A Portable Parallel Library for the Compilation of Dataflow Languages

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Tel: $*(7236)3231-66$, Fax: $*(7236)3338-30$
E-Mail: schreine@risc.uni-linz.ac.at

April 1992

Abstract

We present a portable library $\mathcal{DF}$ for the compilation of non-strict but non-lazy functional languages (dataflow languages) to imperative programs on conventional parallel computers. The $\mathcal{DF}$ library contains several functions that can be added to any sequential language and allow to express dataflow behavior in a simple and efficient way. The purpose of this library is to provide a high-level and machine-independent target language to which dataflow languages can be compiled for parallel execution. Implementations of $\mathcal{DF}$ on a distributed memory transputer system and on a shared memory computer show promising results.

Keywords: parallel programming models; functional programming; dataflow.

1 Introduction

The programming of parallel computers in functional languages gains more and more interest. A functional program is a set of (mutually recursive) function definitions where the order of function applications is not specified. 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 g and h in $f(g(e_1), h(e_2))$). Additional parallelism is provided by functional languages with *non-strict semantics* where a function can execute in parallel with the evaluation of its arguments (*vertical parallelism*, e.g. between f and g in $f(g(e))$). This makes sense if a function does not immediately need its arguments to proceed execution, in particular if an argument is a compound data structure that can be produced in parallel with its consumption.

The *dataflow model* is a technique to exploit the parallelism of functional languages where each expression is evaluated when the required data are available (data-driven execution). This model is often related to special purpose architectures (dataflow machines) but it can also serve as a basis for the compilation of functional languages for conventional parallel computers. For instance, a compiler might generate code for an abstract dataflow machine that is is then boiled down to actual machine code [Sch90]. Alternatively, also a conventional imperative language might be chosen as the compilation target. The compiler for the functional language would then become simpler and the imperative target program could be executed on any architecture by recompilation with conventional compilers.

Of course, there are several features of dataflow languages that cannot be directly compiled into an imperative form. Moreover, for each parallel dialect of an imperative language, there exist different extensions that are not portable at all. One solution of this problem is to encapsulate all machine-dependent features into a small set of library functions that offer the necessary facilities to express the parallel behavior of the functional program. This library has to be ported to each parallel architecture, while the compiler and the target programs may remain unchanged.

In this paper, we present such a library $\mathcal{DF}$ (for dataflow) consisting of a few primitives that allow to express dataflow behavior in any imperative language. The library has been implemented in the C language on a distributed memory system and on a shared memory computer and several dataflow programs have been manually translated into this form. After the presentation of $\mathcal{DF}$, we describe its implementation and the results we gained from our experiments. A comparison with other parallel models and an outlook on our further work conclude this paper.

2 The $\mathcal{DF}$ Library

2.1 Required Features

In this section, we sketch the motivation for an imperative parallel programming model that may serve as a portable and efficient target for the compilation of dataflow languages.

```
-- sort list
quicksort ::  [int] -> [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

The properties that such a model must fulfill shall be demonstrated by a concrete example: 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 by this generalization avoids an additional append operation). Figure 2 shows the dataflow graph of the recursive branch of the function.

`split` subdivides 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 the remaining ones. The first recursive invocation of `qsort` sorts b and appends r and passes the results to the second recursive invocation that concatenates this list with the sorted version of a. The 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 `split`. Hence, in a strict execution model the program could be executed in a sequential fashion only.

However, a non-strict model offers inherent vertical parallelism: since `qsort` reads its first input list element after element and only in its final step requires the second list, `split` can work simultaneously with both `qsort` invocations. While `split` generates the elements of a and b, both `qsort` branches can start to process these elements; synchronization between both branches takes only place when finally one recursive branch 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:

- **Processes:** Functions play the role of processes, i.e. the unit of parallelization is the user-defined function (function-level parallelism, macro-dataflow).
- **Channels:** Shared variables represent the communication medium between processes. Communication is directed and one-to-many.

- **Non-Strict Functions:** Functions can start execution independently of their arguments. Only when arguments are accessed, synchronization takes place.
- **Non-Strict Data Structures:** Data structures can contain empty slots. When such slots are accessed, synchronization must take place.
- **Garbage Collection:** Data structures are not explicitly deallocated. If a structure is referenced no more, it must therefore be automatically reclaimed.

The $\mathcal{DF}$ library aims to support these principles in a conventional programming language on conventional parallel computers.

2.2 The $\mathcal{DF}$ Kernel

In this subsection, the kernel of the $\mathcal{DF}$ library is presented which introduces two basic concepts:

- **Processes** which are the units of parallel execution and
- **Structures** by which processes communicate and synchronize.

The structures of the $\mathcal{DF}$ library are actually arrays where each array slot may contain (besides integer atoms and references to other arrays) the special constant $\perp$ (read "empty"). Structure references may be communicated between processes regardless whether the structures yet contain values or not. Any process that tries to read an empty slot is automatically blocked until the slot is filled by another process. Furthermore, a process may write values into empty slots only ("single assignment rule") releasing all processes blocked on this slot.

The kernel of the $\mathcal{DF}$ library consists of the following types, constants and functions (input parameters are denoted by $\downarrow$, output parameters by $\uparrow$):

REF is a type of all references to structures. This type is word-sized and includes the special constant NIL.

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 the special constant of type REF that is different from any valid structure reference.

par($\downarrow fun$, $\downarrow proc$, $\downarrow n$, $\downarrow arg_1$, $\downarrow arg_2$, ..., $\downarrow arg_n$) starts on processor *proc* a process executing $fun(arg_1, arg_2, \ldots, arg_n)$. *proc* must be either some processor address or -1 (in that case an automatic strategy for process distribution is applied). *fun* must not return any value, not reference any global environment (except the constants described above) and not cause any side-effects (besides the operations described below).

new($\downarrow size$, $\uparrow ref$) allocates a structure with *size* slots and returns a 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 that can be stored in that slot.

get($\downarrow ref$, $\downarrow index$, $\uparrow value$) reads the slot at position *index* of the structure denoted by *ref* and returns 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($\downarrow ref$, $\downarrow index$, $\downarrow value$) writes 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 kernel of the $\mathcal{DF}$ library contains all the features necessary to express dataflow programs. Later on, it will be extended by the concept of *indirections* and by functions to support *garbage collection*.

2.3 A Sample $\mathcal{DF}$ Program

Figure 3 shows a translation of the `qsort` function of subsection 2.1 into the $\mathcal{DF}$ kernel bound to some imperative language. This translation is guided by the attempt to construct the program's dataflow graph as illustrated in Figure 2:

1. **Allocate Channels:** Channels (that denote the arcs of the dataflow graph) are created to establish the communication between the parallel processes. These channels are actually empty structures into which some processes write their results while other processes read their inputs from them.

2. **Spawn Processes:** Then the processes (that denote the nodes in the dataflow graph) are started with those structures as input arguments that correspond to input or output channels. The processes read their input values from these structures and write their results into them.

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.

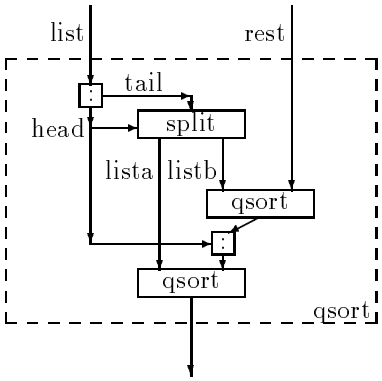

Figure 2: Dataflow Graph of `qsort`

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

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

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

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

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

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

  get(list, 0, ↑hd); put(head, 1, hd);              -- local operations
  get(list, 1, ↑tl); put(tail, 1, tl);
  put(cons, 1, resultb); put(resultb, 0, hd);

end qsort;
```

Figure 3: $\mathcal{DF}$ Version of Quicksort

The $\mathcal{DF}$ version of `qsort` in Figure 3 demonstrates these three phases: First the communication channels are allocated, then one `split` and two `qsort` processes are spawned and finally several local operations are performed that connect the input channels of the procedure with the input channels of the spawned processes: the head and the tail of the input list are received and written into the input structures of the `split` process. Furthermore, the head is linked with the result list of the second `qsort` process and a reference to this new list is written into an input structure of the first `qsort` invocation.

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

2.4 Indirections

A problem with our $\mathcal{DF}$ version of `qsort` is the fact that due to limited memory resources we can actually create processes only down to a certain level in the "recursion tree". Beyond this level, the `qsort` calls must be executed in a recursive and thus sequential manner. However, then a subtle difficulty arises: Each `qsort` process calls `split` and then steps down to the next level until for the first time the base case is reached (the process is than in one "corner" of the recursion tree). In this situation, `qsort` needs the argument `rest` from the next process:

```
if list == NIL then
  get(rest, 1, ↑rest);
  put(result, 1, rest);
  return;
end;
```

Unfortunately, that process will deliver this value only after it has traversed its whole recursion tree and so all processes must basically execute in sequence. This problem did not arise in our original solution, since there each leaf of the recursion tree was a process of its own and all solutions could be computed in parallel. In a coarse-grain dataflow model, parallelism is often seriously restricted by processes that wait for values in order to forward them to other processes. Hence, we introduce the concept of *indirections*:

puti($\downarrow$*ref,* $\downarrow$*index,* $\downarrow$*ivalue*) This procedure behaves like *put* but may be only applied on a reference *ivalue* (called *indirection*). Any get operation on slot *index* will not return *ivalue* itself but the contents of the slot *index* of the structure referenced by *ivalue*.

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

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

The *result* cell does not receive the reference stored in *rest* but a reference to *rest* itself. Thus no get must be performed and the current process is not blocked.

Summarizing, the application of indirections allows to decouple processes such that a parent process may propagate results from one subprocess to another without being blocked itself. Actually, an indirection is nothing but a cheap replacement for an additional subprocess spawned only in order to link the two other subprocesses.

2.5 Garbage Collection

The $\mathcal{DF}$ library supports garbage collection by the *weighted reference counting* algorithm of [Bev87]: 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 environment). If a reference is deleted, the count of the referenced structure is decreased by that weight.

Our version of this algorithm is based on three additional $\mathcal{DF}$ operations:

putr($\downarrow$*ref*, $\downarrow$*index*, $\downarrow$*rvalue*) 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($\updownarrow$*ref*) halves the weight of the reference *ref* (if this weight is a power of two) or sets it to zero (if the weight is one). This operation is usually performed before the reference is *applied* (see below).

del($\updownarrow$*ref*) invalidates the reference *ref*. *ref* must not be *applied* any more (see the description below).

The operation new allocates an empty structure and delivers a reference with a weight that is initially a power of two. The current process and all the processes to which the reference is communicated are now in charge of the correct management of the weight of the reference.

A reference is *applied*, if it is an argument of a get or a put(i/r) operations. In this case, the count of the referenced structure is automatically decreased by the weight of the reference (i.e. we assume that the reference will not be applied any more). However, if we perform a cop before the actual application, the weight of the reference is halved and the reference count of the structure is only decreased by this value. Hence, further applications of the reference are still possible.

If we know that a reference is applied for the last time, the preceding cop operation can be omitted and the reference count is decreased by the whole weight of the reference (which means that the reference is considered to be deleted). If we miss the point of the last application (i.e. a reference is available but of no use any more), a del operation can be performed that has no other effect but decreasing the reference count in the structure.

The main advantage of our scheme is that in most cases the explicit deletion of references can be avoided and no additional access to the referenced tuple is necessary (which in a distributed memory environment involves a message to another processor). If the last application of a reference can be determined at compile time, the access operations get and put(i/r) themselves are in charge of decreasing the reference counts, since the messages submitted by these operations additionally carry the reference weight information.

If a reference has a weight of one that can not be halved any more, this weight is decreased to zero. Hence, one weight unity is lost and the reference count of the corresponding structure will never become zero again. 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). However, any implementor of the $\mathcal{DF}$ model is free to choose other strategies, e.g. the automatic insertion of indirection cells as suggested in [Bev87].

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

C-$\mathcal{DF}$ is the implementation of the $\mathcal{DF}$ library for the C language. Up to now, there exist two versions the this library on two very different kinds of parallel architectures:

1. A distributed memory computer consisting of 16 T800 transputers.
2. A shared memory machine of type Sequent Symmetry with 20 i80386 processors.

In both cases, the library was almost entirely implemented in C (with a few lines of inline assembly code). In the next subsection, we show the layout of $\mathcal{DF}$ structures which is common to both implementations. Afterwards, we describe both implementations (in particular the process management) in more detail.

3.1 Structure Layout

A structure is a sequence of 32 bit words as depicted in Figure 4: The first two halfwords contain the structure's length and its reference count, respectively. The next word contains pairs of available/reference bits: bit $2*i$ of this word is set, if slot i of the structure contains a value, bit $2 * i + 1$ signals whether this value is a reference to another structure or not. The following words are the actual structure slots. Since in the current implementation there is only one word reserved for slot management, a structure may contain up to 16 slots (an extension to an arbitrary length structures is easily possible).

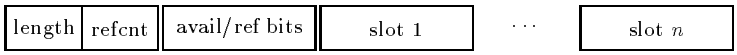

Figure 4: Structure Layout

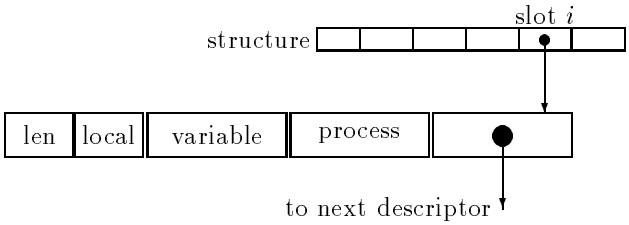

Figure 5: A Process Descriptor

Structure references are 32 bit words constructed as follows: The most significant 5 bits denote the number of the processor that the referenced structure is stored on (not used for the shared memory implementation). The next 3 bits contain the weight of the reference represented as its binary logarithm plus one (a value of zero denotes zero weight). The least significant 24 bits denote the position of the referenced structure relative to the start of the heap. The kernel operations then perform the following tasks:

- new allocates a structure, stores its length and sets the reference count to the maximum weight. The available and reference bits are reset and the slots are initialized with NULL. The weight of the returned reference is set to its maximum.
- get delivers the value stored in the referenced slot (if the slot contains a reference, the weight of this reference is halved before). If the slot is empty, then the slot actually contains a list of blocked process descriptors. In this case, a new descriptor is allocated as depicted in Figure 5 and linked into the list. The process is then descheduled leaving its address in the descriptor.
- put traverses the list of process descriptors linked into the slot. For each process, the desired value is stored at the given address, and the process is reawaked. Finally, the value is written into the slot and the corresponding 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.

3.2 The Distributed Memory Implementation

For the transputer implementation of C-$\mathcal{DF}$, we used the facilites of the Logical Systems Transputer Toolset which contains a compiler for a parallel version of C with the possibility of using inline assembly code [Log90].

3.2.1 Process Creation

The transputer has two registers that determine the current process by the address of the next instruction and the address of the process workspace, respectively. Furthermore, 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 whose address is stored at the end of the process queue.

For starting a parallel process with the par routine, a new process workspace is allocated from the heap. The workspace is initialized with the address of the function to be executed, with its arguments and with the return address of the process on its termination. The process is then started by queuing its workspace into the list of descheduled processes.

If a process is blocked on a get operation, the same scheme is applied as for normal descheduling with the exception that the process workspace is not linked into the scheduling queue but into the corresponding structure slot.

3.2.2 The Communication System

The transputer is equipped with four links each of which allows transmission of messages to other processors. The counterpart of these physical links are software channels that allow uni-directional, unbuffered and point-to-point communication between processes. Since the $\mathcal{DF}$ structure space is distributed among all processors, a system for communication between all processors had to be implemented consisting of one input process and one output process per link (see Figure 6).

If a process wants to send a message to another processor, it writes this message into the buffer of the corresponding link from where it will be forwarded by the output process. If an input process receives a message from a neighbor processor, this message is (according to a routing table) either forwarded to some output process or stored in the buffer of the local server process. This process is in charge of managing the message which may denote either a request from a remote process or a reply to a request that was once submitted by a local process.

3.2.3 Structure Caches

On distributed memory systems, the ratio between tprocessor power and communication bandwidth is still very unbalanced. Hence, a main concern in the design of the $\mathcal{DF}$ model was to minimize the number of messages to be exchanged (e.g. process creation requires only one message). However, the main load of the interconnection network is caused by

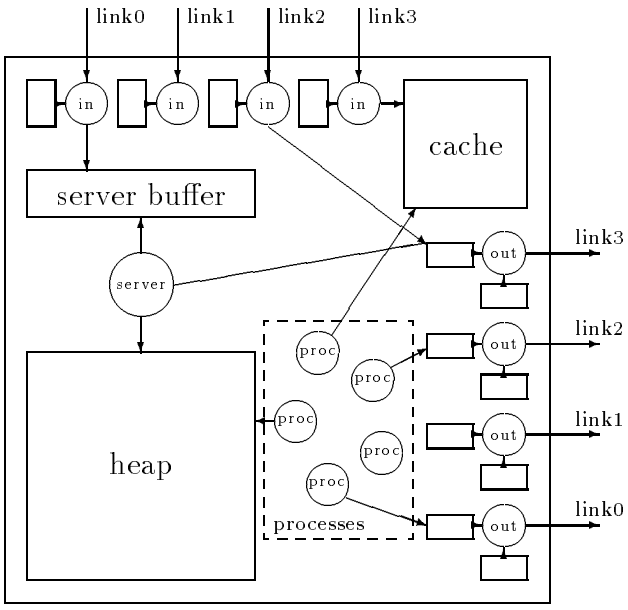

Figure 6: The Communication System

the access to remote structures. If a structure is stored on one processor and demanded by a process on another processor, for each field of of this structure one request has to be sent and one message has to be returned.

In order to decrease network traffic, we introduced in our implementation the concept of a *cache*: if one field of a remote structure is accessed, also the other fields will be probably demanded. For example, if a list is processed, usually both the head and the tail field of each list cell are read. Thus, we 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 read the cell from the cache. The cache is in particular of vital importance for structures that are "global" in the sense that they are read by many processes on that processor.

3.3 The Shared Memory Implementation

A shared memory version of the $\mathcal{DF}$ library was implemented on a Sequent Symmetry S81 computer which is operated under Dynix/ptx, a parallel version of UNIX System V. However, for reasons of efficiency we decided not to apply the UNIX facilities for parallelism, because then the creation of a process would take 10000 times longer than on the transputer. Instead, we built the $\mathcal{DF}$ library on top of the μSystem [BMS91], a parallel programming package available for a variety of UNIX machines. The μSystem introduces the concept of μTasks i.e. light-weight threads of control that are scheduled among a set of virtual processors (implemented by UNIX processes). A $\mathcal{DF}$ process is

$\mathcal{DF}$-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 7: Speed of C-$\mathcal{DF}$ Operations

then the extension of a μTask; the creation of a such a process is 100 times cheaper than that of a real UNIX process.

The $\mathcal{DF}$ heap is in this implementation a global data area which all $\mathcal{DF}$ processes may directly access. Hence, in contrast to the distributed memory version, no communication subsystem and no caching is required. However, a major problem arises due to the fact that the heap is a critical region (protected by a semaphore) that may be read and modified by one $\mathcal{DF}$ process at a time only. Since all other processes then have to wait, this central resource may represent a significant bottleneck. In order to overcome this problem, we split the heap into several subareas that are separately protected. If a process wants to access the heap, the most significant bits of the structure reference are used to determine the appropriate subarea to be locked. $\mathcal{DF}$ processes may thus simultaneously access different subareas of the heap without interference.

4 Experiments

4.1 Absolute Performance

It is common practice to restrict the presentation of parallel languages to to the relative speedups that can be achieved. However, we think that one should also speak frankly about the absolute performance of a system. We have therefore measured the average time of the $\mathcal{DF}$ kernel operations (all executed on local structures) and compared the results to the time required for performing corresponding operations of the C language itself. Figure 7 gives the results measured on the transputer system (the results are similar in the shared memory implementation except for process creation which is there 300 times more expensive than a simple function call).

We think that these results are quite satisfying. Due to lack of human resources, we concentrated our efforts during the implementation of C-$\mathcal{DF}$ mainly on portability. For a really efficient implementation, several critical parts would have to be rewritten in assembly language which would increase the speed of the $\mathcal{DF}$ operations by a factor of 2–3. 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.

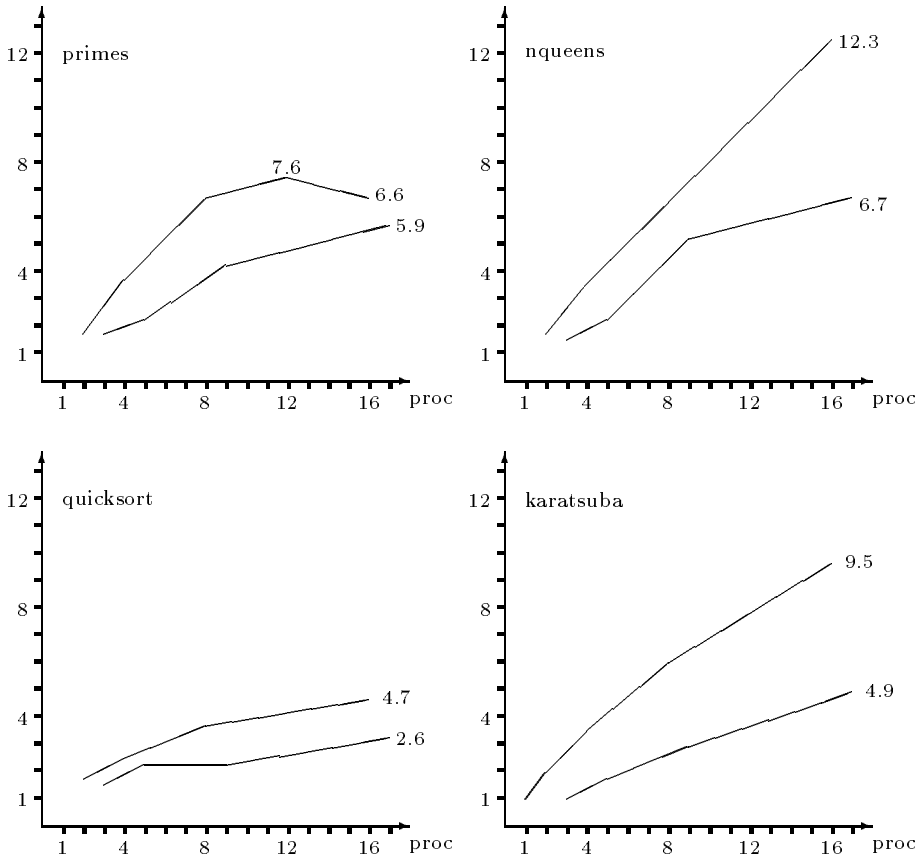

Figure 8: Relative Speedups

4.2 Relative Speedups

We measured the performance of both implementations of C-$\mathcal{DF}$ with several programs that represent typical functional algorithms. The transputer system was configured to hypercube topologies of dimensions 0 to 4 where an additional root processor was linked into one edge of the hypercube. In the shared memory implementation, up to 16 processors were applied. Figure 8 shows the relative speedups of these programs (the lower line always denotes the distributed memory version).

- `primes` computes all primes up to 1000000 where each number is tested in parallel (vertical parallelism). The speedup grows quite well with the number of processors (the slight slowdown in the shared memory version denotes some anomaly still to be explained). On the transputer, the program profited much from the caching mechanism, since the same global list of primes is read by all processes.

- `queens` computes all solutions of the queens problem on a board of size 10. The program is based on a parallel traversal of the search tree; parallel processes are

created in the first few levels only. The performance scales in both systems very well with the number of processors and achieves the best speedups.

- `quicksort` sorts a list of 64000 random numbers. Processes are created up to that level in the recursion tree such that all processors become saturated with work. Since in both implementations the achieved speedups are rather poor, the algorithm actually seems to have far less inherent parallelism than we originally expected.
- `karatsuba` multiplies two long integers of 512 digits each. The algorithm is based on a divide and conquer approach with three recursive function calls. Parallel processes are created up to that recursion level such that all processors become saturated. The shared memory version shows very good speedups, the results of the distributed memory version are significantly worse.

On the one hand, the results of these experiments are very satisfying for the shared memory version of the $\mathcal{DF}$ library (maximum speedup of 12 with 16 processors). On the other hand, they also show the limits of the distributed memory implementation: in the transputer system, communication throughput is actually not in balance with the performance of the single processors, which represents a serious bottleneck for many applications.

5 Related Work

In this section, we give a short overview on the relationship of our work to that of other researcher in this area.

5.1 I-Structures

The most important influence to the $\mathcal{DF}$ library comes from the work done by the dataflow group at the MIT. The structures of the $\mathcal{DF}$ library are actually *I-structures* [ANP87] that we have modified and extended for the application in an imperative language. These extensions include the explicit support for garbage collection and in particular the concept of indirections that is in our opinion essential for the coarse-grain parallelization of functional languages (and that can not be expressed in the I-structure model or in related models as *futures* [Hal85]).

Moreover, we want to emphasize that we are interested in the implementation of functional languages on conventional parallel hardware and not on dataflow machines. Our main concern therefore was to encapsulate all issues of non-strictness, communication and synchronization in a single model that can be efficiently added to a conventional programming language. One contribution of the $\mathcal{DF}$ library is the realization that non-strict structures alone are sufficient for all these items, e.g. it is not necessary to include a non-strict argument passing mechanisms as in [AN88]. Moreover, from our experiments we conclude that special I-structure hardware is not an absolute must for coarse-grain parallel programs.

5.2 Linda

LINDA [CG88] is one of the few programming models that is portable among different parallel architectures. Unfortunately, the model is not really suitable for implementing a functional language, since it allows access to shared objects only by matching their contents against given patterns. It would be far too inefficient to simulate direct memory access to list cells by a search for cells matching a particular "address pattern". Moreover, since there are no explicit object references in LINDA, no form of automatic garbage collection is (nor can be) supported.

The main relation between LINDA and $\mathcal{DF}$ is the fact that both models base process synchronization on the availability of data: A LINDA process s blocked, if it tries to receive a tuple that is not yet in the tuple space, while a process in $\mathcal{DF}$ 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!

5.3 Alfala and Buckwheat

ALFALA is an implementation of the functional language ALFL for the iPSC hypercube, while BUCKWHEAT is the implementation of this language for a shared memory machine [Gol88]. We mention these systems 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 serial combinators which represent sequential tasks that reduce different parts of the program graph. These combinators are translated into pieces of C code that are compiled and linked to a parallel program.

The main difference to our work is the fact that we rely on the dataflow concept (that is eager) while that work is based on graph reduction (that is lazy). Besides differences in the compilation strategy, the choice of the reduction model seems to have serious impacts on the efficiency. From the figures in [Gol88], C-$\mathcal{DF}$ seems roughly 10–100 times faster and extremely more memory efficient than ALFALA (the input sizes in that publication were actually too small to compare them with the statistics in this paper). However, we have to emphasize that their programs were generated by a compiler while ours are still written by hand.

6 Conclusions and Future Work

The $\mathcal{DF}$ library is only a first step in the parallel implementation of a functional language. Starting from function definition, we are able to generate $\mathcal{DF}$ programs that can be quite efficiently executed on different parallel architectures. However, it still remains to show that this translation can be performed in an automatic way. Our current work concentrates on the development of such a parallelizing compiler. Different from other approaches that create many fine-grained tasks, we aim to detect the main source of parallelism in the program in order to spawn only a few coarse-grain processes. This way of parallelization should then be more suitable for todays parallel computers.

References

[AN88] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *PARLE 87, Conference on Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, volume 258 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

[ANP87] 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.

[Bev87] D. I. Bevan. Distributed Garbage Collection Using Reference Counting. In *PARLE 87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

[BMS91] Peter Buhr, Hamish Macdonald, and Richard Stroobosscher. μSystem Annotated Reference Manual (Version 4.4). Technical report, Department of Computer Science, University of Waterloo, Canada, October 1991.

[CG88] Nicholas Carriero and David Gelernter. Linda in Context. Technical Report YALEU/DCS/RR-622, Department of Computer Science, Yale University, New Haven, Connecticut, 1988.

[Gol88] Benjamin F. Goldberg. Multiprocessor Execution of Functional Programs. Research Report YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988.

[Hal85] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Log90] Logical Systems, Corvallis, Oregon. *Transputer Toolset Version 89.1*, January 1990.

[Sch90] Wolfgang Schreiner. ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language, Diploma Thesis, Johannes Kepler University, Linz, Austria, October 1990.