is similar to that of I-structures [AN90], which are used in the data flow language Id. Figure 4 shows, how a DAA is created and accessed.

Table 3 shows the complexity of the basic operations on DAAs. During the computation of $T_\infty(n)$ for *map*, *fold*, *zip*, and *gen*, we assume that no PE sends more than a constant number of orders to its neighbours.

An important advantage of DAAs (compared to lists and sequences) is that the number of messages for operations like *map*, *fold*, and *gen* only depends on the number of PEs and not on the number of DAA elements. Each PE receives at most one *map*, *fold*, or *gen* message, and in the case of *fold* returns a result. The rest of the operation is done locally. Of course additional messages are needed, if the description of the argument function is not completely contained within the message or if the evaluation of the argument function causes additional messages.

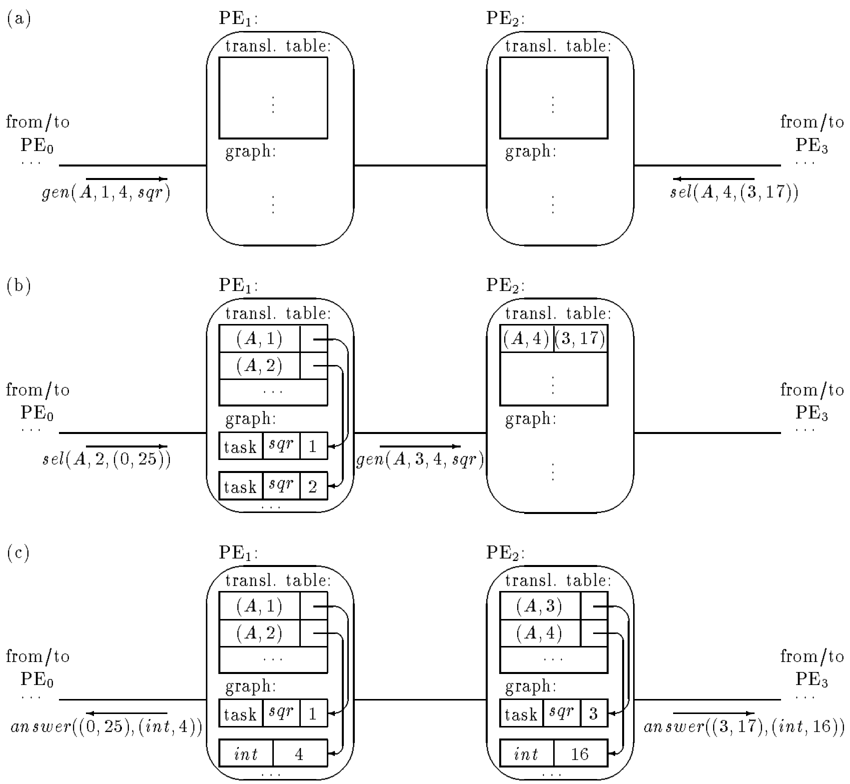


Figure 4: (a) PE$_1$ receives an order to generate $A[1]$ to $A[4]$ using the sqr-function; at the same time PE$_3$ tries to access $A[4]$ on PE$_2$. (b) PE$_1$ creates tasks for $A[1]$ and $A[2]$ (we assume that those elements are bound to it) and inserts pointers at them into its translation table. Furtheron, PE$_2$ is informed that it shall generate $A[3]$ to $A[4]$. At the same time, PE$_2$ notices that $A[4]$ has not yet been created and inserts an entry into its translation table. This entry indicates to which address the element has to be sent, when it is ready. Simultanously, PE$_0$ wants to access $A[2]$. (c) PE$_1$ evaluates $A[2]$ and returns the result to the requesting PE$_0$. PE$_2$ creates tasks for $A[3]$ and $A[4]$ (which may be bound to it) and inserts corresponding entries into its translation table. Since $A[4]$ is already requested, it is evaluated and send to the destination (local address 17 on PE$_3$).

2.3.2 Modification Cells

Modifications of DAAs by inserting, deleting, or updating elements can be allowed, if modification cells are used, which record in which way the new DAA differs from another one, at which this cell points.

If many modifications are executed, a tree of such cells will be produced. [AH87]

operation	$T_1(n)$	$T_\infty(n)$
hd, sel	$O(1)$	$O(1)$
map, fold, zip, gen	$O(n)$	$O(log\ n)$

Table 3: complexities of DAA operations

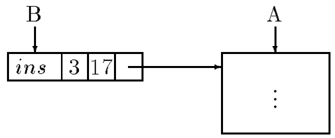


Figure 5: representation of $B = (ins\ A\ 3\ 17)$ using a modification cell; if $B[3]$ is searched, it is sufficient to access the cell, otherwise the pointer needs to be followed.

show, how such a tree may be restructured to allow a fast access to the most interesting (e.g. the most recently generated) version.

In the worst case an access to a DAA element has to pass all modification cells until the "real" DAA is found. Hence, all the complexities in table 3 have to be multiplied by the number of modifications. Also, each modification cell establishes a bottleneck, since it sequentializes the accesses to the DAA elements. Consequently, modification cells should be avoided, if possible.

2.3.3 Consequences on Work Load Distribution

DAAs influence the work load balance, since the tasks working on them are sent to the elements. Hence, the work load distribution depends on the element distribution mechanism, which is fixed at compile time, and it is possible that an overloaded PE will receive additional work.

These problems do not occur, if lists or tree structures are used. Here, the work load distribution only depends on the load of the PEs [KW90].

2.3.4 Distribution Mechanism

Next, we want to see how the elements of a DAA should be distributed among the different stores. First of all, the DAA should be divided into contiguous blocks, where each PE receives one block and where neighbour blocks are given to neighbour PEs. Remember that in order to reduce the communication overhead $fold$ is implemented in such a way that on each PE the local elements are first combined, and then the intermediate result is returned. If a non-contiguous distribution is used, the argument function of $fold$ consequently needs to be commutative, which is not the case in all interesting applications. On the other hand, a non-contiguous distribution has no considerable advantages.

Also, the blocks should have nearly the same size to get a well balanced work load distribution.

Another question is, whether the distribution of elements should depend on the DAA identification or only on the index. In many example programs, a new version B of a DAA A is built, where each element $B[i]$ depends on $A[i]$ (and perhaps on $A[i-1]$ and $A[i+1]$). Here, it is advantageous, if the distribution only depends on the index, because then (nearly) all elements concerned are placed on the same PE.

The translation table on each PE can for instance be implemented by a hashing table with open addressing. The hash function depends on the DAA identification and the index of the considered element.

3 Experimental Results

Now, we want to compare the data structures sequences and distributed applicative arrays based on the run times and speedups for some example programs. These examples have been written in a Miranda-like functional language, which is parallely implemented on a system of Transputers [LK89][Ku90]. The run time system of this implementation is based on the parallel abstract machine PAM, which performs compiled graph reduction, i.e. each reduction step is compiled into a sequence of elementary graph manipulation commands. Here we consider a PAM implementation, where each such command is interpreted in OCCAM. A new PAM implementation, which directly produces Transputer machine code, works approximately 8 times faster, but unfortunately DAAs are not yet included.

The following example programs have been tested:

- *map* (+1) [1..n]
- *fold* (+) [1..n]
- multiplication of a $n \times n$-matrix with itself

Due to space limitations, we only give a small subset of our experimental results here. More can be found in [Ge90].

Figure 4 shows the experimental results for each example program depending on the argument size. For the data structures sequences and DAAs, we give the run times on one PE and the speedups on 12 and 48 PEs. Also, the quotient q_i of the run times for sequences and DAAs on i PEs ($i = 1, 12, 48$) is shown.

For lists, we only give the run times on one PE. Because of the sequential nature of lists, there is no considerable speedup for *fold* and *map* on more than one PE. For matrix multiplication, the situation is different. Here, only small lists are used, and each element needs a lot of computations. In such situations, lists allow a good speedup (between 7 and 8 for matrix multiplication on 12 PEs), too.

The run times on 1 PE are faster for lists than for sequences (but slower than for DAAs). This is caused by the additional overhead for internal nodes of sequences.

The speedups using DAAs are rather good and, especially for *map*, close to the number of PEs. For sequences, they are a bit worse, especially for smaller argument sizes. DAAs are between 3 and 15 times faster then sequences. This advantage increases the more processors are available.

		lists	sequences			DAA			seq./DAA		
problem	n	sec	sec	s12	s48	sec	s12	s48	q_1	q_{12}	q_{48}
fold	1000	5.2	12.6	7.3	18.5	3.5	8.7	20.1	3.6	4.3	4.1
	2500	12.8	31.2	8.0	23.2	8.8	9.1	25.7	3.6	4.1	4.0
	5000	25.6	62.5	8.3	26.8	17.6	9.4	28.7	3.6	4.0	4.3
	10000	51.3	125.1	8.6	28.4	35.7	9.6	29.2	3.5	3.9	4.3
	15000	76.9	187.6	8.9	30.1	52.7	9.4	29.6	3.6	3.8	4.1
map	1000	4.8	13.1	7.2	15.7	2.7	11.2	38.2	4.8	7.6	11.8
	2500	12.0	32.7	8.2	22.0	6.8	11.4	39.7	4.8	6.8	8.5
	5000	23.9	65.5	8.4	26.1	13.5	11.5	40.7	4.9	6.6	8.3
	10000	47.8	130.7	8.4	28.5	27.0	11.5	41.1	4.9	6.6	7.8
	15000	71.7	196.2	8.4	29.2	40.5	11.5	41.3	4.9	6.6	7.6
matmult	10	47.1	86.1	8.3	19.2	10.0	7.6	21.0	8.6	7.8	10.9
	20	-	824.6	8.4	22.8	78.9	8.0	26.9	10.5	10.0	15.1
	30	-	-	-	-	264.8	8.1	27.1	-	-	15.6
	40	-	-	-	-	625.4	8.0	27.6	-	-	-

Table 4: Run times in seconds on one PE and speedups on 12 (s12) and 48 (s48) PEs for some example programs depending on the argument size n. q_i is the quotient of run times for sequences and DAAs on i PEs (i=1,12,48). '-' indicates that the value could not be measured due to space limitations.

		sequences			DAA		
problem	n	*#mess*	*#comm*	*#nodes*	*#mess*	*#comm*	*#nodes*
fold	2500	2500	40	1500	10	10	900
map	2500	5000	60	3000	10	16	1700
matmult	10	9000	110	800	900	15	700

Table 5: For some example programs it is shown which costs have approximately been produced by each of 12 PEs depending on the data structure. *#mess* is the number of messages sent, *#comm* is the number of PAM-commands executed (in thousands), and *#nodes* is the number of graph nodes needed.

Table 5 shows that DAAs need much fewer messages, PAM-commands, and graph nodes than sequences. Especially for *fold* and *map* the number of messages only depends on the number of PEs and not on the argument size.

It is important to note that for DAAs *map* works much slower, if the argument function is not completely contained in the creation order. For instance *map* (+1) [1..15000] will be about 4 times slower on 48 PEs, since than the PE storing the representation of the argument will continuously be asked for this information and will become a bottleneck.

We may conclude that DAAs are preferable where a monolithical treatement of the data structure is sufficient. Sequences are more flexible and can be used in situations where a large data structure is built by repeatedly concatenating substructures (e.g. quicksort, towers of Hanoi [Ku90]). In a multicomputer environment, lists are only appropriate, if they are rather small and each element needs a lot of computations.

Acknowledgement

The authors thank Maria Kuchen, Rita Loogen, and Andreas Maaßen for helpfull comments.

References

[AH87] A. Aasa, S. Holmström, C. Nilsson: An Efficient Comparison of Some Representations of Purely Functional Arrays, internal report, Programming Methodology Group, Göteborg, 1987

[AN90] Arvind, R.S. Nikhil: Executing a Program on the MIT Tagged-Token Dataflow Architecture, IEEE Tr. on Computers, Vol. 39(3), 1990, 300-318

[Ge90] G. Geiler: Implementierung verteilter applikativer Arrays, Diplomarbeit, RWTH Aachen, 1990, (in german)

[HB85] P. Hudak, A. Bloss: The Aggregate Update Problem in Functional Programming Systems, ACM Symp. on Principles of Programming Languages, 1985, 300-314

[Hu90] J. Hughes: Why Functional Programming Matters, in: Research Topics in Functional Programming, Addison Wesley, 1990, 17-42

[IN87] INMOS: The Transputer Instruction Set – A Compiler Writers Guide, INMOS Ltd., Bristol, England, 1987

[Ku90] H. Kuchen: Parallele Implementierung einer funktionalen Programmiersprache auf einem OCCAM-Transputer-System unter besonderer Berücksichtigung applikativer Datenstrukturen, PhD. Thesis, RWTH Aachen, 1990, (in german)

[KW90] H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies, Aachener Informatik-Berichte Nr. 90-5, RWTH Aachen, also to appear in: Proceedings Workshop on Distributed and Parallel Processing, Sofia, Elsevier Publishers, 1990

[LK89] R. Loogen, H. Kuchen, K. Indermark, W. Damm: Distributed Implementation of Programmed Graph Reduction, Conf. on Parallel Architectures and Languages Europe, LNCS 365, Springer Verlag, 1989, 136-157

[My84] E. Myers: Efficient Applicative Data Types, ACM Symp. on Principles of Programming Languages, 1984, 66-75

[Qu87] M.J. Quinn: Designing Efficient Algorithms for Parallel Computers, McGraw-Hill, 1987

[Tu90] D.A. Turner: An Overview of Miranda, in: Research Topics in Functional Programming, Addison Wesley, 1990, 1-16

[Wi87] D.S. Wise: Matrix Algebra and Applicative Programming, Conf. on Functional Programming and Computer Architectures, LNCS 274, Springer Verlag, 1987, 134-153

A Basic operations and Examples

basic functions on lists:

```
hd (Cons x L) = x
sel (Cons x L) 1 = x
sel (Cons x L) (i+1) = sel L i
ins L 0 y = Cons y L
ins (Cons x L) (i+1) y = Cons x (ins L i y)
upd (Cons x L) 1 y = Cons y L
upd (Cons x L) (i+1) y = Cons x (upd L i y)
del (Cons x L) 1 = L
del (Cons x L) (i+1) = Cons x (del L i)
map f (Cons x L) = Cons (f x) (map f L)
map f Nil = Nil
fold g (Cons x (Cons y L)) = g x (fold g (Cons y L))
fold g (Cons x Nil) = x
zip g (Cons x L) (Cons y R) = Cons (g x y) (zip g L R)
zip g Nil Nil = Nil
gen f n = generate f 1 n
          where generate f i n = Cons (f i) (generate f (i+1) n), i <= n
                               = Nil, otherwise
length (Cons x L) = 1 + length L
length Nil = 0
left L = prefix L (length L div 2)
         where prefix L 0 = Nil
               prefix (Cons x L) (i+1) = Cons x (prefix L i)
right L = postfix L (length L div 2)
          where postfix L 0 = L
                postfix (Cons x L) (i+1) = postfix L i
append (Cons x L) R = Cons x (append L R)
append Nil L = L
```

matrix multiplication on lists:

```
foldint l u f g = f l, l = u
                = g (foldint l ((l+u) div 2) f g)
                    (foldint ((l+u) div 2 + 1) u f g), otherwise
matmult A B n = foldint 1 n row append
                where row i = Cons (foldint 1 n (sprod i) append) Nil
                      sprod i j = Cons (foldint 1 n (mult i j) (+)) Nil
                      mult i j k = (sel (sel A i) k) * (sel (sel B k) j)
```

basic operations on DAAs:
gen, *sel* and *fold* are predefined and the following equations hold:

sel (gen f n) i == f i if $1 \le i \le n$
fold g S n == foldint 1 n (sel S) g

the other basic operations on DAAs are defined as follows:

$$\begin{array}{l}
map\ f\ S = gen\ g \\
\qquad\qquad where\ g\ i = f\ (sel\ S\ i) \\
zip\ g\ L\ R = gen\ h \\
\qquad\qquad\quad where\ h\ i = g\ (sel\ L\ i)\ (sel\ R\ i)
\end{array}$$

matrix multiplication on DAAs ($A[i,j] = sel\ A\ ((i-1)*n+j)$):

$$\begin{array}{l}
matmult\ A\ B\ n = gen\ f\ (n*n) \\
\qquad where\ f\ k = sum\ ((k\text{-}1)\ div\ n + 1)\ ((k\text{-}1)\ mod\ n + 1)\ 1\ n \\
\qquad\qquad sum\ i\ j\ l\ u = (sel\ A\ ((i\text{-}1)*n+l))\ * \\
\qquad\qquad\qquad\qquad\quad (sel\ B\ ((l\text{-}1)*n+j)),\ l = u \\
\qquad\qquad\qquad\qquad = (sum\ i\ j\ l\ ((l+u)\ div\ 2)) + \\
\qquad\qquad\qquad\qquad\quad (sum\ i\ j\ ((l+u)\ div\ 2 + 1)\ u),\ otherwise
\end{array}$$