A Model for the Compilation of Dataflow Languages on Conventional Parallel Architectures

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

November 1991

Abstract

We present a new model $\mathcal{DF}$ for the implementation of non-strict but non-lazy functional languages (dataflow languages) on conventional parallel hardware. $\mathcal{DF}$ consists of a small set of functions that can be added to any imperative language and allow to express dataflow behaviour in a simple and efficient way. The purpose of this model is to provide a high-level and machine-independent target language to which dataflow languages can be compiled for parallel execution. A binding of the $\mathcal{DF}$ model to the C language has been implemented on a multi-transputer system. Several manually translated dataflow programs show very promising results.

1 Introduction

The programming of parallel computers in functional languages gains more and more interest. This interest is motivated by the fact that conventional imperative languages are based on a sequential execution semantics, which makes them not very suitable for parallel machines: on the one hand, the capabilities of automatically parallelizing compilers are still restricted to FORTRAN DO-loops iterating on arrays (which is of importance for numerical applications only); on the other hand, parallel extensions to imperative languages burden a human programmer with an additional level of complexity and make parallel programming a difficult and error-prone task.

Functional languages do not suffer from these basic problems. A functional program is a set of equations that define mathematical functions, i.e. descriptions of how some wanted object can be constructed from other given ones. The temporal order in which functions are to be applied is not specified in the program. Hence, functions that do not depend on the result of each other may be executed in any order, in particular they may be executed in parallel (*horizontal parallelism*, e.g. between the functions g and h in the expression $f(g(e_1), h(e_2)))$. In other words, functional languages do not overspecify the order of program execution which offers lots of possibilities for automatic parallelization.

Additional parallelism is provided by functional languages with *non-strict semantics*. While in strict languages a function can be applied only to fully evaluated arguments, in non-strict languages function application is decoupled from argument evaluation. On parallel machines, this feature may be exploited by executing a function in parallel with the evaluation of its arguments (*vertical parallelism*, e.g. between f and g in $f(g(e))$). This makes sense in the case that a function does not immediately need (all of) its arguments to proceed execution, in particular if an argument is a compound data structure that can be produced in parallel with its consumption.

Today research concentrates on functional languages where non-strictness is achieved by *lazy evaluation*. In this execuction model, an expression is evaluated when its value is actually demanded (demand driven execution) and no expression is evaluated more than once (full laziness). While laziness contributes to the expressive power of functional languages (e.g. by potentially infinite data structures), this feature is quite costly. Today, the standard technique for implementing lazy evaluation is *programmed graph reduction* [Peyton Jones, 1987], where each function is compiled into a piece of code that transforms the graph representing the expression to be evaluated. This method can be generalized to a parallel graph reduction model as well [Loogen, 1990].

The most important example of non-strict but non-lazy functional languages are the so called *dataflow languages*: in the dataflow model, program execution is controlled by the availability of data (data-driven execution). Each expression is evaluated as soon as possible (i.e. when the required data are available) regardless whether its value is necessary for the final result or not. Hence, in the dataflow model no infinite data structures can be applied since they would lead to non-terminating subcomputations. However, dataflow seems to impose less overhead on program execution than graph reduction, since it is more related to the conventional call-by-value function call mechanism.

For historical reasons, the dataflow model is often connected with special purpose architectures (dataflow machines), while the compilation of functional languages for conventional parallel computers is usually based upon the graph reduction model. Since this model is very different from the execution mechanism available in hardware, functional language compilers produce code for an abstract graph reduction machine and this abstract machine code is then boiled down to the actual machine code. There are many compilation and optimization phases necessary, which makes compilers large and complicated.

As an alternative approach, a conventional imperative language may be chosen as the compilation target. The compiler for the functional language then becomes much simpler and the imperative target program can be executed on any architecture by applying the already available compilers for the imperative language.

Of course, there are several features of a non-strict functional language that cannot be directly compiled into an imperative form. Moreover, for each parallel implementation of an imperative language, there exist different parallel extensions that are not portable at all. A possible solution to these problems is to encapsulate all machine-dependent features into a small set of library functions that offer the necessary facilities for expressing the parallel behaviour of the functional program. This library has to be ported to each parallel architecture, while the functional language compiler and the generated imperative programs may remain unchanged.

The above idea has been already applied for implementing the parallel graph reduction model, e.g. [Goldberg, 1988]. The target language is usually LISP or C; the laziness properties of the functional language are expressed by procedures that create the representation of an unevaluated expression and that evaluate such an representation, respectively. However, demand-driven behaviour is difficult to model in an imperative control-driven language. Therefore the imperative target code is very low-level and much longer than the functional source.

On the other hand, there has almost no work been done on the application of the dataflow model for implementing functional languages on conventional parallel hardware. This is very deplorable since the model seems to impose principially less overhead and is closer related to conventional control-driven program execution. Its inability to handle infinite data structures and to prevent non-terminating subcomputations is in our opinion a more theoretical than practical deficiency. It just imposes a programming style that is different from that in lazy languages.

In this paper, we present our programming model $\mathcal{DF}$ (for dataflow) consisting of a few primitives that allow to express dataflow behavior in any imperative language. A binding of this model to the C language has been implemented on a multi-transputer system and several dataflow programs have been manually translated into this form. After the presentation of $\mathcal{DF}$, we describe its current implementation and the results we gained from experiments with the model. An overview on related work done by other researchers and an outlook on our further work concludes this report.

2 The Programming Model $\mathcal{DF}$

2.1 Required Features

As already sketched in the introduction, our intention is to develop an imperative parallel programming model that shows the following properties:

1. **Functionality:** It provides all features required to express the parallelism that the dataflow model can exploit from a functional program.
2. **Portability:** It can be added to any imperative programming language, is independent of the underlying parallel architecture and hides the particularities of varying parallel language dialects.
3. **Efficiency:** It can be implemented in an efficient way on conventional parallel architectures, i.e. it does not demand anything that is too difficult to realize.

In the following, we will demonstrate the demanded functionality by an example that shows the most important features of a parallel functional program. Consider the function `qsort` in Figure 1 that represents a generalized version of the Quicksort algorithm that sorts the first input list and appends the second one (and avoids by this generalization an additional append operation). Figure 2 shows a simplified illustration of the dataflow graph of the recursive branch of the function.

```
-- sort list
quicksort ::  [int] -> [int]
quicksort l = qsort l [ ]

-- sort h:t and append r
qsort ::  [int] -> [int] -> [int]
qsort [ ] r  =  r
qsort h:t r  =  qsort a h:(qsort b r)
                where (a,b) = split h t

-- partition h:t by e into two sublists
split ::  int -> [int] -> ([int],[int])
split e [ ]  =  ([ ], [ ])
split e h:t  =  (h:a,b), if h < e,
             =  (a,h:b), else
                where (a,b) = split e t
```

Figure 1: Quicksort

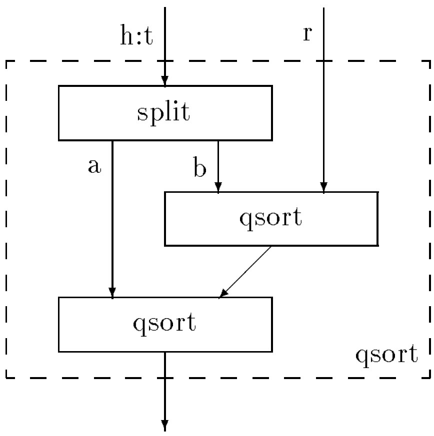

Figure 2: Dataflow Graph of `qsort` (Simplified)

`split` partitions the first input list t by its head h into two sublists a and b such that a contains all elements less than h and b all other ones. The first recursive invocation of `qsort` sorts b and appends the list r and passes its result to the second recursive `qsort` invocation that concatenates this list to the sorted version of a.

This function does not contain any horizontal parallelism, since the second `qsort` invocation depends on the result of the first one which itself depends on the result of the `split` operation. Hence, in a strict execution model the program would be executed in a totally sequential fashion.

However, a non-strict model offers lots of inherent vertical parallelism: since t`qsort` processes its first input list one element after the other and only in its final step requires the second input list, `split` can work in an overlapping fashion with both `qsort` invocations. While `split` produces the elements of a and b one after the other, both `qsort` branches can independently process the elements of a and b, respectively; synchronization between both branches takes only place when finally one `qsort` needs the result of the other one.

This program demonstrates very well various principles that an imperative programming model must obey to be a suitable target for a dataflow language:

Property 1 (Processes) Functions play the role of processes.

The unit of parallelization is in general the (user-defined) function. For instance, in the above `qsort` program, all three function activations might be executed on different processors (function-level parallelism). Of course, not each function will constitute a process of its own; for reasons of efficiency, functions below a certain grainsize should be executed in sequential within and by the current process.

Property 2 (Channels) Shared variables play the role of communication channels.

Since two function activations can be executed on different processors, a shared variable (i.e. a variable that is defined by one function and input to the other one) is the only mean of communication between these processes. For instance, the variable a in `qsort` connects the `split` process with one of the subprocesses. This communication is directed (since there is exactly one defining equation for each variable) and one-to-many (since the variable may be input argument of many functions).

Property 3 (Non-Strict Functions) Functions can start their execution independently of their arguments.

In the above program, both recursive `qsort` invocations can be executed in parallel only because the one invocation does not need the result of the other one for performing most of its work. Only in its final step, when the non-recursive branch of `qsort` needs its second argument for execution, both processes must be synchronized.

Property 4 (Non-Strict Data Structures) Data structures can contain undefined elements.

A pipelined form of parallelism between `split` and `qsort` is only possible, if `split` is not required to produce *first* its result list and *then* pass this list to `qsort`. Instead, `split` must be able to produce list cells with still undefined tails (links to the sucessor cells) such that `qsort` can start processing the list while simultaneously `split` extends the list by new cells. For proper cooperation between `split` (the producer) and `qsort` (the consumer), there must be some automatic form of synchronization that prevents a process to read undefined structure fields.

Property 5 (Garbage Collection) Data structures that are no more used shall be automatically reclaimed.

In a functional program, data structures are allocated as side-effects of certain primitive operators (e.g. the list builder *cons*) but never explicitly deallocated. An automatic garbage collection mechanism is in charge of reclaiming the space of structures that are not referenced any more. Especially in a parallel environment this can become quite a difficult task. Our programming model should therefore provide enough aid to make this process managable.

Reviewing this collection of demands, especially properties 3 and 4 seem to be of central importance. Moreover, since they are not at all fulfilled by imperative language constructs, they may be quite difficult to realize. The $\mathcal{DF}$ model shows that this is nevertheless possible.

2.2 The $\mathcal{DF}$ Kernel

The kernel of the $\mathcal{DF}$ model already fulfills most of the required properties listed in the previous subsection. It only omits a certain optimization (described in Subsection 2.4) and the support for garbage collection that will be discussed in Subsection 2.5.

The $\mathcal{DF}$ kernel consists of the following types, constants and procedures:

- **Types:**

  REF

  denotes a reference to a $\mathcal{DF}$-structure (structure in short). This type is word-sized and includes the special constant NIL (see below).

- **Constants:**

  PROCNUM

  is the number of processors in the system (each of which has a unique address in the range from 0 to PROCNUM-1.

  ACTPROC

  is the address of the current processor, i.e. the processor on which the current process is executed.

  NIL

  denotes a constant of type REF that is different from any reference to a structure.

- **Procedures:**

  par(↓fun, ↓proc, ↓n, ↓arg$_1$, ↓arg$_2$, ..., ↓arg$_n$)
  Start on processor *proc* a process executing *fun(arg$_1$, arg$_2$, ..., arg$_n$)*. *proc* must be either the address of some processor in the network or -1 (in that case some automatic strategy for process distribution is applied). *fun* must not return any value (i.e. it is actually a procedure), not reference any global environment (except the constants described above) and not cause any side-effects (besides the operations described below).

  new(↓size, ↑ref)
  Allocate a structure with *size* slots and return the reference *ref* to this structure. All slots are numbered from 0 to $size-1$ and may contain any word-sized value. All slots are initialized to the special constant $\perp$ (read "empty") that is different from any value in any type.

  get(↓ref, ↓index, ↑value)
  Read the slot at position *index* of the structure denoted by *ref* and return the word-sized *value* stored there. *ref* must be a structure reference different from NIL and *index* must be a valid slot position in this structure. If the slot contains $\perp$, the current process is blocked until another process writes some value into that slot (which is then returned as *value*).

  put(↓ref, ↓index, ↓value)
  Write into the slot at position *index* of the structure denoted by *ref* the given word-sized *value*. *ref* must be a structure reference different from NIL and *slot* must be the position of a slot in this structure that contains $\perp$. All processes that have been blocked on an attempt to read this slot are released.

The above definitions introduce two basic concepts:

- **Parallel Processes** created by the procedure par and
- **$\mathcal{DF}$-Structures**, an abstract data type that is generated and accessed by the procedures new, get and put. $\mathcal{DF}$-structures are nothing but indexed arrays that have the following additional properties:

  1. Each slot may contain the special constant $\perp$ ("the slot is empty").
  2. A process that tries to read an empty slot is blocked.
  3. A process may only write values into slots that are empty ("single assignment rule") and releases by this operation all processes that are blocked on this slot.

  $\mathcal{DF}$-structures are a global communication medium between parallel processes and the get operation simultanously performs communication and synchronisation.

In the following subsection, we will demonstrate the translation of a sample dataflow program into the $\mathcal{DF}$ kernel and compare the features of $\mathcal{DF}$ with the properties that we have demanded before.

2.3 A Sample $\mathcal{DF}$ Program

Figures 3 and 4 show a possible translation of the `quicksort` program presented in subsection 2.1 into the $\mathcal{DF}$ kernel bound to some imperative language dialect. This translation is guided by the attempt to construct the components of the program's dataflow graph as illustrated in Figure 2. This construction generally proceeds in the following steps:

1. **Allocate Channels:** First, channels have to be allocated to establish the communication between the parallel processes to be spawned (i.e. they denote the arcs of the dataflow graph). These channels are represented by structures with empty slots into which one process (the producer) will write the result of its computation while other processes (the consumers) will read their input values from them. In particular, if the transmitted value is a compound data structure, the slot contains a reference to this structure, i.e. in the case of lists, a reference to the first cell of the list.

2. **Spawn Processes:** Then the processes (that denote the nodes in the dataflow graph) are spawned. Each process executes a procedure with those structures as input arguments that represent input or output channels. The process reads the actual input values from these structures and writes its result value into an output structure. Synchronisation with other processes always occurs at **get** operations. If a desired value is not yet available, the current process is temporarily suspended.

3. **Local Operations:** Finally, several operations are performed by the current process. These operations can be considered as an optimization that makes the creation of an additional process superfluous by inlining its code into the body of the current procedure. By this optimization, the overhead of process creation can be avoided and excessive parallelism can be prevented.

The $\mathcal{DF}$ translation of the `qsort` function in Figure 3 shows these three phases very well. First the communication channels are allocated, then a `split` and two `qsort` processes are spawned and finally several local operations are performed that connect the input arguments of the procedure with the input arguments of the spawned processes. These statements read the head element of the input list, insert it at the front of the result of the second `qsort` process and write it the input structure for the `split` process (as well as the tail of the input list).

All input arguments to `quicksort`, `qsort` and `split` are references to structures that will (but might not yet) contain the input arguments or will receive the result values, respectively. They are *not* the input arguments themselves! This feature allows to model-non strict behaviour in an imperative language within a strict (i.e. call by value) argument passing mechanism. A process can start execution even if one of the input arguments has not yet been written into the input structure. On an attempt to read the empty structure slot (because the input argument is necessary for further computation), the process is automatically blocked until the required value is available.

Only because of this feature, parallelism between both `qsort` branches is possible. `qsort` reads the value stored in its second argument *rest* only in the the base case of the recursion,

```
------------------------------------
-- result := quicksort(list)
------------------------------------
proc quicksort(REF list, REF result)
  REF nil;
  new(2, ↑nil);                -- allocate cell
  qsort(list, nil, result);    -- result := qsort(list, nil)
  put(nil, 1, NIL);            -- write value into cell
end quicksort;

------------------------------------------
-- result := qsort(list, rest)
------------------------------------------
proc qsort(REF list, REF rest, REF result)

  REF head, tail, lista, listb;
  REF cons, resultb, tl;
  INT hd;

  -- get actual list reference
  get(list, 1, ↑list);

  -- base case
  if list = NIL then
    get(rest, 1, ↑rest); -- result := rest
    put(result, 1, rest);
    return;
  end;

  -- allocate channels
  new(2, ↑head); new(2, ↑tail);
  new(2, ↑lista); new(2, ↑listb);
  new(2, ↑cons); new(2, ↑resultb);

  -- start processes
  par(split, ACTPROC, 4, head, tail, lista, listb); -- (lista,listb):=split(head,tail)
  par(qsort, ACTPROC, 3, lista, cons, result);      -- result := qsort(lista, cons)
  par(qsort, next(), 3, listb, rest, resultb);      -- resultb := qsort(listb, rest)

  -- local computations
  get(list, 0, ↑hd); put(head, 1, hd);              -- head := list.hd
  get(list, 1, ↑tl); put(tail, 1, tl);              -- tail := list.tl
  put(cons, 1, resultb); put(resultb, 0, hd);       -- cons := hd:resultb

end qsort;
```

Figure 3: $\mathcal{DF}$ Quicksort Kernel

i.e. when the input *list* is NIL and the value of *rest* has to be written into the *result* structure. In the recursive branch, *rest* is just forwarded to one `qsort` process without demanding its contents. Hence, it makes sense to start two `qsort` processes to execute in parallel even if the result of the one (*resultb*) is input to the other one (in the form of the *cons* argument that represents the list *hd:resultb*).

However, in most cases a clever compiler may optimize this non-strict form of argument passing to the more efficient call-by-value mechanism. E.g. the *head* and *tail* structures that are passed to the `split` process are totally superfluous. They will receive the head and tail of the input *list* after all processes are spawned. An automatic strictness analyzer could tell us that that without these values neither `split` nor one of the `qsort` processes can start execution. Hence we might as well get these values before `split` is spawned and pass them directly to the process.

The $\mathcal{DF}$ translation of `split` shows some other form of optimization where the recursion is translated into an iterative while-loop. The function traverses its input *list* and appends its elements one after the other to the output list *lista* or *listb*, respectively. This function shows the typical example of consumer-producer parallelism. While it consumes the input list that is the result of some previous `split` process, it produces two results lists that will be fed into other `split` processes. All these processes may work in an overlapping fashion since get provides automatic synchronization betweeen them.

Another obvious optimization is to avoid a separate process for the `split` procedure by inlining it into the `qsort` program. In order to preserve parallelism, this should be done after both `qsort` processes are spawned that can then execute in parallel with the current one that performs the `split`.

Now we want to investigate how the first four properties demanded in Subsection 2.1 are actually fulfilled by the $\mathcal{DF}$ model:

1. **Processes:** The par statement creates a parallel process that executes a particular function, which is exactly the desired property.

2. **Channels:** Communication between a producer and a set of consumers is performed by a shared $\mathcal{DF}$-structure whose "anchor" is allocated in advance and passed to all participating processes.

3. **Non-Strict Functions:** Non-strictness of functions is achieved by the non-strictness of data-structures that are passed to the functions and contain the actual input values and receive the results, respectively.

4. **Non-Strict Data Structures:** The essence of $\mathcal{DF}$-structures as defined by the new, get and put operations is the fact that synchronization automatically occurs on empty structure slots.

The essential idea in the $\mathcal{DF}$ model is that all the central issues of communication, synchronization and non-strictness can be expressed by a single concept, namely non-strict structures. Its efficient implementation is therefore of crucial importance to the usefulness of $\mathcal{DF}$.

```
-------------------------------------------------------
-- (lista, listb) := split(element, list)
-------------------------------------------------------
proc split(REF element, REF list, REF lista, REF listb)

  REF cell;
  INT elem;

  -- get actual list and element
  get(list, 1, ↑list);
  get(element, 1, ↑elem);

  -- traverse input list
  while list <> NIL do
    get(list, 0, head);    -- head := list.hd
    if head < elem then
      new(2, ↑cell);       -- append head to lista
      put(lista, 1, cell);
      put(cell, 0, head);
      lista = cell;
    else
      new(2, ↑cell);       -- append head to listb
      put(listb, 1, cell);
      put(cell, 0, head);
      listb = cell;
    end;
    get(list, 1, ↑list);   -- list := list.tl
  end;

  -- close output lists
  put(lista, 1, NIL);
  put(listb, 1, NIL);

end split;
```

Figure 4: $\mathcal{DF}$ Quicksort Partitioning

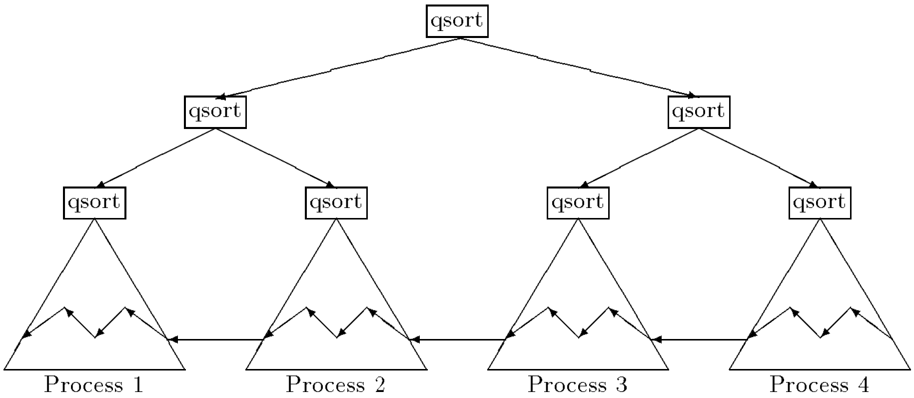

Figure 5: $\mathcal{DF}$ Coarse Grain Quicksort

2.4 Indirections

There is one problem with our current $\mathcal{DF}$ version of the `quicksort` program: it generates too many processes. On conventional parallel machines, process creation is an expensive task concerning time and in particular memory: each process needs a private workspace of considerable size that has to be allocated from the heap. For longer lists, our current `quicksort` version would try to spawn many thousands of processes which would very soon yield a heap overflow.

An obvious solution is to create parallel processes only up to a certain depth in the "process tree". Beyond this depth, the `qsort` calls are executed in a recursive and thus sequential manner. With a maximum depth of, say, 5, only $32 = 2^5$ processes are created that may be easily distributed among all processors. Figure 5 shows this approach.

But do these coarse-grain processes really execute in parallel? If we investigate their dynamic behavior in detail, we are faced with a bad surprise. Each `qsort` process performs a `split` and then goes down to the next level of recursion by the first `qsort` invocation. This is repeated until the the base case is reached for the first time (the process is than in the "rightmost bottom" edge of the recursion tree). In this situation, the process must receive the value of the `rest` argument to perform concatenation with the result list that it is going to compute. Unfortunately, this value is delivered by the next process only after that process has traversed its own recursion tree from the rightmost edge to the leftmost one. However, in the rightmost bottom edge of the tree, this process itself requires the result of its successor process and so on. In a nutshell, all processes must essentially execute in sequence, we have "optimized" our parallel program to death.

This situation is very annyoing, since the implicit concatenation of results essentially contributes to the efficiency of a large class of functional programs (e.g. backtracking algorithms). The problem did not arise in our original solution, since there each leaf of the recursion tree was a process of its own. Hence all solutions were computed in parallel, only the final concatenation was sequentially performed.

Our general problem can be described as follows: a process may require the result of another process only for structural reasons, i.e. because it is formally part of its own solution, but the process is actually not really interested in it. Its own contribution to the total result can be computed independently of that value, but as long as that value is not available, it is blocked and cannot start execution.

To break this vicious circle, we introduce the concept of *indirections* by an additional $\mathcal{DF}$ operation:

puti(↓ref, ↓index, ↓ivalue)

> This procedure behaves the same way as *put* but may be only applied on a reference value *ivalue* (called *indirection*). Any **get** operation on slot *index* will not return *ivalue* itself but the contents of the corresponding slot of the structure referenced by *ivalue* (this structure must have a slot with position *index*). To **puti**, the same restrictions concerning garbage collection apply as to the operation **putr** described in the following section.

Now we can rewrite the non-recursive branch of `qsort` as follows:

```
if list = NIL then
  puti(result, 1, rest);
  return;
end;
```

The *result* value does not receive the reference stored in *rest* as it has been computed by another process but a reference to *rest* itself. Thus the current process needs not perform a **get** and is not blocked any more. However, since **puti** has delivered *rest* as an indirection, this reference is transparent to any **get** operation and any user of the list will automatically receive the correct value.

Summarizing, the application of indirections allows to decouple processes such that the result of one may be used by the other without that the receiving process is blocked. However, indirections should be only used in the case described above, since they consume resources and slow down **get** operations. Actually, an indirection is nothing but a cheap replacement for an additional process that is spawned to connect two $\mathcal{DF}$ structures without blocking the father process.

2.5 Garbage Collection

The $\mathcal{DF}$ model supports a garbage collection method based on the *weighted reference counting* algorithm as defined in [Bevan, 1987]. This algorithm is quite simple to understand and perfectly suitable for a parallel system. Its basic idea is to associate a weight to each reference and to count in each structure the sum of the weights of all references to it. If this count drops to zero, the structure is not referenced any more and may be reclaimed. If a reference is copied, both copies get half of the original weight, which does not change the count in the referenced structure (this is the main advantage of this method in a distributed memory environment). If a reference is deleted, the count of the referenced structure is decreased by that weight.

Our version of this algorithm is a little bit different and bases on three additional operations that complete the $\mathcal{DF}$ model:

putr(↓ref, ↓index, ↓rvalue)

This operation behaves the same way as put but is used for writing a reference value into a structure slot. put itself is restricted to uninterpreted atomic values (integers, floating points, ...).

cop(↕ref)

Halve the weight of the reference *ref* (if this weight is a power of two) or set it to zero (if the weight is one). This operation is usually performed before the reference is applied (see the definitions below).

del(↕ref)

Delete the reference *ref*. *ref* must not be applied any more (see the definitions below).

We are now going to explain our version of the algorithm and the correct application of the cop and del operations. Several definitions will make this explanation somewhat more precise.

Definition 1 (Entry) A reference *enters* a process, if it is

1. passed as an initial argument to the process, or
2. the result of a new operation, or
3. returned by a get statement.

When a reference enters a process, a process becomes responsible for the correct management of its weight in order to keep the reference count of the referenced structure in a valid state.

Definition 2 (Real Weight) The *real weight* of a reference is defined as follows:

1. **Initialization:** When a reference enters a process, its real weight is either zero or some power of two. This (unknown) quanitity is also called its *initial weight*.
2. **Decrease:** If a reference is passed to a cop statement and its real weight is greater than one, this weight is halved. If its weight is one, it is set to zero. If it is already zero, it remains zero.

By the operation new, an empty structure is allocated and a reference to it is returned with an initial weight that is some power of two. By replicated applications of cop, this weight can be successively halved down to one; another application of cop sets its weigth to zero which is not changed any more. The concrete value of a reference weight is neither known nor important important to us, we only need to care of the properties defined below.

Definition 3 (Virtual Weight) The *virtual weight* of a reference is defined as follows:

1. **Initialization:** When a reference enters a process, its virtual weight is zero.
2. **Increase:** Each of the following operations increases the virtual weight of a reference by its actual real weight. These operations are said to *apply* the reference.
   (a) **Referencing:** The reference is passed as first argument to put, putr, puti or get.
   (b) **Storing:** The reference is passed as third argument to putr or puti.
   (c) **Deleting:** The reference is passed to del.
   (d) **Forwarding:** The reference is passed as a process argument to par.

The application of a reference by one of the listed operations sends a message to the referenced structure that decreases its reference count by the actual reference weight. In case of the par statement, the current process assumes that the new process will eventually apply the reference. The virtual weight of a reference is therefore the quantity by which the count of the referenced structure has been (or will eventually be) decreased. The relationship between a reference's virtual weight and the correctness of the reference counting method is now established by the following theorem:

Theorem 1 (Correctness) For the correctness of weighted reference counting, the virtual weight of each reference in each process must be less than or equal its initial weight.

Each operation that applies a reference increases its virtual weight by its actual real weight. On an application of a reference, the correctness of reference counting can therefore be maintained by one of the following actions only:

1. After the application, the reference is applied no more.
2. Before the application, a cop operation is performed.

In the first case, the virtual weight of the reference becomes at once its initial weight. From the referenced structure's point of view, this is equivalent to say that the reference has been deleted (the reference count is decreased by the weight of the reference).

In the second case, the virtual weight of the reference is only increased by half of its real weight. The reference count of the structure is decreased by this quantity, but there is still some weight left to perform further applications.

Why don't we always issue a cop operation before a reference is applied and by this maintain correctness? This question is answered by the following theorem:

Theorem 2 (Usefulness) If a reference is erased whose virtual weight is less than its initial weight, the referenced structure will never be reclaimed.

The final application of a reference must not be preceded by a cop to signal that the reference is not applied any more and the referenced structure can be safely reclaimed (if there are no other references to it). The application sequence of a reference *ref* must therefore always look like

> *ref enters process;*
> cop(*ref*);
> *apply ref;*
> cop(*ref*);
> *apply ref;*
> ...
> cop(*ref*);
> *apply ref;*
> *apply ref;*

Obviously, the scheme is always fulfilled by a sequence of cop and application pairs trailed by a del statement that finally invalidates the reference. But the general scheme has the advantage that the explicit use of del can be avoided in most cases: we can usually determine the final application of a reference beyond which the reference will not be applied any more and thus just omit the cop before this application. Hence, on a distributed memory system, no explicit deletion message to the referenced structure is necessary.

Finally, let us consider the case where a sequence of cop operations leads to a reference weight of one that can not be halved any more. The above description says that a cop on a reference with weight one will decrease its weight to zero. Hence, we have lost one weight unity and the reference's virtual weight can never again become its initial weight; the referenced structure will thus not be reclaimed. However, if the initial weight is sufficiently large, this will be very rarely the case (moreover, for such structures the probability is very high that they will be alive during the whole runtime of the program). For purists that do not want any unreferenced "zombie" structures in the heap, also other solutions exist, e.g. the application of indirection cells as suggested in [Bevan, 1987].

2.6 The $\mathcal{DF}$ Model

We conclude this section with a comprehensive summary of all types, constants and procedures that the $\mathcal{DF}$ model consists of:

Types:

REF ... a reference to a $\mathcal{DF}$-structure.

Constants:

NIL ... the empty reference.

PROCNUM ... the number of processors.

ACTPROC ... the address of the current processor.

Procedures:

par($\downarrow$*fun*, $\downarrow$*proc*, $\downarrow n$, $\downarrow arg_1$, ...$\downarrow arg_n$): start a new process.

new($\downarrow$*size*, $\uparrow$*ref*): allocate a structure of specified size.

get($\downarrow$*ref*, $\downarrow$*index*, $\uparrow$*value*): read a value from a structure slot.

put($\downarrow$*ref*, $\downarrow$*index*, $\downarrow$*value*): write an integer into a structure slot.

puti($\downarrow$*ref*, $\downarrow$*index*, $\downarrow$*value*): write an indirection into a structure slot.

putr($\downarrow$*ref*, $\downarrow$*index*, $\downarrow$*value*): write a reference into a structure slot.

cop($\updownarrow$*ref*): duplicate a reference.

del($\updownarrow$*del*): delete a reference.

3 Implementation of C-$\mathcal{DF}$

The previous section demonstrated how the programming model $\mathcal{DF}$ fulfills the required features of functionality and portability. In this section, we want to show that the model can be also efficiently implemented on conventional parallel hardware. We present our implementation of C-$\mathcal{DF}$, a binding of $\mathcal{DF}$ to the C programming language, on a parallel system consisting of 17 T800 transputers [Inmos, 1988b].

The transputer is for our purposes a very interesting architecture because it is the only currently available processor that provides special support for parallelism: it allows extremely fast process creation and context switches and has four links each of which allows communication with another processor simultaneously with the execution of the program.

For our implementation, we used the Logical Systems Transputer Toolset that contains a compiler for a parallel version of C with the posssibility of inlining transputer assembly code [Logical Systems, 1990]. The implementation consists of some 1500 lines of C code and several inline assembly instructions. The $\mathcal{DF}$-library that has to be linked to the user program is about 15 KB large.

3.1 Process Creation

The transputer has two special registers that determine the process that is currently under execution: these registers contain the address of the next instruction to be executed and the workspace pointer (i.e. the address of the top of the program stack), respectively. Besides the process that is actually executed, there exists a queue of processes that are temporarily descheduled and wait for execution.

If a process is descheduled, the address of its next instruction is written into its workspace. Since the transputer has no conventional register set (but only a small hardware stack), no more registers have to be saved. The process is now fully defined by the address of its workspace that is usually stored at the end of the process queue. Then the process at the front of the queue is scheduled for execution by loading the workspace and the instruction

length	refcnt	avail/ref bits	slot 1	...	slot n

Figure 6: $\mathcal{DF}$-Structure

register with the corresponding values. On transputers, a process switch is hence very fast (about 1 μs).

For creating a parallel process with the `par` routine, the following tasks have to be performed:

1. **Allocation:** A new and sufficiently large process workspace has to be allocated on the heap. The default workspace size is determined by a user-definable variable `WORKSPACE`.
2. **Initialization:** The workspace has to be initialized with the address of the function to be executed, the arguments to the function and the address of the instruction to be executed when the process terminates.
3. **Start:** The process is then started by queuing its workspace into the list of descheduled processes.

Process creation is therefore quite simple and efficient, but there exists a minor problem concerning the termination of processes. Before a process may terminate, it has to deallocate its workspace that would be lost otherwise. However, if the process is finally descheduled, its instruction address is automatically stored within the workspace. Since the workspace has at this time already been deallocated, care has to be taken that that the alteration of the affected memory cell does not confuse the heap management. Moreover, no other process must allocate this space before the current process has terminated.

Since the heap management was implemented from scratch, we were able to exclude the first case. Furthermore, since the transputer can deschedule processes only at certain instructions, also the second case could be avoided.

3.2 $\mathcal{DF}$-Structures

A $\mathcal{DF}$-structure is constructed as depicted in Figure 6 (on transputers, the standard wordlength is 32 bits):

- **Structure Management:** The first two halfwords contain the sructure's length and its reference count, respectively.
- **Slot Management:** The next word contains pairs of available/reference bits: bit i of this word is set, if slot $2 * i$ of the structure contains a value, bit $2 * i + 1$ signals whether this value is a reference to another structure or not.

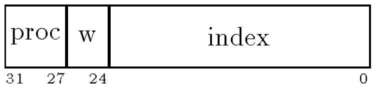

Figure 7: $\mathcal{DF}$-References

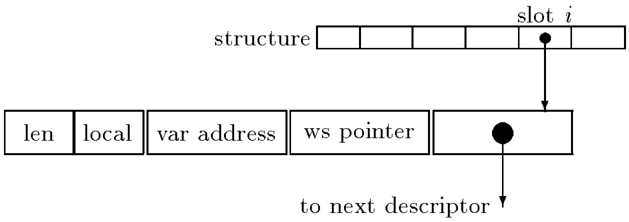

Figure 8: A Process Descriptor

- **Slots:** The following words denote the structure slots. Since there is only one word reserved for slot managment, a structure may contain up to 16 slots.

Structure references are constructed as depicted in Figure 7:

- **Processor Number:** The most significant 5 bits denote the number of the processor that the referenced structure is stored on.
- **Reference Weight:** The next 3 bits contain the weight of the reference represented as its binary logarithm plus one (a value of zero denotes zero weight).
- **Structure Index:** The least significant 24 bits denote the position of the referenced structure relative to the start of the heap (in units of words).

Using these storage representations, the operations on $\mathcal{DF}$ structures are implemented as follows:

- new allocates a new structure, stores the total length of the structure in the length field and sets the reference count field to the maximum weight. The available and reference bits are reset and the slot fields are initialized with the NULL pointer, which indicates that no processes are waiting on the slots. The weight field of the returned reference is set to its maximum.
- get checks the available bit of the denoted slot. If the slot is not empty, its value is written into the result variable (if the value is a reference, then its weight has to be halved before). If the available bit is not set, the slot points to to a list of descriptors of blocked processes. In this case, a new descriptor is allocated as depicted in Figure 8 and linked into the list. The adress of the result variable is

stored into the descriptor as well as the address of the current process workspace. Then the process is descheduled which leaves the address of the next instruction in the workspace.

- put (respectively putr) checks the available bit of the referenced slot such that the single assignment rule cannot be violated (which would trigger a fatal error). Then the routine traverses the list of process descriptors linked into the slot. For each process, the desired value is stored into the result variable, the process workspace is queued into the scheduling queue and the descriptor is deallocated. Finally, the value is written into the slot and its available bit is set (as well as the reference bit in the case of putr).

Having fulfilled their task, get, put and putr decrease the reference count of the affected structure by the weight of the given reference (this is also the only purpose of the del procedure). If the count decreases to zero, the structure is reclaimed. The references stored in the reclaimed structure are deleted by decreasing the counts of the referenced structures which might now be recursively reclaimed, too.

The cop procedure decreases the weight field of the given reference by one (if it is not yet zero). Since the weight field actually contains the binary logarithm of the weight, this operation effectively halves the weight of the reference (or decreases it from 1 to 0).

Up to now we have described the $\mathcal{DF}$ model under the assumption that all operations can be performed locally, i.e. without involving other processors. The following subsections will extend this description to the multi-processor case.

3.3 The Communication System

The transputer provides communication with other processors by four links that each allow transmission of messages in both directions at a rate of 10 MBit/s. The software counterpart of these physical links are channels that allow uni-directional, unbuffered and point-to-point communication between processes on remote processors. Each hardware link represents two software channels, one into each direction.

The transputer allows communication between direct neighbours only, but the $\mathcal{DF}$ model demands a global structure space distributed among all physical processors. Hence, we had to implement a routing system that allows communication between any pair of processors. This system consists of four input processes and four output processes, one for each physical link (see Figure 9).

- Each **input process** waits for a block of messages that is stored into a local buffer. Each message of this block is managed according to its destination: messages that have reached the wanted processor are stored into a queue where a server process will find and manage them (see the next subsection). For any other message, a routing table is looked up to decide through which link it must be forwarded. The message is then stored into the corresponding link buffer from where it will be transmitted by an output process.

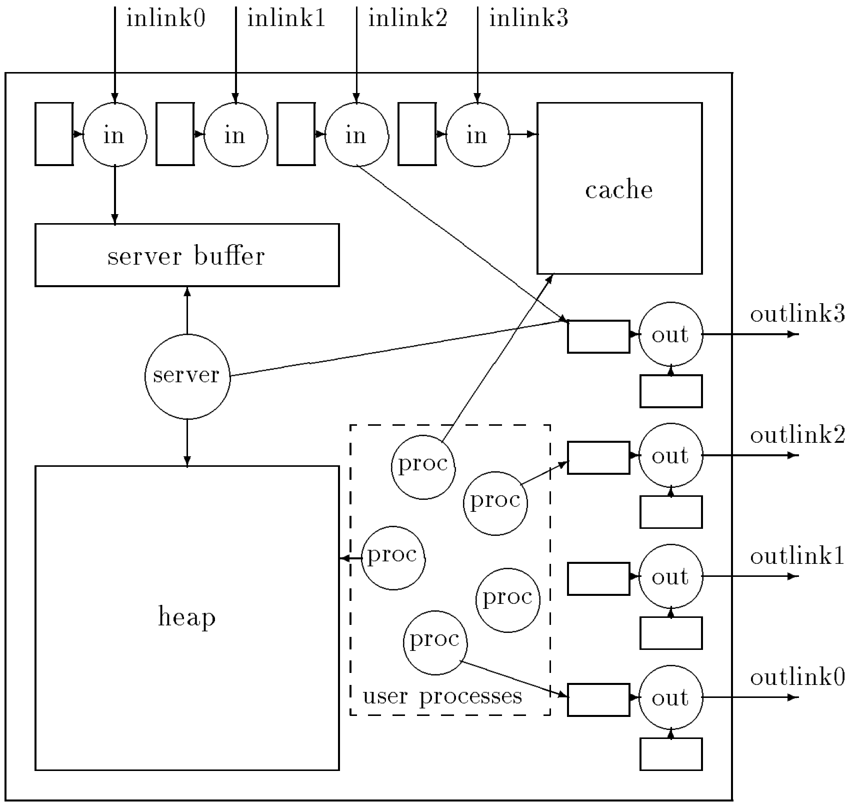

Figure 9: The Communication System

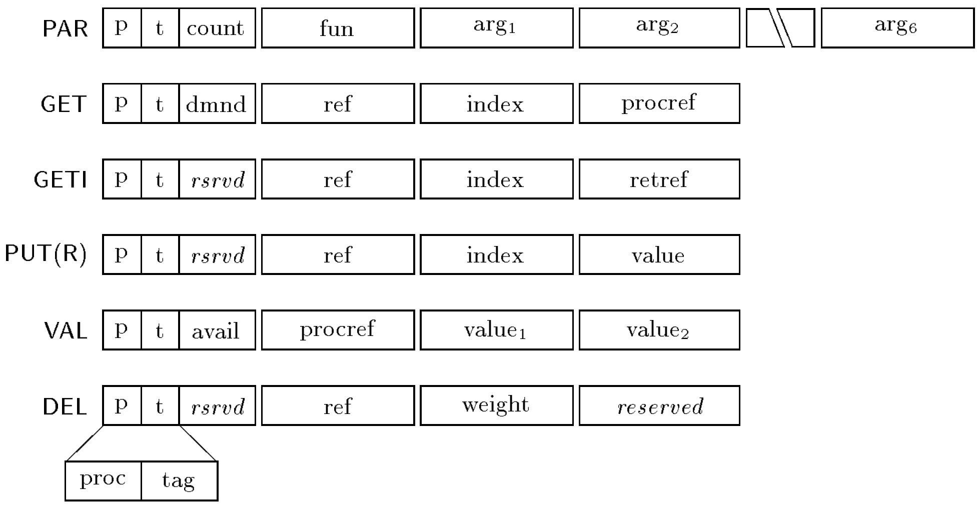

Figure 10: $\mathcal{DF}$ Messages

- Each **output process** is in charge of transmitting messages through a particular link. It moves all messages stored in the corresponding link buffer into a local send buffer and signals on the link that it is ready for communication. As soon as the input process on the other side has become ready too, the block of messages is transmitted and the output process can continue execution.

The communication scheme is free from deadlock as long as the link buffers do not overflow. Our experiences show that for sufficiently large buffers (about 32 messages) this case does never occur. The only topology-dependent part of our scheme is the routing table that can be be easily adapted to new interconnection networks.

3.4 The Server Process

On each processor, a server process manages all messages that have been sent from other processors. The input processes store these messages into a buffer from where the server can receive them. The messages denote either requests from remote user processes or answers to requests that have been issued by local user processes. Basically, there are six types of messages that are all of equal size (four words), with the exception of the eight word PAR message (see Figure 10).

- PAR: If a parallel process is to be started on another processor, a PAR message is sent that contains the address of the function to be executed, the number of its input arguments (up to six) and their values. The server on that processor will receive this message and start the denoted process as described in Subsection 3.1.

- GET: If a process requires a value that is stored in some non-local structure, this process has to be temporarily stopped. For this purpose, a process descriptor is allocated as described in Subsection 3.2. The process sends a GET message to the processor where the structure is sited on and is suspended. Besides the structure reference and the index of the desired slot, the message contains a reference to the descriptor of the requesting process. If the desired value is availble, the remote server will return it using a VAL message. If the value is not yet availabe, the request will be linked into the structure slot similar to the descriptors of blocked local processes.

- GETI: If a puti instruction is executed, the indirection is actually *not* written into the denoted structure slot. Instead a GETI message is sent to the structure referenced by the indirection and the user process continues execution. The message does as a return address not contain a reference to a process descriptor but a reference to the original structure. If the value denoted by the indirection is available, the server returns it in form of a PUT (respectively PUTR) message to the original structure from where it may be directly read by other processes. If the value is not available, the request is linked into the slot of the indirection structure.

- PUT/PUTR: If a value is to be written into the slot of a structure on another processor, a PUT (respectively PUTR) message is sent that contains the structure reference, the slot index and the value to be delivered. The server on the denoted processor will write this value into the referenced slot, release the waiting local processes and send VAL messages to the processes that wait on remote processors.

- VAL: A VAL message delivers a value that was was once requested from another processor. This reply contains a reference to the descriptor of the process that requested this value and was then stopped. The server uses these quantities to fill the value into the result variable and to restart the process then. Besides a primary value, a VAL message may also transmit a secondary value that it used in connection with a caching mechanism (see Subsection 11).

- DEL: A DEL message is sent, if a del operation is applied to a non-local reference. It causes the server of a remote processor to decrease the reference count of the affected structure and to reclaim its space, if the count drops to zero.

A problem in our transputer implementation of the $\mathcal{DF}$ model is the necessity to provide the system processes (i.e. the server and the link processes) with enough computing time. If there are more than some dozens user processes that compete with the system processes for the processor, the overall system performance can be extremely degraded. One solution would be to provide with system processes with a higher priority than ordinary user processes such that they are more often scheduled for execution.

However, the transputer's possibilities for priorizing processes are rather poor. Besides the queue of ordinary low-priority processes, there exists a second queue of high-priority processes that are always executed prior to the other ones, i.e. low-priority processes only come to execution if the high-priority queue is empty. The only possibility to let low-priority processes execute is to disable all high-priority processes for some fixed amount of time; if a high-priority process is enabled again, it immediately interrupts any executing low-priority process. This process scheme is basically intended for real-time applications but is not a suitable tool for a scalable form of process priorization.

We have therefore chosen another solution. We artificially decrease the average time that user processes are scheduled for execution by inserting additional "rescheduling points" at certain $\mathcal{DF}$ instructions. Each time that one of these points is reached the user process is rescheduled and allows the next process to be executed. Since process switching is very fast, this does not cost much processing time but allows system processes to be executed more often than user processes. The number of rescheduling points is controlled by the global variable TIMESLICE that can be set according to the expected number of user processes.

3.5 Process Distribution

The par statement for process creation explicitly demands the number of the processor that the new process shall be executed on. This reflects our belief that the compiler (perhaps even the programmer) should be given a chance to explicitly map processes to processors, since they have more information about the program than the runtime system and can therefore make better decisions in many cases.

However, a processor number of -1 in a `par` signals that some automatic form of process distribution is desired. In this case, we apply a simple form of *diffusion scheduling* that usually provides good results (see [Goldberg, 1988]): If a new process shall be automatically mapped to some processor, the current processor looks for its minimal loaded neighbour. If the load of the processor itself is greater than this neighbour's load plus some treshold value, the process is forwarded to that processor, otherwise it is executed locally.

As a measure for the load of a processor, we use the number of active (i.e. created and not yet terminated) processes. Each time a block of messages is transmitted through a link, the corresponding output process communicates this number to the neighbour processor, too. Thus all processors have some idea about the current load of their neighbours compared to their own one.

The treshold represents some measure for the preference of user processes to remain on the local processor. The smaller its value is, the faster processes will spread through the network (and the more costly communication will become); the larger it is, the more processes will be executed locally (and the less parallelism will be exploited). Since the performance of a program may be very sensitive to this threshold, the user may control its value by the global variable LOCPREF.

The distribution of processes among processors can be made visible by the global variable STATISTICS. If this variable is not set to 0, each processor sends certain statistics information to the root processor where it can be communicated to the user. For instance, each processor may signal when it starts execution of the first user process and when it becomes idle again.

3.6 Structure Caches

One of the main problems with distributed memory computers is the unbalanced ratio between the computing power of individual processors and the communication bandwith of the interconnection network. This is to a smaller extent also true for transputer systems and is often the reason for serious bottlenecks.

One main concern in the design of a communication protocol for our $\mathcal{DF}$ model has therefore been to minimize the number of messages to be exchanged between processors. In fact, this goal has already been considered in the design phase of the $\mathcal{DF}$ model itself. For instance, the creation of a process on a remote processor requires only one message: process arguments can be transmitted together with the function to be executed and the father process is neither responsible for allocating the remote workspace nor for deallocating it after the son process has been terminated. On the contrary, the process creation protocol described in [Arvind and Nikhil, 1988] demands $5 + n$ messages where n is the number of arguments.

However, the main load of the interconnection network is caused by the access to values stored in structures on remote processors. Let us consider the most common case, namely that of a list cell that has been allocated on one processor and that is read by a process on another processor. As well for the head as for the tail field of this cell, a GET message has to be sent and, for each request, one VAL message has to be returned. If another process on the same processor requires the contents of the list cell too, these actions have to be repeated.

In order to decrease network traffic, we introduced in our transputer implementation of the $\mathcal{DF}$ model the concept of a *cache*. Our idea relys on the observation that if one field of a remote structure is required by some process, it is very likely that also the other fields will be required at a future moment. For example, when a list is processed, usually the head and the tail field of each list cell are both read. Thus, why not transmit the whole cell on the first demand of one of its fields and store it in a cache on the requesting processor? If then the other field is demanded, no more network message is necessary. Moreover, if other processes on that processor demand this list cell too, they can immediately access the cache without involving any further messages.

In our implementation, this idea has been generalized in the following way (compare Figure 11):

- On each processor, a separate region of memory is used as a cache. This cache consists of four-word entrys, where the first word contains the reference to the structure of which a section is cached, the second word essentially contains the start index of this section, a pending flag and the reference and available bits of the cached structure slots. The remaining two words then represent the cached section (i.e. two slots) of the structure itself.

- Before a process sends a GET message and is then stopped, it uses the given structure reference and the slot index as a hash key to determine whether an entry for the desired structure slot is already available in the cache. If no suitable entry can be found, a new one is reserved and initialized with the reference and the index of the

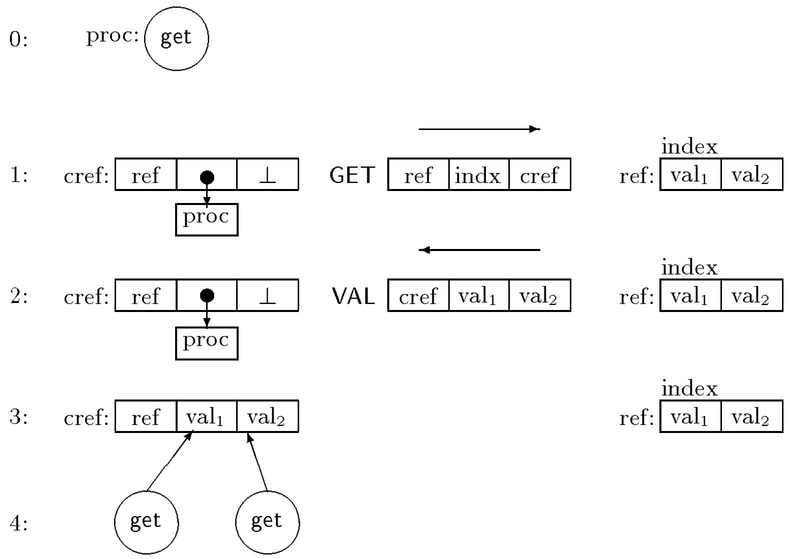

Figure 11: The $\mathcal{DF}$ Caching Mechanism

slot section that matches the given slot. The slots are initialized to empty and the pending flag is set. A workspace descriptor is allocated and linked into the slot that will receive the demanded value. Then the GET message is sent containing a reference to the cache entry and the process is stopped.

- If a process finds the value demanded by a **get** operation already in the cache, it just copies it from there and continues execution. However, it may be the case that the matching cache entry is indeed available but the required slot is still empty (e.g. if the GET message has not yet been answered). If the pending flag signals, that a message for this structure section is already on the way, a process descriptor is linked into the desired slot and the process is stopped. If no message is on the way, then the process sets the pending flag and sends a GET message before it stops itself.

- If the request for a structure value is fulfilled, not only this primary value but also the secondary value, i.e. the content of the neighbour slot, is returned (if this slot is not empty). The delivered values are stored in the cache entry and all processes blocked on these slots are released. However, if only one value could be delivered and there are processes waiting on the other slot, a new GET message has to be sent.

The reason why cache entries have only two slots is twofold. First, the data structures most used in functional languages are lists built from cells with two slots. Second, we don't want to give up our idea of fixed sized messages and in our four word VAL message exactly two values can be transmitted. So the most common application of data structures is handled very nice in our cache scheme.

However, a cache reduces the number of messages only in the case that both values of the cached section can be transmitted in one message and that both values are really

$\mathcal{DF}$-Operation	μs	C-Operation	μs	Factor
`new(s, r)`	55.9	`r = malloc(4*s)`	29.1	1.9
`put(r, i, v)`	21.6	`r[i] = v`	1.1	19.0
`get(r, i, v)`	25.0	`v = r[i]`	1.0	25.7
`par(f, 0, 6,` a_1`, ...,` a_6`)`	42.0	`f(`a_1`, ...,` a_6`)`	3.3	12.7

Figure 12: Speed of C-$\mathcal{DF}$ Operations

demanded. Moreover, for correct garbage collection unfortunately one additional message is required. If a new cache entry is reserved, in general another entry must be replaced. If the replaced entry contains a structure reference (as the tail field of a list cell), this reference must be deleted which requires a DEL message to the referenced structure.

Therefore the cache is actually not of such importance for structures that are read only once but for those that are used by many processes. In fact, our idea arised from experiments with the `primes` program described in the next section. This program works on a global list that is distributed among the network and read by all processes. Here the cache was able to significantly reduce the amount of network traffic.

3.7 The C-$\mathcal{DF}$ Extensions

We conclude this section with a summary of the global variables by which the programmer can influence the behaviour of a C-$\mathcal{DF}$ program:

WORKSPACE ...size of process workspace in words.

TIMESLICE ...additional time slices for system processes.

LOCPREF ...local preference in automatic scheduling.

STATISTICS ...boolean flag for statistics output.

4 Experiments

4.1 Absolute Performance

When new parallel systems or languages are introduced, the presentation of practical results is in most cases restricted to the relative speedups that can be achieved. However, we think that one should also speak frankly about the absolute performance of a system since this is an important piece of information for users and researchers. Hence, we have measured the average time of the $\mathcal{DF}$ kernel operations in our transputer implementation of C-$\mathcal{DF}$and compared the results to the time required for performing corresponding operations of the C language itself. The figures we gained from this comparison are just the outcome of a particular benchmark test and may therefore vary in real programs. However, they give a rough feeling about the range in that the performance of our implementation is settled. Several remarks will help to interpret the results that are presented in Figure 12:

- In the benchmark program, we allocated in a loop several thousand cells from the heap. In this case, the **new** operation takes about 2 times longer than the C heap allocation routine `malloc`. However, in a real application, where heap cells are allocated and deallocated in an unpredictable order, the heap becomes very soon fragmented. Then our heap management (that is based on the buddy method) will perform significantly better than the C heap management (that is based on a boundary tag method).
- **get** and **put** take roughly 20 times longer than read and write operations on C arrays (assuming that **get** does not block the current process and that **put** does not release any blocked processes). A significant part of the difference is based on the fact that for portability reasons, all $\mathcal{DF}$ operations are implemented in C and each call of **put** or **get** involves at least one function call. A direct implementation of these procedures in assembly language would speed up these routines with an factor of 2–3.
- The creation of a parallel process (on the local processor) is quite fast compared to simple function calls. The main overhead of the **par** statement is caused by the fact, that the workspace for the new process cannot be taken from the program stack must be allocated from the heap (see the time for the **new** operation).

The critical time that determines the performance of a $\mathcal{DF}$ program is that of the most frequently executed **get** operation. For instance, we compared a C-program for the generation of prime numbers with a parallel $\mathcal{DF}$ program where each number was tested for primality by a parallel process (see the next subsection) but all processes were executed on the same processor. The $\mathcal{DF}$ program was roughly as much slower as the factor between `get` and the C assignment statement says.

Give these results raise to satisfaction or to disappointment? This basically depends on the original expectations. Programs written in $\mathcal{DF}$ are one to two orders of magnitude faster than programs that arise from other parallel implementations of functional languages (see Section 5) but they are also one order of magnitude slower than sequential C-programs. Is the glass half full or half empty?

We think that the results are quite satisfying. Due to lack of human ressources, we concentrated our efforts during the implementation of the $\mathcal{DF}$ language mainly on portability (almost the whole code is written in C). For a really efficient implementation, several selected parts would have to be rewritten in assembly language which would make $\mathcal{DF}$ really competitve also from the performance point of view. Moreover, many applications are not only time-bound but actually memory-bound. Since our implementation is very memory-effective, it gives us the possibility to experiment with programs that solve problems of considerable complexity.

4.2 The Test Programs

In this subsection, we give a short overview on the dataflow programs that were used to measure the relative speedups that can be achieved with our C-$\mathcal{DF}$ implementation. These sample programs all fulfill the following criteria:

- The problems that are solved by these programs are of a certain complexity and interest. This excludes algorithms that are simply to parallelize but only solve problems that are as exicting as computing the n-th factorial of a given number.
- The applied algorithms are most efficient also on sequential machines. This excludes algorithms that are simply to parallelize but much slower than other ones (e.g. that version of the quicksort algorithm that explicitly concatenates the subresults such that both recursive invocations can be independently executed).

The algorithms are given in a functional notation as described in e.g. [Bird and Wadler, 1988]. Additionally, the behavior of each algorithm is illustrated by two dataflow graphs: one show the static data dependencies within the function and is actually just a pictoral representation of the main branch of this function. In the other graph, the recursive invocations of the main function are expanded such that the process topology becomes visible as it emerges during execution on a parallel machine.

4.2.1 Prime Number Generation

The first program solves the problem of computing all primes up to some bound n. The main part of our solution is a function `sieve` that receives the list *plist* of all primes less than some value t and returns the list of all primes from t to n. In a naive approach, `sieve` would apply *plist* to test t and, if t is prime, call itself recursively with a new list *plist0* that is a copy of *plist* with the new prime *test* being appended. However, due to the required list duplications, this solution would be very inefficient.

In our version, `sieve` forwards to its recursive invocation the original argument *plist* and returns as its result the list delivered by the recursive call with t inserted at the front. But how do the recursive `sieve` invocations ever get access to the new prime t? The main point is the top-level function `primes` that calls `sieve` with its own result *plist* as input argument. Hence, each prime that is returned as result of a `sieve` call becomes available to all of the following recursive invocations. This application of a *circular data structure* (i.e. a data structure that is defined in terms of itself [Allison, 1989]) does only work in a non-strict execution model where on a call of a function its arguments may not be fully evaluated.

The following functions describe this solution in detail:

```
-- all odd primes up to n
primes ::  int -> [int]
primes n  =  [ ], if n < 3
          =  plist, else
             where plist = 3:(sieve n 5 plist)
```

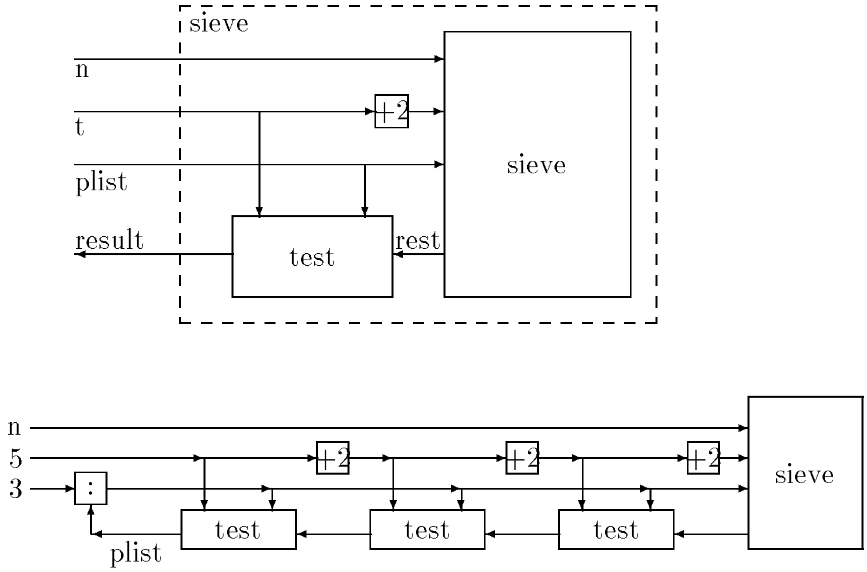

Figure 13: Prime Number Generator

```
-- primes from t to n, plist has all primes less than t
sieve ::  int -> int -> [int] -> [int]
sieve n t plist  =  [ ], if t > n
                 =  test t plist rest, else
                    where rest = sieve n (t+2) plist

-- append t to rest, if t is prime
test ::  int -> [int] -> [int] -> [int]
test t p:plist rest  =  t:rest, if t < p*p
                     =  rest, if t mod p == 0
                     =  test t plist rest, else
```

The above algorithm also gives raise to a lot of parallelism: since the recursive `sieve` call does not directly depend on the `test` result, `sieve` can go down the recursion and spawn a set of `test` processes each of which is in charge of testing one number for primality. Since `qsort` initially stores the prime 3 in *plist*, all of these processes can start execution. The first `test` processes will soon feed more primes into the list and the more primes have been computed, the more processes can decide about the primality of their test numbers.

This program is especially interesting since it is an example for vertical parallelism in its pure form. There is no possibility for the independent evaluation of function arguments at all, all parallelism comes from unfolding the `sieve` recursion. In some sense, the dataflow graph of this program represents a bidirectional pipeline connected by two streams of data: in one direction, the `qsort` process sends primes into the pipeline of `test` processes; in the other direction, the `test` processes themselves send new primes to the `root`. However, this flowing of data is only a mental picture that helps to illustrate the interaction between processes.

4.2.2 The N-Queens Problem

The 8-queens problem is to a find all possible positions of 8 queens on a chess board such that no queen can beat another. This example is one of a class of search space problems that can be solved by the backtracking technique: we generalize the problem to variable input sizes (hence the n-queens problem) and solve it for all values from 0 to n. In stage i of the solution process, we may assume that we have found all valid positions of $i-1$ queens (on rows 1 to $i-1$) on the board. We then extend these partial solutions in all possible ways, i.e. with queens in row i and columns 1 to n. This procedure continues until we have found all solutions of the original problem. If a partial solution is inconsistent (i.e. two queens can beat each other), we can discard it since it can no more lead to any valid solution.

Our n-queens program essentially consists of two mutually recursive functions `solve` and `test`. `solve` takes the current board position *row* and *column* and a partial solution *part* that contains $row-1$ queens. The function returns all solutions starting with *part* and extended by the queens at positions *column* to n (i.e. it traverses the search tree at level *row*). For this purpose, `solve` applies the function `test` that returns all solutions starting with *part* and extended by the queen at the current *column* position (i.e. `test` goes one level down in the search tree). `test` checks whether this extended partial solution is valid and, if this is the case, itself calls `solve` to extend this it to the final solutions.

In order to avoid explicit concatenation of lists of results that emerge from different partial solutions, `solve` and `test` are given an additional parameter *rest*. The result list that is returned by a `solve` operation is directly fed via this parameter into the next `test` operation and vice versa. Instead of returning the empty list in the base case of recursion, these functions return the `rest` argument as their result and hence perform automatic concatenation of results.

The following definitions show the program in detail:

```
-- list of queen columns
board = [int]

-- solve n-queens problem and return list of boards
queens ::  int -> [board]
queens n = solve n 1 1 [ ] [ ]

-- traverse search tree horizontally using partial solution
solve ::  int -> int -> int -> board -> [board]
solve n col row part rest  =  rest, if col > n
                           =  test n col row part sols, else
                              where sols = solve n (col+1) row part rest
```

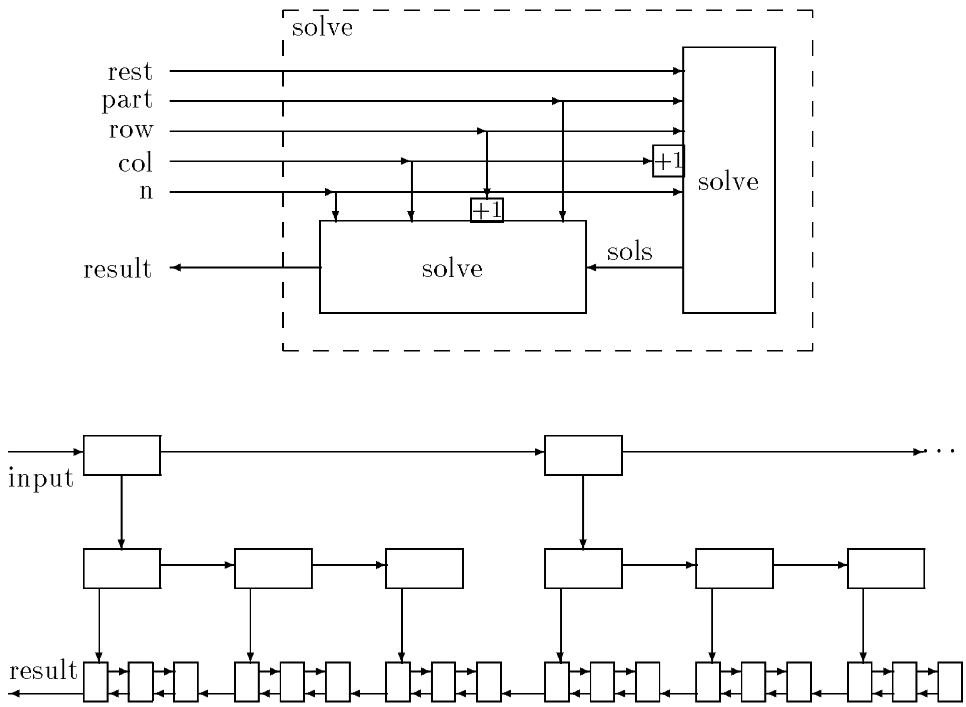

Figure 14: The N-Queens Problem

```
-- traverse search tree vertically using partial solution
test ::  int -> int -> int -> board -> [board]
test n col row part rest  =  rest, if conflict col 1 part
                          =  (col:part):rest, if row == n
                          =  solve n 1 (row+1) (col:part) rest, else

-- test whether queen conflicts with board
conflict ::  int -> int -> board -> bool
conflict  _    _     []           =  false
          col  dist  (col0:rest)  =  true, if beats col col0 dist
                                  =  conflict col (dist+1) rest, else

-- test whether two queens at given row distance can beat each other
beats ::  int -> int -> int -> bool
beats col col0 dist = col == col0 || col == col0+dist || col == col0-dist
```

On a parallel machine, different subtrees of solutions may be searched in parallel: the `solve` function can immediately go down recursion and create a set of `test` processes each of which starts with a different partial solution. Each `test` checks the current solution for consistency. If the newly added queen does not beat any of the other ones, `test` calls `solve` to find all solutions starting with the new partial solution. Hence, a tree of processes emerges where the leaves represent the actual search space that is traversed (see Figure

14). All solutions are concatenated by the leaves and yield the final result list.

As in the previous example, the `queens` program does not provide any horizontal parallelism: the simplified dataflow graph of `solve` in Figure 14 (with the `test` function expanded) shows that both recursive branches are related by the *sols* argument. The possibility to avoid explicit list concatenation and still maintain parallelism, is a major reason for the importance of non-strictness in parallel functional programming.

4.2.3 Quicksort

Finally, we again present the functional version of the Quicksort algorithm as already described in Section 2. As in our solution to the n-queens problem, an explicit concatenation of lists can be avoided by an additional *rest* argument that is returned in the base-case of recursion.

```
-- return sorted version of input list
quicksort ::  [int] -> [int]
quicksort list = qsort list [ ]

-- sort first list and append second one
qsort ::  [int] -> [int] -> [int]
qsort  [ ]        rest  =  rest
       head:tail  rest  =  qsort lista head:(qusort listb rest)
                           where (lista,listb) = split head tail

-- partition list by element in two sublists
split ::  int -> [int] -> ([int],[int])
split  _     [ ]        =  ([ ],[ ])
       elem  head:tail  =  (head:lista,listb), if head < elem
                        =  (lista,head:listb), else
                           where (lista,listb) = split elem tail
```

On a parallel machine, this program gives raise to a tree-like topology of parallel processes where the leaf processes perform concatenation of sublists (see Figure 15).

4.3 Benchmark Results

We measured the performance of our transputer implementation with three C-$\mathcal{DF}$ programs (listed in the Appendix) that were manually derived from the algorithms presented in the previous subsection. For the benchmarks, the transputer system was configured to hypercube topologies of dimensions 0 (1 processor) to 4 (16 processors) where the additional root processor was linked into one edge of the hypercube. Figure 16 shows absolute times of these experiments and the relative speedups compared to the time that each program required on the root transputer. Since this processor runs at a higher clock frequency (25 Mhz) than the hypercube processors (17 Mhz), all timings were adjusted to a common basis. The benchmark provided the following results:

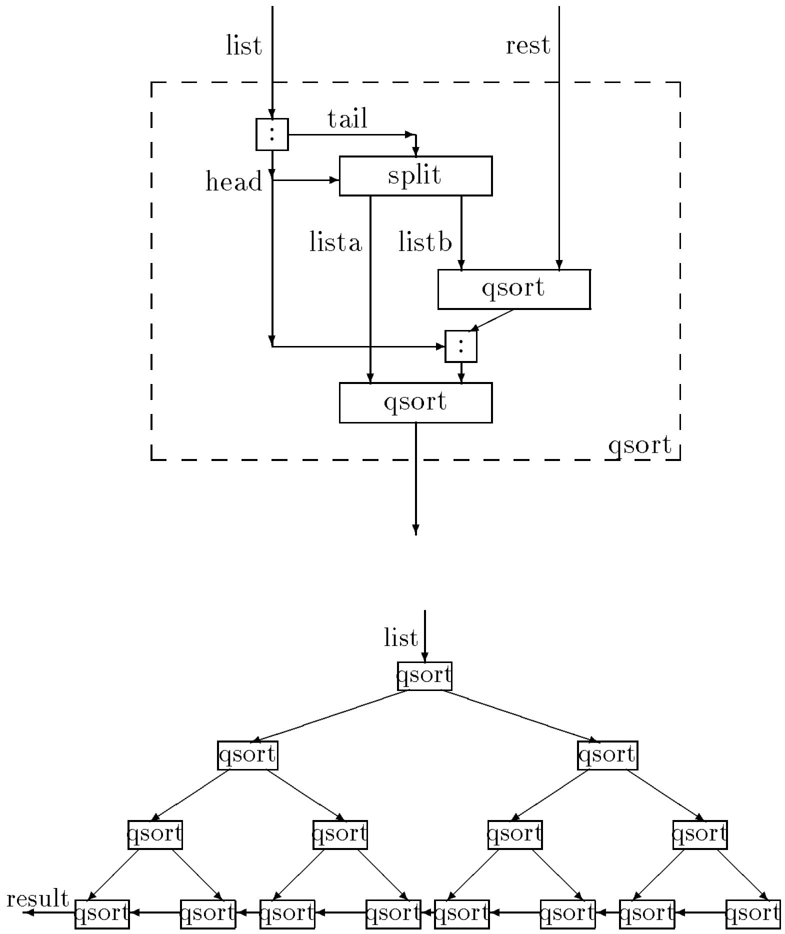

Figure 15: Quicksort

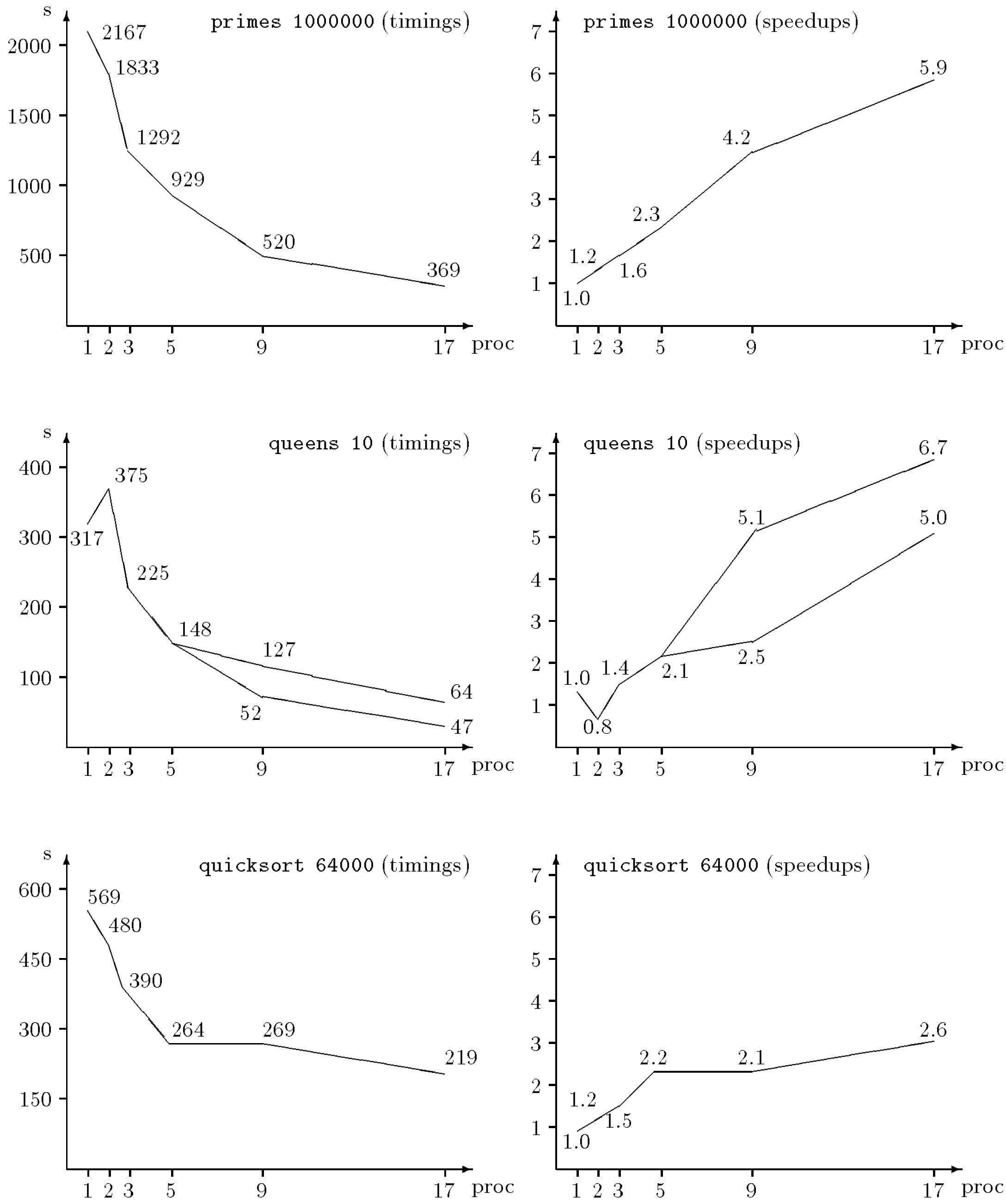

Figure 16: Benchmark Results

- `primes` was applied to compute all primes up to one million. For each test number, a process was spawned and distributed by the application of the automatic diffusion scheduling strategy. The treshold LOCPREF for the length of the process queue was set to 4, the number TIMESLICE of additional system time slices was set to 4, too. The performance of this program scaled very well with the number of processors and lead to a maximum speedup of 5.9. The program profited very much from our implementation of a caching mechanism, since the same global list of primes is accessed many times by all processes on all processors.

- `queens` was applied to solve the n-queens problem on a board of dimension 10. In the two processor configuration, performance was slightly degraded. However, from three processors on, significant speedups could be achieved. In the $\mathcal{DF}$ program, generally only the first level of the search tree was parallelized (i.e. 10 processes were spawned); for the application of all 17 processors, the first two levels (100 processes) were distributed among the system. Initially, the automatic distribution strategy (with LOCPREF set to zero) performed as well as an explicit placement of processes (where each new process was mapped to the next processor). Beginning with 9 processors, the automatic scheduling strategy performed significantly worse (speedup less than 5.0) than the manual process placement (that achieved a speedup of 6.7).

- `quicksort` was applied to a list of 64000 random numbers. We parallelized the recursion tree up to the smallest level such that all processors were saturated with processes by the automatic distribution strategy (with LOCPREF set to 0). In general, the speedups that we could achieve were rather poor (up to 2.6). The main reason for this fact is the large amount of communication between processors: all processes have to send requests for each element of their input lists to the neighbour processors where these lists are stored on. Since the target hardware consisted of a machine with distributed memory, the ratio between computation and communication was too worse to achieve better speedups.

On the one hand, the results of these experiments are rather promising (we achieved a maximum speedup of 6.7 with 17 processors), on the other hand, they also show the limits of our implementation: The $\mathcal{DF}$ model is essentially built upon a shared structure space, while the transputer is based on a distributed memory model. For parallel algorithms where the amount of inter-process communication is low (or can be artificially decreased by caching, as in the `primes` program), this difference is not so important. However, if processes are very tightly coupled (as in the `quicksort` program), communication between processors becomes a significant bottleneck. Obviously, an efficient parallelization on distributed memory architectures is only possible, if parts of an algorithm can be extracted that may execute in parallel without exchanging too large amounts of data.

5 Related Work

The $\mathcal{DF}$ programming model owes much to concepts developed by other groups working on the parallel implementation of functional languages or related topics. The following subsections give a short overview on these ideas and their relationship to our work.

5.1 I-Structures

The most important influence to the $\mathcal{DF}$ model comes from the work done by the dataflow group at the MIT. In fact, our concept of $\mathcal{DF}$-structures is a generalized version of their I-structures [Arvind *et al.*, 1987] that has been redesigned for the practical application in an imperative language. While we concentrate on the implementation of functional languages on conventional multiprocessors, the main concern of this group is the construction of parallel architectures based on a new execution paradigm (dataflow machines). They developed the concept of I-structures to extend the pure dataflow scheme with the possibility of handling compound data structures. Besides the dataflow graph in which the flow of tokens drives program execution, there exist special I-structure units that receive requests tokens from the graph and return tokens as result.

The main contribution of the $\mathcal{DF}$ model is the realization that non-strict data structures alone are sufficient for the simulation of dataflow behavior: On the one hand, this concept so powerful that it is able to cover all matters of communication and synchronization between parallel processes and the whole issue of non-strictness. On the other hand, it can be added as an abstract datatype to any sequential language and can be efficiently implemented also on conventional parallel hardware. Hence, the $\mathcal{DF}$ model can basically rely on the simple and efficient control-driven form of program execution.

Of course, the $\mathcal{DF}$ model has been designed to simulate dataflow behavior not on the level of individual instructions but on that of whole functions (macro-dataflow). However, we think that this fully sufficient and, moreover, that it is the only form that can be efficiently implemented. From the outside, there exists almost no difference between the behavior of pure dataflow programs and of control-driven programs that simulate this behavior.

5.2 ADAM & EVE

The author's diploma thesis [Schreiner, 1990] dealt with the design and implementation of the abstract dataflow machine ADAM and the functional programming language EVE for programming this machine. A high-level EVE program is compiled into code for this abstract machine and executed by a parallel ADAM interpreter on a multi-transputer system. Despite of its name, ADAM actually represents a hybrid architecture that extends a conventional control-driven processor by several features (among them I-structures) to support dataflow behavior.

The goal of the project was to investigate the behavior of dataflow programs on control-driven parallel machines. During our work, we learned much about the paradigms of control-flow and data-flow, their differences and their relationship. As a consequence, the idea arose to extend a conventional programming language by a small set of operations that are necessary to support dataflow-like programming. This language extension should then be a suitable target of a parallelizing compiler for a functional language. While the ADAM runtime system was large, not portable and slow, this small dataflow library should be small, efficient and easily portable by using the compilers available on today's parallel machines. The $\mathcal{DF}$ model is the practical outcome of this idea.

5.3 LINDA

LINDA [Carriero and Gelernter, 1988] is one of the few parallel programming models that has received wider acceptance on different architectures. It is machine-independent, portable and can be added to any sequential language. These features sound very similar to our demands for the $\mathcal{DF}$ model. Could LINDA therefore be a suitable model for the parallel implementation of a functional language? Unfortunately not. Besides the matter of efficiency (efficient implementations of LINDA for distributed memory machines have not yet come into existence [Leichter and Whiteside, 1989]), we have excluded LINDA for the following reason:

The LINDA tuple space operations are not based on addresses but on the contents of tuples. This makes LINDA a convenient tool for a human programmer who can explicitly read tuples of a particular pattern. However, this form of data access is not appropriate for the linked lists of functional languages where each cell has an explicit reference to its successor. Of course, additional tuple fields can play the role of addresses, but searching for a tuple that matches a particular address field is far less efficient than random memory access. Moreover, since tuples have to be explicitly extracted from a tuple space, no automatic garbage collection is provided in LINDA (nor can it be, since there do not exist any explicit tuple references).

Still there is an intersting relationship between LINDA and $\mathcal{DF}$, since both models base synchronization between processes on the availability of data: A process in LINDA is blocked, if it tries to receive a tuple that is not yet in the tuple space, while a $\mathcal{DF}$-process is suspended on the attempt to read an empty structure slot. In this sense, LINDA tuples play the role of slots in $\mathcal{DF}$-structures while a $\mathcal{DF}$-structure itself corresponds to a whole tuple space! However, there can be at most one value in a $\mathcal{DF}$ structure slot (single assignment rule), while there may be several tuples matching a specific input pattern (which introduces non-determinism in LINDA).

5.4 CSP and Transputers

Hoare's model of communicating sequential processes [Hoare, 1985] has laid the foundation for several parallel languages and at least one parallel architecture: the modern message passing concept (like in OCCAM [Inmos, 1988a]) and the communication system of the transputer [Inmos, 1988b] are both build on top of this model. We would therefore like to clarify the relationship between CSP and $\mathcal{DF}$, too.

Like LINDA and $\mathcal{DF}$ (and probably like any good model for parallel programming), the synchronization of processes is in CSP based on the availability of data: if one process tries to read a value from a communication channel into which no value has yet been written by a sender process, the receiver is automatically blocked (and vice versa). We have $\mathcal{DF}$-structure slots several times compared with communication channels. Are $\mathcal{DF}$-structures then nothing but arrays of CSP channels? The answer is no.

There are two main differences between CSP channels and $\mathcal{DF}$ structure slots: First, every process is blocked on an attempt to write a value into a channel where no process is waiting yet (in contrast to the non-blocking `put` operation in $\mathcal{DF}$). Second, a channel is a communication medium between exactly two processes through which a sequence of values may be transmitted. On the contrary, not more than one value must be written into a $\mathcal{DF}$ structure slot, but many processes may read this value. In short, a CSP (respectively transputer) channel is an unbuffered one-to-one communication medium, while a $\mathcal{DF}$ structure slot is a one-to-many communication medium with a buffer of size one that is never emptied.

From this definition we can draw the conclusion that a hardware implementation of $\mathcal{DF}$ structures using current (transputer) technology is not very difficult. The get and put operations of the $\mathcal{DF}$ model can be as efficiently implemented as the transputer's in and out instructions for message passing [Inmos, 1988b]. Since the next generation of transputers [Inmos, 1991] also provides point-to-point communication between any two processors in a network, a global space of $\mathcal{DF}$-structures seems to be easily feasible, too.

5.5 ALFL and ALFALA

ALFALA is an implementation of the functional language ALFL on an iPSC hypercube multi-processor [Goldberg, 1988]. We mention this system as one example for a number of projects that deal with the parallel implementation of lazy languages by programmed graph reduction.

The ALFL compiler decomposes a functional program into a set of so-called serial combinators that basically represent sequential tasks to be distributed among the processors. These combinators are parts of the program graph augmented by constructs for the creation and synchronization of tasks and for initiating the evaluation of delayed expressions. Serial combinators are translated into pieces of C code and combined to a program that manipulates the graph which is distributed among all processors of the hypercube system.

The most distinguishing fact between our implementation of C-$\mathcal{DF}$ and the ALFALA system is that we build our system on the dataflow concept (that is eager) while ALFALA is based on graph reduction (that implements lazy evaluation). Since the dataflow principle can apply a call-by-value function call mechanism, the compilation of a functional program into $\mathcal{DF}$ code is much simpler. It basically keeps the structure of the original program and increases the code size of only one order of maginitude (see the Appendix). On the other hand, the code generated for ALFALA is no more related to the functional source and at least two orders of magnitude larger.

This difference also is important when comparing the efficicency of both systems: From the figures in [Goldberg, 1988] we conclude that C-$\mathcal{DF}$ is roughly 10–100 times faster and far more memory efficient than ALFALA (the input sizes in that paper are too small to compare them with the problems given in this article). However, we have to stress that their programs have been generated by a compiler while ours are still manually coded (see the concluding remarks).

6 Conclusion

The definition of $\mathcal{DF}$ is only a first step in the parallel implementation of a functional language. In this paper, we have shown that $\mathcal{DF}$ is a suitable model to implement the parallelism expressed in functional programs. Starting from functional specifications of parallel algorithms, we are able to generate $\mathcal{DF}$ programs that can be efficiently executed on a transputer system (we will soon port the current version of C-$\mathcal{DF}$ to a shared memory machine, too). However, it still remains to prove that this tranlation can be performed automatically by a parallelizing compiler.

Our future work will concentrate on the development of such a compiler. Different from other approaches that use simple local criteria (e.g. the strictness of function arguments) and thus create many fine-grained tasks, we want to look at the program as a whole and try to detect the parallel algorithm that the programmer has expressed in it (e.g. the parallel creation of a data structure). Using this information, the generated program shall exploit the parallel algorithm but only spawn few coarse-grained processes.

By our experiments with the manual transformation of functional specifications into parallel $\mathcal{DF}$ programs, we have learned much about the main sources of parallelism in functional languages and how they can be utilized. We believe that most algorithms are based on a few simple parallelization principles and that a compiler can detect these principles in concrete programs. The effective application of functional languages for parallel programming should then become feasible.

A C-$\mathcal{DF}$ Programs

The following pages present the C-$\mathcal{DF}$ programs that were used to achieve the benchmark figures of Section 4. We only omitted those parts of the programs that dealt with activating the program and printing the results (these parts are contained in the function `real_main` that is spawned as the initial user process on the root processor).

```
/***************************************************************************
 *
 * primes.c                                                Wolfgang Schreiner
 * dataflow prime number generator                         (c) RISC-Linz 1991
 *
 ***************************************************************************/

#include <df.h>        /* dataflow support */

#define next() AUTO    /* process distribution */

INT WORKSPACE  = 128; /* process workspace in words */
INT TIMESLICE  = 5;   /* additional server timeslices */
INT LOCPREF    = 5;   /* local preference in AUTO scheduling */
INT STATISTICS = 0;   /* print statistics */

/***************************************************************************
 * Link with function real_main() to start program and print results.
 ***************************************************************************/

/***************************************************************************
```

```
 * primes :: int -> [int]
 * primes n = [], if n < 3
 *          = plist, else
 *            where plist = 3:(sieve n 5 plist)
 **************************************************************************/
VOID primes(INT n, REF result) {

  REF plist;

  if (n < 3) {                          /* result.tl := NIL */
    put(result, 1, NIL);
    return;
  }

  new(2, &plist);

  cop(&plist);                          /* result.tl := plist */
  putr(result, 1, plist);

  cop(&plist);                          /* plist.hd := 3 */
  put(plist, 0, 3);

  cop(&plist);                          /* plist.tl := sieve(n, 5, plist)
  par(sieve, 0, 4, n, 5, result, plist);

}

/**************************************************************************
 * sieve :: int -> int -> [int] -> [int]
 * sieve n t plist = [ ], if t > n
 *                 = test t plist rest, else
 *                 = where rest = sieve n (t+2) plist
 **************************************************************************/
VOID sieve(INT n, INT t, REF plist, REF result) {

  REF rest;

  if (t > n) {                      /* result.tl := NIL */
    put(result, 1, NIL);
    del(&plist);
    return;
  }

  new(2, &rest);

  cop(&rest); cop(&plist);          /* rest.tl := sieve(n, t+2, plist) */
  par(sieve, next(), 4, n, t+2, plist, rest);

  test(t, plist, rest, result);     /* result.tl := test(t, plist, rest) */

}
```

```
/*************************************************************************
 * test :: int -> [int] -> [int] -> [int]
 * test t p:plist rest = t:rest, if t < p*p
 *                     = rest,   if t mod p == 0
 *                     = test t plist rest, else
 *************************************************************************/
VOID test(INT t, REF plist, REF rest, REF result) {

  INT p;
  REF cell;

  while (1) {

    get(plist, 1, &plist);          /* plist := plist.tl */
    cop(&plist);                    /* p := plist.hd */
    get(plist, 0, &p);

    if (p*p > t) {                  /* cell.hd := t */
      new(2, &cell);                /* result.tl := cell */
      cop(&cell);                   /* cell.tl := rest.tl */
      put(cell, 0, t);
      cop(&cell);
      putr(result, 1, cell);
      puti(cell, 1, rest);
      return;
    }

    if (num % prime != 0) {         /* result.tl := rest.tl */
      puti(result, 1, rest);
      return;
    }

  }

}

/*************************************************************************
 * end of primes.c
 *************************************************************************/
```

```
/*************************************************************************
 *
 * queens.c                                             Wolfgang Schreiner
 * dataflow solution to n queens problem               (c) RISC-Linz 1991
 *
 *************************************************************************/

#include <df.h>       /* dataflow support */

#define next() (AUTO) /* process distribution */
#define MAXROW 1      /* maximum depth for process creation */

INT WORKSPACE  = 128; /* process workspace in words */
INT TIMESLICE  = 0;   /* additional server timeslices */
INT LOCPREF    = 0;   /* local preference in AUTO scheduling */
INT STATISTICS = 0;   /* print statistics */

/*************************************************************************
 * queens :: int -> [board]
 * queens n = solve n 1 1 [] []
 *************************************************************************/
VOID queens(INT n, REF result) {

  REF nil;

  new(2, &nil);      /* nil.tl := NIL */
  cop(&nil);
  put(nil, 1, NIL);

  solve(n, 1, 1, nil, nil, result);

}

/*************************************************************************
 * solve : int -> int -> board -> [board] -> [board]
 * solve n col row part rest = rest, if col > n
 *                           = test n col row part sols, else
 *                             where sols = solve n (col+1) row part rest
 *************************************************************************/
VOID solve(INT n, INT col, INT row, REF part, REF rest, REF result) {

  REF sols;
  REF cons0, cons1;

  while(1) {

    if (col > n) {                /* result.tl := rest.tl */
      puti(result, 1, rest);
      del(&part);
      return;
    }
```

```
    new(2, &sols);
    cop(&sols); cop(&part);

    if (row > MAXROW)              /* execute recursive branch locally */
      solve(n, col+1, row, part, rest, sols);
    else                           /* start process */
      par(solve, next(), 6, n, col+1, row, part, rest, sols);

   /**********************************************************************
    * test(n, col, row, part, sols)
    *
    * test n col row part rest = rest, if conflict col 1 part
    *                          = (col:part):rest, if row == n
    *                          = solve n 1 (row+1) (col:part) rest, otherwise
    **********************************************************************/

    cop(&part);                      /* result.tl := sols.tl */
    if (conflict(col, 1, part)) {
      puti(result, 1, sols);
      del(&part);
      return;
    }

    if (row == n) {                  /* result.tl := sols */
      cop(&sols);                    /* sols.hd := cons0 */
      putr(result, 1, sols);         /* cons0.hd := col */
      new(2, &cons0);                /* cons0.tl := part.tl */
      cop(&cons0);
      putr(sols, 0, cons0);
      cop(&cons0);
      put(cons0, 0, col);
      puti(cons0, 1, part);
      return;
    }

    new(2, &cons0);                  /* cons0.hd := col */
    cop(&cons0);                     /* cons0.tl := part.tl */
    put(cons0, 0, col);
    puti(cons0, 1, part);
    new(2, &cons1);                  /* cons1.hd := cons0 */
    cop(&cons1);
    putr(cons1, 1, cons0);

    col = 1;                        /* execute recursive branch locally */
    row = row+1;
    part = cons1;
    rest = sols;

  }

}
```

```
/***************************************************************************
 * conflict :: int -> int -> board -> bool
 * conflict _   _    []          = false
 * conflict col dist (col0:rest) = true, if beats col col0 dist
 *                               = conflict col (dist+1) rest, otherwise
 **************************************************************************/
BOOL conflict(INT col, INT dist, REF part) {

  INT col0;

  get(part, 1, &part);           /* part := part.tl */
  while (part != NIL) {

    cop(&part);                  /* col0 := part.hd */
    get(part, 0, &col0);

   /************************************************************************
    * beats :: int -> int -> int -> bool
    * beats col col0 dist = col==col0 || col==col0+dist || col==col0-dist
    ***********************************************************************/
    if ((col == col0) || (col == col0+dist) || (col == col0-dist)) {
      del(&part);
      return(TRUE);
    }

    dist = dist+1;               /* part := part.tl */
    get(part, 1, &part);

  }

  return(FALSE);

}

/***************************************************************************
 * end of queens.c
 **************************************************************************/
```

```
/*************************************************************************
 *
 * qsort.c                                              Wolfgang Schreiner
 * dataflow quicksort                                   (c) RISC-Linz 1991
 *
 *************************************************************************/

#include <df.h>         /* dataflow support */

#define next()    AUTO  /* process distribution */
#define MAXLEVEL 5      /* level of process creation */

INT WORKSPACE  = 128;  /* process workspace in words */
INT TIMESLICE  = 1;    /* additional server timeslices */
INT LOCPREF    = 0;    /* local preference in AUTO scheduling */
INT STATISTICS = 0;    /* print statistics */

/*************************************************************************
 * Link with function real_main() to start program and print results.
 *************************************************************************/

/*************************************************************************
 * quicksort(list, result)
 *
 * quicksort :: [int] -> [int]
 * quicksort list = qusort list []
 *************************************************************************/
VOID quicksort(REF list, REF result) {

  REF nil;

  new(2, &nil);
  cop(&nil);
  put(nil, 1, NIL);
  qsort(list, nil, result, 0);

}

/*************************************************************************
 * qsort :: [int] -> [int] -> [int]
 * qsort []        rest = rest
 *       head:tail rest = qsort alist head:(qsort blist rest)
 *                        where (alist,blist) = split head tail
 *************************************************************************/
VOID qusort(REF list, REF rest, REF result, INT level) {

  INT head;
  REF lista, listb, resultb, cons;

  get(list, 1, &list);                /* list := list.tl */
  while (1) {
```

```
    if (list == NIL) {                 /* result.tl := rest.tl */
      puti(result, 1, rest);
      return;
    }

    cop(&list);                        /* head := list.hd */
    get(list, 0, &head);               /* list := list.tl */
    get(list, 1, &list);

    new(2, &lista);
    new(2, &listb);
    new(2, &resultb);
    new(2, &cons);

    cop(&cons); cop(&resultb);         /* cons.tl := resultb */
    putr(cons, 1, resultb);            /* resultb.hd := head */
    cop(&resultb);
    put(resultb, 0, head);

    if (level < MAXLEVEL) {            /* process for recursive branch */
      cop(&listb); cop(&lista);
      split(head, list, lista, listb);
      par(qusort, next(), 4, listb, rest, resultb, level+1);
    } else {
      cop(&lista); cop(&listb);        /* execute recursive branch locally */
      split(head, list, lista, listb);
      qusort(listb, rest, resultb, level+1);
    }

    list = lista;                      /* execute other recursive branch */
    rest = cons;                       /* always by current process */
    level = level+1;

  }

}

/****************************************************************************
 * split :: int -> [int] -> ([int],[int])
 * split elem []  = ([],[])
 *       elem head:tail = (head:alist,blist), if head < elem
 *                      = (alist,head:blist), else
 *                        where (alist,blist) = split elem tail
 ***************************************************************************/
VOID split(INT elem, REF list, REF lista, REF listb) {

  INT head;
  REF cell;

  while (list != NIL) {

    cop(&list);                   /* head := list.hd */
```

```
    get(list, 0, &head);

    if (head < elem) {           /* lista.tl := cell */
      new(2, &cell);             /* cell.hd := head */
      cop(&cell);                /* lista := cell */
      putr(lista, 1, cell);
      cop(&cell);
      put(cell, 0, head);
      lista = cell;
    } else {                     /* listb.tl := cell */
      new(2, &cell);             /* cell.hd := head */
      cop(&cell);                /* listb := cell */
      putr(listb, 1, cell);
      cop(&cell);
      put(cell, 0, head);
      listb = cell;
    }

    get(list, 1, &list);         /* list := list.tl */

  }

  put(lista, 1, NIL);            /* lista.tl := NIL */
  put(listb, 1, NIL);            /* listb.tl := NIL */

}

/****************************************************************************
 * end of qsort.c
 ****************************************************************************/
```

References

[Allison, 1989]
Lloyd Allison.
Circular Programs and Self-referential Structures.
Software — Practice and Experience, 19(2):99–109, 1989.

[Arvind and Nikhil, 1988]
Arvind and R. S. Nikhil.
Executing a Program on the MIT Tagged-Token Dataflow Architecture.
Computation Structures Group Memo 271, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1988.
Also in: PARLE 87, Conference on Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, June 15–19, 1987, Volume 259 of Lecture Notes in Computer Science, Springer, Berlin.

[Arvind *et al.*, 1987]
Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali.
I-Structures: Data Structures for Parallel Computing.
Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987.
Also in: Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico, September 28 – October 1, 1986.

[Bevan, 1987]
D.I. Bevan.
Distributed Garbage Collection Using Reference Counting.
In *Proceedings of PARLE, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, pages 176–187, Eindhoven, The Netherlands, June 15–19, 1987.
Volume 259 of Lecture Notes in Computer Science, Springer, Berlin.

[Bird and Wadler, 1988]
Richard Bird and Philip Wadler.
Introduction to Functional Programming.
Series in Computer Science. Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[Carriero and Gelernter, 1988]
Nicholas Carriero and David Gelernter.
Linda in Context.
Technical Report YALEU/DCS/RR-622, Department of Computer Science, Yale University, New Haven, Connecticut, 1988.

[Goldberg, 1988]
Benjamin F. Goldberg.
Multiprocessor Execution of Functional Programs.
Research Report YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988.

[Hoare, 1985]
C.A.R. Hoare.
Communicating Sequential Processes.
Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.

[Inmos, 1988a]
Inmos.
Occam 2 Reference Manual.
Prentice Hall, New York, 1988.

[Inmos, 1988b]
Inmos Limited, Bristol, UK.
The Transputer Databook, November 1988.

[Inmos, 1991]
Inmos Limited, Bristol, UK.
The T9000 Transputer Products Overview Manual, 1991.

[Leichter and Whiteside, 1989]
Jerrold S. Leichter and Robert A. Whiteside.
Implementing Linda for Distributed and Parallel Processing.
Technical Report YALEU/DCS/TR-715, Department of Computer Science, Yale University, New Haven, Connecticut, 1989.

[Logical Systems, 1990]
Logical Systems, Corvallis, Oregon.
Transputer Toolset Version 89.1, January 1990.

[Loogen, 1990]
Rita Loogen.
Parallele Implementierung funktionaler Programmiersprachen (Parallel Implementation of Functional Programming Languages) (in German), volume 232 of *Informatik-Fachberichte.*
Springer, Berlin, 1990.

[Peyton Jones, 1987]
Simon L. Peyton Jones.
The Implementation of Functional Programming Languages.
Prentice-Hall, New York, 1987.

[Schreiner, 1990]
Wolfgang Schreiner.
ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language.
Master's thesis, Johannes Kepler University, Linz, Austria, October 1990.
Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 91-1, Austrian Center for Parallel Computation, January 1991.