A Simple Model of Distributed Functional Data Structures and Its Implementation*

Tyng–Ruey Chuang

Institute of Information Science
Academia Sinica
Taipei 115, Taiwan

trc@iis.sinica.edu.tw

Abstract

We report results from experimenting with a parallel functional programming environment based on distributed data structures. The main results are: 1) A distribution model to support efficient fold/unfold operations over data structures. The distribution model is simple yet general, and easy to implement. 2) A novel programming environment assembled from currently available hardware/software systems. Functional programs execute in SPMD (Single Program Multiple Data) style under this environment and exhibit good speedup.

1 Motivation

The development in recent years on computer systems, software tools, and functional languages seem to have converged to a point where it is natural to conduct the following experiment: to build an environment for parallel functional programming by assembling readily available hardware and software systems. We have just done that, on and off for two months starting in mid February, 1998, and the outcome is quite satisfactory. We report here results from our experiment, with emphasis on a simple model of distributed functional data structures.

Developments in the following four areas are sufficiently mature for us to conduct this experiment.

Hardware system. Workstation clusters based on distributed or shared memory systems are very affordable today. Multiple CPU's are interconnected on the desktop ready to be programmed.

Communication library. There exist standard, portable, and efficient libraries to exchange data among multiple processors over network or via shared memory. Examples include MPI (Message Passing Interface) and PVM (Parallel Virtual Machine). These libraries all have C language interface.

Functional language. Many implementations of modern functional programming languages also have C language interface. Examples include Objective Caml, SML/NJ, and Glasgow Haskell. Also quite importantly, the parametric module systems of the ML family provide a convenient way to build new libraries (i.e., structures in ML) based on existing libraries (i.e., functors). This abstraction power and the associated discipline help derive type-safe and distributed implementations of algebraic data types (ADT's).

Programming model. Functional programs, and their parallelizations, based on fold/unfold operations over algebraic data types are well studied.[1] The SPMD (Single Program Multiple Data) model for programming symmetric multiprocessors is also well understood.[2]

*Alternatively titled as *Do-it-yourself parallel functional programming*. This paper is available on-line as technical report TR-IIS-98-010 from the Institute of Information Science, Academia Sinica, via `http://www.iis.sinica.edu.tw`.

[1] The fold/unfold primitives are the bases of the Bird–Meertens formalism, and are also called catamorphisms/anamorphisms [9].

[2] Under the SPMD models, many copies of the same program execute at the same time, one on each processor. Data are exchanged among the various executing programs by ex-

Fold/unfold operations are the high-level functional abstractions we aim to implement, by using low-level side-effecting SPMD code, for programming multiprocessor systems.

We use the following specific components from the above four areas to conduct the experiment: IBM SP2 [15], MPI [10], Objective ML [8], and fold/unfold primitives [9] with SPMD programming model. From these, we quickly build up a working environment, using *existing* hardware/software resources, for parallel execution of functional programs, and observe good speedup. We are a little surprised by how well the experiment goes. (This certainly a testimony to the high quality of the components we used). Since we are not aware of similar experiments, we think our experience deserves to be shared.

We use the four selected hardware/software/model components because they are familiar and available to us. Other selections of components are certainly possible. The language Objective Caml plays important role in our experiment because it glues the other three components wonderfully. It generates native PowerPC code on IBM SP2 and provides C interface to MPI. It has a parametric module system that allows us to manage the generation of new, distributed, implementations of ADT's based on existing sequential implementations.

There are two goals we set up to achieve.

(1) The same program executes both on single- and multi-processor systems, using our distributed implementations of ADT's. If the programs use fold/unfold operations, they should exhibit speedup on multiprocessor systems.

(2) The module signature of the distributed implementation of an ADT is the same as the one of the sequential implementation. However, customized distributions can be specified by programmers by additional predicates (this will be made clear in Section 2). The distributed data is used in a purely functional style, and the users do not worry about low-level mechanism for data distribution.

We shown in the next section, Section 2, the distribution model and implementation principle we used to achieve (2). Section 3 show results that confirm (1). We then relate our approach to other work in Section 4.

plicitly subroutine calls to the underlining communication library.

2 A Simple Model of Distribution

We assume that the functional program is executed under the SPMD model on multiple processors. However, unlike imperative SPMD programs, there will be no explicit calls to underlining communication library to exchange data among the processors. On each processor, the program executes as if it has the whole data structure, and the outcome from all processors will be identical. Note that functional languages have its advantage over imperative languages in parallel processing because these data structures, though distributed, will not be side-effected once they are generated.

The distribution model we use is a simple one. Parts of a data structure can either be replicated or partitioned. If the part is replicated, then each processor have a copy of the part and knows that it is a replica. If the part is partitioned, then only one processor has it, it knows that it has the only copy, and all other processors know that this processor has it. Collective operations on a distributed structure are equally straightforward. For data that is replicated, each processor operates on its own replica, independent to one another. For data that has been partitioned, the processor that owns the unique copy operates on it and sends the result to all processors. Furthermore, work on data partitions can be carried out independently and in parallel on all processors. After partial results from all partitions are computed, however, they are exchanged in a coordinated matter such that all processors get all partial results. Then each processor merges its local set of partial results independently.

What we describe above is at the implementtaion level. This is to be provided by us by a mechanical transformation of the original implementation of ADT's. At the user-level, programmers use our distributed implementations of ADT's and don't concern such details. Computations on data structures of those ADT's will be automatically distributed. We will use an example, of data type `tree`, throughout this section to explain the model and its implementation in detail. The general principle, however, apply to other data types.

2.1 Polynomial data types and their fixed points

The functional data structures we distribute are values of *polynomial* (or, *sum-of-product*) data types and their fixed points. It is called a polynomial data type because the set of its values is the disjoint sum of several sets of product values, with products from different sets distinguished by value constructors ("tags").

The following is an example of polynomial data type. Values of type `quad` are all pairs, but are distinguished from on another by the four tags `AA`, `AB`, `BA`, `BB`. The types of pair-products are parameterized by type variables `'a` and `'b`.

```
type ('a, 'b) quad = AA of 'a * 'a
                   | AB of 'a * 'b
                   | BA of 'b * 'a
                   | BB of 'b * 'b
```

The following recursive definition of data type `tree`:

```
type 'a tree = AA of 'a       * 'a
             | AB of 'a       * 'a tree
             | BA of 'a tree  * 'a
             | BB of 'a tree  * 'a tree
```

can be alternatively expressed as the fixed point of the following type equation:

```
type 'a tree = ('a, 'a tree) quad
```

where `quad` is viewed as a type function of two variables. In Objective Caml (as in many functional languages), one needs an extra tag, named `Rec` below, when defining `tree` as the fixed point of `quad`:

```
type 'a tree = Rec of ('a, 'a tree) quad
```

Accompanying the above definition, is a pair of injection and projection functions `up` and `down` that move values in between types `('a, 'a tree) quad` and `'a tree`.[3]

```
let up   t       = Rec t
let down (Rec q) = q
```

Note that one can choose an identifier other than `Rec` as the tag in the definition of `tree`. Functions `up` and `down` provide a level of abstraction that separates us from this non-essential naming decision.

[3]These two functions are usually named `in` and `out`. However, `in` is a keyword in Objective Caml, so we use `up` and `down`.

2.2 Fold/unfold functions over recursive data types

It is well known that for a data type that can be expressed as the fixed point of a polynomial, as shown above for type `tree`, fold/unfold functions can be systematically defined to provide reduction/generation operations for values of that type, as shown below for type `tree`.

```
let rec fold f t =
    match down t
     with AA (u, v) -> f (AA (u, v))
        | AB (u, v) -> f (AB (u, fold f v))
        | BA (u, v) -> f (BA (fold f u, v))
        | BB (u, v) -> f (BB (fold f u,
                              fold f v))

let rec unfold g s =
    match g s
     with AA (u, v) -> up (AA (u, v))
        | AB (u, v) -> up (AB (u, unfold g v))
        | BA (u, v) -> up (BA (unfold g u, v))
        | BB (u, v) -> up (BB (unfold g u,
                               unfold g v))
```

We put all related declarations about data type `tree` together as a signature (module type in Objective Caml) and name it as `TREE` in Figure 4 in Appendix A.

The above definitions of fold/unfold also express potential parallelism during the evaluation of their return values. For if the tag is `BB`, then evaluations for the two constituting parts of this product can be conducted in an independent and parallel matter. The challenge is to specify this distribution of work such that it can be easily understood and utilized by users, and efficiently implemented by existing hardware and software systems.

2.3 The distribution model

We now describe in detail how values of type `tree` are distributed, and, once distributed, how they are accessed. Though we use type `tree` as an example, the distribution model is not specific to it, and can be applied to any polynomial data types and their fixed points. To ease further discussion, we say that a functional data structure is built up with *nodes*, which are tagged values of product types. In a node, its constituting parts may again refer to other

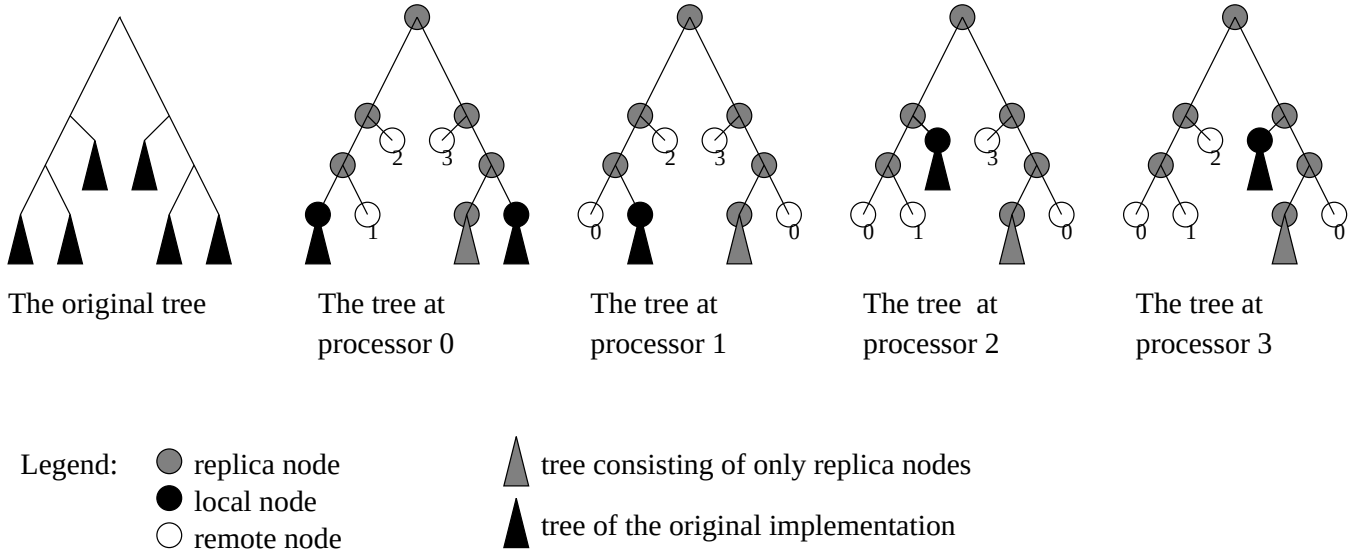

Figure 1: A tree distributed to four processors.

nodes. Note that in strict functional languages, programmers cannot construct values whose nodes refer to one another in cycles (except when mutable values are used). The nodes of a value will be structured like a directed acyclic graph (dag).

As an example, for `tree` value, the nodes will be tagged `quad` values and they form no cycle. For the following discussion, we limit all nodes to be tagged `quad` values, and the term *tree* refers to `tree` value. A node can have parents if it is used to build another tree (via the `up` function), and it can have children if parts of it are also trees (as exemplified by nodes with `AB`, `BA`, or `BB` tags). In single-processor execution of tree programs, all nodes reside in (the memory) of the single processor. In multiple-processor execution, nodes are distributed over processors. When a tree is distributed, its constituting nodes at each processor are classified into three disjoint categories: *replica* nodes, *local* nodes, and *remote* nodes. The three categories represent the following three disjoint distribution properties:

(a) A replica node is replicated at all processors.

(b) A local node is available only at one processor.

(c) At each other processor where it does not have the local node, it has a corresponding remote node.

with the following additional constraints:

(d) A replica node can have replica, local, and remote nodes as children.

(e) Local and remote nodes do not have any child.

(f) A local node contains the sub-tree consisting of itself and all of its descendant nodes in their original, non-distributed, implementations. This sub-tree is called *local tree*.

(g) A remote node keeps the id number of the only processor that has the corresponding local tree.

Using this data distribution model, we can easily describe how a tree is distributed. We use the term *distribution tree* to refer to the tree consisting of replica, local, and remote nodes (but not the local trees) at each processor. Figure 1 shows a tree distributed to four processors. The original tree has 6 sub-trees, of which one is replicated to all processors. The other 5 sub-tree are assigned to the four processors in a round robin matter. They become local trees and each is headed by a local node. Given above data distribution model, the corresponding data access model for fold/unfold operations is quite straightforward.

For fold operation:

(1) Each processor travels its distribution tree and perform fold operations only at local trees. This is a synchronization-free stage.

(2) Each processor travels its distribution tree and perform fold operations over all nodes. If it encounters a local node, it already folded the local tree at stage (1). It now broadcasts this result to all other processors. If it encounters

a remote node, then it waits the result to be broadcast. After the broadcasting, the fold operation resumes, independently, on all processors. Note that the trees are traveled in the same order on all processors, and as a consequence, the broadcasting operations will be matched in order as well. This stage needs m synchronizing broadcasts where m is the total number of local nodes over all processors.

For unfold operation:

(1) Each processor generates its distribution tree. A user-provided predicate is used to guide whether a node should be replicated or not by applying it to the node. If the predicate returns true, then the node is replicated; and replicated nodes are recursively unfolded to produce distribution tree. Otherwise the node will not be replicated, and a processor will be assigned to generate the corresponding local tree at the next stage; all other processors are instructed to have the corresponding remote nodes. This is a synchronization-free stage.

(2) Each processor travels its distribution tree and generates its own local trees. This stage is synchronization-free.

The cost involved by a fold/unfold operation can be estimated as well. For a tree of n nodes in the original implementation, let $replica(n)$ denote the number of replica nodes in the distribution tree at one processor (note: the number is the same for all processors), and $local_i(n)$ the total size of local trees at processor i. Suppose there are k processors. Then the total work, $work(n)$, and elapsed time, $time(n)$, for a fold/unfold operation over the distributed data structure will be

$$\begin{aligned} work(n) &= w_1 \cdot k \cdot replica(n) + w_2 \cdot \sum_{0 \le i < k} local_i(n) \\ time(n) &= t_1 \cdot replica(n) + t_2 \cdot \max_{0 \le i < k} local_i(n) \end{aligned}$$

where w_1, w_2, t_1, t_2 are some constants. For fold operation, an additional term for broadcasting must be added to the *time* function. (We will address broadcast issues shortly in Section 2.4.) The two equations are quite evident from the above two-stage descriptions of distributed fold/unfold operations.

2.4 Values *vs.* references

When folding a distributed tree, the result from folding the local tree associated with a local node must broadcast to all the corresponding remote nodes. How is this result sent? Should it be sent as a data item readily to be used (i.e., value), or just a notice informing the remote node its availability at the local node (i.e., reference)? There are tradeoff between sending as values and references. Values are more convenient as they can be immediately consumed at the other ends. However, they pose two problems. First, the values have to be packed at the sending side and unpacked at the receiving side using an external format. The task can be time-consuming and not type-safe. We are greatly helped by Objective Caml's `Marshal` package, which provides a pair of `to_string` and `from_string` functions that packs/unpacks a value of any type to/from a string. The string is broadcast at the local-node side and received at the remote-node sides. The second problem is that sometimes one really wants the value to remain at the local-node side. For example, a tree mirroring function can be formulated as a fold function, and one would expect the local tree mirror stays at the local node. This reduces communication overhead greatly. In general, if the fold function is a tree-to-tree structural transformation operation, then it would be better to let partial results remain local; hence no broadcast operation is needed.

Currently two versions of fold are provided. One for value result and the other one for reference result. The value one has identical type signature of the original (non-distributed) fold function. The reference one is a little complicated as one will need to pass additional information about the resultant distributed structure.

Similarly, two versions of unfold are provided as well. One version will always generate a distributed tree with a root-level local node at one processor (hence all other processors with root-level remote nodes). That is, the whole tree effectively resides in one processor, and other processors only have references to it. As a consequence, fold operations on the resultant tree cannot be parallelized. This version has identical type signature of the original unfold function. The other unfold function need an additional predicate to tell it whether the currently generated node will be replicated or not. If the predicate returns true, replica nodes are generated

at all processors. If not, a local node is generated at only one processor. When generating the local tree associated with a local node, the predicate is not consulted. This way, the programmers can supplied (high-level) policy for data distribution while (low-level) distribution mechanism is implemented by the unfold operation. The predicates are like data distribution directives (which are user hints supplied as special comments) in High Performance Fortran [6]. Only that they are much more flexible (because they are general functions) and they are not just hints (because distributions are dictated by them).

Here is another note related to value *vs.* reference: Sometimes, one would like a sub-tree to be shared by several nodes in such a way that no multiple copy of the sub-trees is generated. However, we do not do so in the original unfold operation. As a consequence, in the distributed unfold operation, multiple copies of a local tree will be generated and assigned to processors as well. We can design a unfold function that preserve sharing by first comparing the to be generated value to all previous generated values, and reuse the previously generated value if there is a match. A sharing-preserving distributed unfold operation could be similarly implemented but it would need cross-processor communication for checking shared values. This seems complicated and expensive. We decide adhere to the simple, not sharing-preserving, unfold operation.

2.5 Incremental data accesses

Incremental data accesses produce and consume distributed data using the `up` and `down` functions. Currently, the `up` function always produce replica nodes. Similarly, `down` will produce a replica `quad` value when applied to a replica node. However, if the `down` function is applied to a local/remote node, then it produces a `quad` value with its constituting nodes being respectively local/remote. This implementation decision ensures that each `down` function call takes only constant time (though it may involve communication in case of a local/remote node). The other implementation decision, that `down` call always produces a result that is entirely replicated, has cost proportionally to the size of the value it is applied to.

If the input to a `down` call is a local/remote node, then this node is side-effected to become a replica node with the returned `quad` value (whose constituting nodes are still local/remote). This way, repeated `down` invocations to the same local/remote node will not incur repeated communication overhead. Only the first call needs to broadcast the resultant node.

2.6 Data redistribution and load-balancing issues

In our model, we do not provide functions for data redistribution. Instead, data will be explicitly regenerated by programmers to have the desirable distribution from existing data. This is often done by first using a fold operation on the old data to replicate its value to all processors. After the replicated data is adjusted (to have balanced heights, for example) at all processors, a unfold operation is used on the adjusted data to distribute it to all processors. This means that load-balancing has to be explicitly performed by the programmers.

Nevertheless, a limited form of task parallelism is provided by our current implementation. Recall that a reference version of unfold is provided to generate the entire tree local to one processor, and the corresponding remote nodes to all other processors. Note as well that this version of unfold is synchronization-free. Therefore, if a sequence of this kind of unfold operations are invoked, and the processors assigned to generate the top-level local trees take terms in a round robin matter, then the over-all computation load can be distributed.

2.7 Objective Caml binding for MPI routines

We use the C interfacing facility provided by Objective Caml to bind the C interfaces of MPI routines. Only 5 MPI-related Object Caml functions are defined. Their names and types are:

```
val initialize: string array -> unit
val finalize: unit -> unit

val size: unit -> int
val rank: unit -> int

val broadcast: 'a option -> int -> 'a
```

Functions `initialize` and `finalize` are called before and after all MPI routines are invoked respectively, as required by MPI. Functions `size` and `rank` are used, respectively, to query the total number of

processors (np) and the id of the current processor. The processor id is a unique integer number starting from 0 and less than np.

The `broadcast` function sends a value of arbitrary type from a processor to all processors. At the sending end, it is called as

```
broadcast (Some value) root
```

where `value` is the value to be broadcast, and `root` the sender's processor id. At the receiving ends, it is called as

```
broadcast None root
```

where again `root` is the sender's processor id. Th execution of `broadcast` is blocking: the execution is resumed only when all processors have issued the broadcast function and the value is exchanged. All `broadcast` functions, including the one at the sending end, return with `value`. The original broadcast function in MPI (`MPI_BCAST`) is very restrictive. It requires that both the sending end and the receiving ends all spell out the type of the data to be exchanged, as well as the location and size of the sending/receiving buffers. Our Objective Caml implementation of broadcast is more abstract and easier to use. We are greatly helped by Objective Caml's `Marshal` package which pack/unpack a value of arbitrary type to/from a string. The string is then exchanged using the native `MPI_BCAST` functions. Note that, although `broadcast` is polymorphic, it is not type-safe because pack and unpack are not. If the receiving end insists on interpreting the broadcast string by the wrong type, it will get the wrong value.

2.8 A little code walk

We show here some code segments in the implementation of distributed `tree` values and the associated fold/unfold operations. All declarations related to the distributed version of `tree` are collected in a signature called `D'TREE`, as shown in Figure 4 in appendix. We define a functor `D'Tree` that takes a structure with `TREE` signature (i.e., the original implementation for `tree`) and produces a structure with signature `D'TREE`. What we show here are some code segments in functor `D'Tree`.

First, the type `node` denotes the three kinds of nodes in a distribution tree:

```
type ('a, 'b) node = Replica of 'a
                   | Local   of 'b
                   | Remote  of int
```

Now the type `tree` of signature `D'TREE` is defined as

```
type 'a tree = Rec of (('a, 'a tree) quad,
                       'a Base.tree) node ref
```

in functor `D'Tree`. In the above, `Base` is the original implementation module for `tree`, and `Base.tree` the type of the original tree.

The following function, `fold3`, travels the distribution tree and applies functions `f`, `g`, and `h` to its nodes depending on their tags.

```
let rec fold3 (f, g, h) t =
    match t with Rec r -> match !r
     with Replica (AA (u, v)) -> f (AA (u, v))
        | Replica (AB (u, v)) ->
                f (AB (u, fold3 (f, g, h) v))
        | Replica (BA (u, v)) ->
                f (BA (fold3 (f, g, h) u, v))
        | Replica (BB (u, v)) ->
                f (BB (fold3 (f, g, h) u,
                       fold3 (f, g, h) v))
        | Local  s -> g s
        | Remote k -> h k
```

It is used to build a two-stage traversal function `fold'`:

```
let fold' (f, g) (f', build') t =
    let up'     x = build' (Replica x)
 in let local'  x = build' (Local   x)
 in let remote' x = build' (Remote  x)
 in
    fold3 (up' $ f', local', remote') (
    fold3 (up, local $ (Base.fold (g $ f)),
           remote)  t)
```

In the above, symbol `$` is the infix operator for function composition. The first pair of functions, `(f, g)`, tells how to fold a local tree, while the second pair, `(f', build')`, tells how to fold the distribution tree. The `fold'` function can be thought as the reference version of the fold function as the argument `build'` is often supplied with value `build` in signature `D'TREE`. Function `build` is the default mapping from `node` values to distributed `tree` values, and contains no synchronizing broadcast. Functions `build`, `up`, `local`, and `remote` can be considered as the constructors for distributed `tree` value, and are defined as

```
let build  n = Rec (ref n)
let up     q = build (Replica q)
let local  t = build (Local t)
let remote k = build (Remote k)
```

An illustrating example will be shown in Section 3, where `fold'` is used to perform tree mirroring.

The value version of the fold function is then defined as

```
let fold f t =
    let pid = MPI.rank ()
 in let id x = x
 in let synch node =
        match node
         with Replica x -> x
            | Local   x ->
              MPI.broadcast (Some x) pid
            | Remote  x ->
              MPI.broadcast None x
 in
    fold' (f, id) (f, synch) t
```

where explicit broadcasting is used to exchange data during the folding of the distribution tree.

Similarly, two unfold functions are provided. Function `unfold'` needs an additional user-suppiled predicate to tell if the generated node will be replicated over all processors or not. Of the nodes not to be replicated, each will be placed on a single processor in a round robin matter, and the corresponding local tree generated at the processor. Function `unfold`, which will always generate the whole tree as a root-level local tree on some processor, is defined as

```
let unfold g s = unfold' (g, fun x -> false) s
```

where predicate `fun x -> false` says no node will be replicated.

3 Examples

We show five test cases and their performance results. The code segements of the test cases are shown in Figure 2 and 3. In the code segments, structure `T` is the original implemenetation of `tree`, and structure `DT` the distributed implementation. Each test case first uses unfold to generate a tree, then applys fold to the generated tree. Wall clock time[4] is measured for the completion for the two operations, not including the startup time for loading the program to the Parallel Operation Environment of IBM SP2. We run the programs on a non-dedicated cluster of 8 processors. Each reported time is the result of a single run, and may be effected by the computation load of the machine at the time. (The computation load of the machine is very light, however.) We test each case for 5 scenarios: original version (using structure T) on one processor, distributed version (using structure DT) on one, two, four, and eight processors. Measurements from the first two scenarios give us some feedback of the overhead of the distribution implementation over the original imlpementation.

The test case **count** first generates a (height-balanced) tree of n numbers $(0, 1, 2, \ldots, n-1)$ using generation function `iota`, then counts the numbers in the tree using reduction function `count`. The tree generation is guided by a predicate `iota_rep` so that the tree is evenly distributed to all available processors. (For a tree of 1000 numbers over 4 processors, processor 1 will have the local tree with numbers 0 to 249, processor 2 the local tree with numbers 250 to 499, etc.)

Similarly, the test case **swap** generates the same tree, but perform a tree mirroring operation on the tree. Two versions of swap are tested. The value version results in each processor receiving a mirror of the entire tree, while the reference version let the lcoal mirrors stay distributed. Note that the same reduction function `swap` is used in all versions of the swap test case.

In the test cases of count and swap, the trees can be made to evenly distributed to all available processors. Hence, for load-balancing purpose, it is sufficent to let each processor has only one local tree, and let all local trees be of the same size. For some applications, we may not know how to evenly disrtibute the tree in this matter. For such applications, we can parition the tree into many local trees (more than the number of available processors) of sufficient granularity and let the unfold operation assign the local trees to processors in a round robin matter. This achieves load-balancing. The test case **fib** is such an example. We first generate the "call graph" for function $fib(n)$, where the call graph is a tree with leave values either being integer 0 or 1. A leave value of k represents a call to function $fib(k)$. Since $fib(0) = 1$ and $fib(1) = 1$, the number of leaves is exactly the value of $fib(n)$. Once the call graph is

[4] The `Unix.gettimeofday` function in Objective Caml is used.

Common code segment:

```
module T = Tree(Quad)
module DT = D'Tree(Common)(T)

let n = .... (* data size *)

let iota (x, y) =
    if y - x = 1
       then AA (x, y)
       else if y - x = 2
               then AB (x, (x+1, y))
               else BB ((x, (x+y)/2),
                        ((x+y)/2+1, y))

let iota_rep (x, y) = (* if true, replicate *)
    (y - x + 1) > (n / (MPI.size ()))

let fibtree x =
    if x = 2
       then AA (0, 1)
       else if x = 3
               then AB (1, 2)
               else BB (x-2, x-1)

let fibtree_rep x = (* if true, replicate *)
    (n - x) <= (MPI.size ())

let count quad =
    match quad
     with AA (u, v) -> 2
        | AB (u, v) -> 1 + v
        | BA (u, v) -> u + 1
        | BB (u, v) -> u + v

let swap quad =
    match quad
     with AA (u, v) -> AA (v, u)
        | AB (u, v) -> BA (v, u)
        | BA (u, v) -> AB (v, u)
        | BB (u, v) -> BB (v, u)

let rec mirror tree =
    match T.down tree
     with AA (u, v) -> T.up (AA (v, u))
        | AB (u, v) -> T.up (BA (mirror v, u))
        | BA (u, v) -> T.up (AB (v, mirror u))
        | BB (u, v) -> T.up (BB (mirror v,
                                 mirror u))

let rec d_mirror tree =
    match DT.down tree
     with AA (u, v) -> DT.up (AA (v, u))
        | AB (u, v) -> DT.up (BA (d_mirror v, u))
        | BA (u, v) -> DT.up (AB (v, d_mirror u))
        | BB (u, v) -> DT.up (BB (d_mirror v,
                                  d_mirror u))
```

Figure 2: Five test cases. Part 1.

count (original):

```
let d = T.unfold iota (0, n-1)
let m = T.fold count d
```

count (distributed):

```
let d = DT.unfold' (iota, iota_rep) (0, n-1)
let m = DT.fold count d
```

swap (original):

```
let d = T.unfold iota (0, n-1)
let b = T.fold (T.up $ swap) d
```

swap (distributed, by value):

```
let d = DT.unfold' (iota, iota_rep) (0, n-1)
let b = DT.fold (T.up $ swap) d
```

swap (distributed, by reference):

```
let d = DT.unfold' (iota, iota_rep) (0, n-1)
let b = DT.fold' (swap, T.up) (swap, DT.build) d;
```

fib (original):

```
let d = T.unfold fibtree n
let m = T.fold count d
```

fib (distributed):

```
let d = DT.unfold' (fibtree, fibtree_rep) n
let m = DT.fold count d
```

direct swap, no fold (original):

```
let d = T.unfold iota (0, n-1)
let b = mirror d
```

direct swap, no fold (distributed):

```
let d = DT.unfold' (iota, replicate) (0, n-1)
let b = d_mirror d
```

Figure 3: Five test cases. Part 2.

generated, we assign the sub-tree corresponding to $fib(x)$ as a local tree to a processor if $n-x > k$. This ensures there are sufficient number of local trees, and each local tree is of sufficient granularity. Test case fib also shows how distributed data structures can be used to drive parallel functional evaluations (even if no data structure is evident in the definition of the functions).

The final test case is **direct swap** which is used not to measure potential speedup, but to measure the overhead of distributed incremental data accesses. Direct recursive definition is used to define the tree mirroring function, instead of using fold operation. There is no expected speedup in programs

written in this style. Furthermore, traveling a whole tree using only `down` calls will involve excess communication overhead for the distributed tree, and, as shown by the timing results, the overhead also increases as more processors take parts during the communication.

As shown by the performance results in Table 1, the distributed versions achieve good speedup once the data sizes are large enough and fold/unfold operations are used. It also shows that the reference version of distributed fold operation does perfrom much better than the value version, just as one expects. We like to point out two observations. First, the one-processor run of the distributed, value version, of the swap test case takes very long time (1175.37 sec.). This probably is caused by packing/unpacking a tree of 1,000,000 numbers to/from a string and sending/receiving it to/from itself. The physical memory probably is exhausted, so we observe very bad paging behavior. Second, there are several "super-linear" speedup: e.g. 8-processor run of distributed count *vs.* the original run (8.01 *vs.* 69.49), and 8-processor run of distributed swap (reference version) *vs.* the original run (17.01 *vs.* 144.25). This is entirely possible because not only we have four times the CPU power, we also have four times the physical memory space. This helps reduce garbage collection time.

We breifly mention two issues in our approach to distributed functional data structures. The first issue is: how does one handle distributed structural transformation functions between different ADT's? For example, can one transform a distributed tree value into a distributed list value by using the reference version of `fold'` function? One can do that by using functions `List.up` and `DistList.build` (of the corresponding `list` algebraic data type and its distributed implementation, instead of `T.up` and `DT.build` in the swap case), and use a suitable `quad` to `cons` reduction function in the call to function `fold'`.

The second issue is about data structures that are distributed in more complicated forms. We can go beyond unfold-based distributions and build customized distributed data structures by using the `up`, `local` and `remote` functions and processor id's. For example, the following expression places a local tree at processor 0, and have all other processors refer to processor 0 for it. It then builds a distributed tree `my_tree` based on that tree at all processors.

count:

data size	original	distributed			
	1 proc.	1 proc.	2 proc.	4 proc.	8 proc.
10^6	69.49	72.61	34.16	16.86	8.01
10^5	5.24	5.27	2.89	1.08	0.76
10^4	0.10	0.10	0.23	0.60	0.57
10^3	0.00	0.00	0.04	0.66	0.64

swap (value version):

data size	original	distributed			
	1 proc.	1 proc.	2 proc.	4 proc.	8 proc.
10^6	144.25	1175.37	163.29	116.20	114.35
10^5	12.21	16.76	9.65	8.35	6.98
10^4	0.20	0.46	0.78	1.22	0.80
10^3	0.04	0.02	0.64	0.69	1.05

swap (reference version):

data size	original	distributed			
	1 proc.	1 proc.	2 proc.	4 proc.	8 proc.
10^6	144.25	219.05	88.27	35.60	17.01
10^5	12.21	18.46	10.43	4.78	1.56
10^4	0.20	0.92	1.33	1.21	0.66
10^3	0.04	0.01	0.70	0.74	0.62

fib:

data size	original	distributed			
	1 proc.	1 proc.	2 proc.	4 proc.	8 proc.
32	247.17	255.75	154.89	74.02	27.75
30	94.47	98.72	56.09	21.37	12.07
20	0.09	0.12	0.18	1.04	0.52
10	0.00	0.10	0.09	0.08	0.08

direct swap, no fold:

data size	original	distributed			
	1 proc.	1 proc.	2 proc.	4 proc.	8 proc.
10^6	134.87	1536.29	N/A	N/A	N/A
10^5	10.14	69.38	136.70	162.99	239.03
10^4	0.11	5.64	12.22	15.74	13.06
10^3	0.00	0.07	1.00	1.64	1.35

Note: N/A means the execution time is too lengthy to be included here.

Table 1: Performance results, in seconds.

```
let my_tree =
if   MPI.rank () = 0
then DT.up (AB (0, DT.local (T.up (AA (1, 2)))))
     (* this branch executes on proc. 0 *)

else DT.up (AB (0, DT.remote 0))
     (* this executes on all other procs. *)
```

Though more flexible, programs written in this way are easy to be in errors and difficult to reason about.

4 Related Work and Conclusion

Parallel function programming is a broad area that attracts many researchers [5]. Approaches based on fine-grain data parallelism and coarse-grain task parallelism have been well studied (see, for example [1, 4, 11, 14]). Parallelism exploited by functional primitives or program skeletons is also well understood (see, for example [2, 3, 7, 9, 13]). Distributed data structures have been active research subjects as well (see, for example [12]).

Our work differs from the above work in several ways. First, previous works on parallel executions of functional programs often assume a shared memory model where data is shared among the processors with almost no cost [1, 4, 11]. Their emphases are task creation and management, e.g. on scheduling the task queue, and on concurrent memory management. On the contrary, our functional programs are executed in SPMD style (hence no need of task scheduling). However, we assume a distributed memory model (which is more realistic for common workstation clusters); hence data distribution and communication must be carefully managed. Like High Performance Fortran, we separate the policy of data distribution (which is controled by users) from the mechanism of data distribution (which is the responsibility of the system). However, our distribution policies can be much more flexible.

Secondly, our work shares the common themes of exploiting parallelism by using structured primitives (or skeletons) [1, 2, 3, 7, 9, 13]. However, many of the skeleton approaches have yet to be realized as running systems. The exceptions are works on APL and NESL, and works on using structured Fortran and C prgram skeletons to compose larger parallel programs. They deal with limited kinds of data types, such as array and list, while ours applied to any polynomial data types.

We have presented a programming environment based on a simple model of distributed data structures for parallel functional programming. Our approach can be considered as coarse-grain data parallelism. The environment is assembled and integrated using currently available hardware and software systems. Our experiements and results show that such a "do-it-youself" environment can be set up, functional programs using distributed data structures are easy to construct, and parallelism in the programs is indeed expressed and exploited. We expect to run larger programs using the environment, and to further experiment with new designs and implemenations of data distribution models.

References

[1] Guy E. Blelloch. NESL: A nested data–parallel language (version 3.1). Technical Report CMU–CS–95–170, School of Computer Science, Carnegie Mellon University, USA, 1995.

[2] Wai-Mee Ching and Alex Katz. An experimental APL compiler for a distributed memory parallel machine. In *Proceedings of Supercomputing '94*, pages 59–68. Washington, D. C., USA, November 1994. IEEE Computer Society Press.

[3] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[4] Benjamin F. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Department of Computer Science, Yale University, April 1988. Available as technical report YALEU/DCS/RR-618.

[5] Kevin Hammond. Parallel functional programming: An introduction. In *First International Symposium on Parallel Symbolic Computation*, September 1994. World Scientific Publishing Company.

[6] High Performance Fortran Forum. High Performance Fortran Language Specification, version 2.0. Technical report, Center for Re-

search on Parallel Computation, Rice University, Texas, USA, January 1997.

[7] Paul Kelly. *Functional Programming for Loosely–coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[8] Xavier Leroy. *The Object Caml system release 1.07: Documentation and user's manual.* INRIA, France, December 1997.

[9] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and bared wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144. Cambridge, MA, USA, August 1991. Lecture Notes in Computer Science, Volume 523, Springer–Verlag.

[10] Message Passing Interface Forum. MPI: A Message–Passing Interface Standard, version 2.0. Technical report, University of Tennessee, Knoxville, Tennessee, USA, July 1997.

[11] R. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transaction on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[12] Anne Rogers, Martin Carlisle, and John H. Reppy. Supporting dynamic data structures on distributed-memory machines. *ACM Transaction on Programming Languages and Systems*, 17(2):233–263, March 1995.

[13] David B. Skillicorn. Architecture–independent parallel computation. *IEEE Computer*, 23(12):38–43, December 1990.

[14] Guy L. Steele Jr and W. Daniel Hillis. Connection Machine Lisp: Fine–grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 279–297. Cambridge, MA, USA, ACM, August 1986.

[15] Various Authors. Scalable parallel computing. *IBM Systems Journal*, 34(2):143–325, 1995.

A Module Signatures

```
module type QUAD =
sig
 type ('a, 'b) quad = AA of 'a * 'a
                    | AB of 'a * 'b
                    | BA of 'b * 'a
                    | BB of 'b * 'b
end

module type COMMON =
sig
 type ('a, 'b) node = Replica of 'a
                    | Local   of 'b
                    | Remote  of int
end

module type TREE =
sig
 module Quad: QUAD
 type ('a, 'b) quad = ('a, 'b) Quad.quad
 type 'a tree

 val up: ('a, 'a tree) quad -> 'a tree
 val down: 'a tree -> ('a, 'a tree) quad

 val fold: (('a,'b) quad->'b) -> 'a tree -> 'b
 val unfold: ('b->('a,'b) quad) -> 'b -> 'a tree
end

module type D'TREE =
sig
 module Common: COMMON
 module Base: TREE
 type ('a, 'b) node = ('a, 'b) Common.node
 type ('a, 'b) quad = ('a, 'b) Base.Quad.quad
 type 'a tree

 val build: (('a, 'a tree) quad,
             'a Base.tree) node -> 'a tree
 val up: ('a, 'a tree) quad -> 'a tree
 val local: 'a Base.tree -> 'a tree
 val remote: int -> 'a tree
 val down: 'a tree -> ('a, 'a tree) quad

 val fold: (('a,'b) quad->'b) -> 'a tree -> 'b
 val unfold: ('b->('a,'b) quad) -> 'b -> 'a tree

 val fold': ((('a,'b) quad->'c) * ('c->'b))
     -> ((('a,'d) quad->'f) * (('f,'b) node->'d))
     -> 'a tree -> 'd
 val unfold': (('b->('a,'b) quad) * ('b->bool))
              -> 'b -> 'a tree
end
```

Figure 4: Signatures of `TREE` and `D'TREE`, for original and distributed implementations of data type `tree`.