An Overview of the Parallel Language Id
(a foundation for pH, a parallel dialect of Haskell)

Rishiyur S. Nikhil

Digital Equipment Corporation
Cambridge Research Laboratory

Draft: September 1993

Abstract: Id is an architecture-independent, general-purpose parallel programming language that has evolved and been in use for a number of years. Id does not have a sequential core; rather, it is implicitly parallel, with the programmer introducing sequencing explicitly only if necessary. Id is a mostly-functional language, in the family of non-strict functional languages with a Hindley-Milner static type system. Unlike other non-strict functional languages, it uses lenient, not lazy evaluation, for reasons of parallelism, as well as to give meaning to non-functional constructs. The non-functional constructs come in two layers: I-structures (which preserve determinacy, but not referential transparency, and are closely related to logic variables), and M-structures, which are side-effects with implicit synchronization. The layers are distinguished by syntax and types, so that it is possible for an implementation to force a program to be within a desired layer (*e.g.,* purely functional).

In this paper, we present an overview of Id and its implementation, focusing on its novel features and properties. We relate it to parallel languages in general, but also specifically to other functional languages. Our presentation is based on examples, with references to other papers for formal syntax and semantics.

We also discuss a recent development: recognizing the large overlap between Id and Haskell, Id researchers have decided to transform Id into pH, a parallel dialect of Haskell.

1 Introduction

Today, parallel programming is still not a routine activity, *i.e.,* one cannot write a parallel program with the same ease as sequential programs and also reasonably expect a performance improvement when run on a parallel machine. Parallel programming today often involves serious compromises in ease of coding, and/or a lot of effort in adjusting and tuning for performance, much of which is architecture- or machine-specific, making programs non-portable and difficult to understand and maintain.

The first generation of successful parallel programs[1] are so-called "multicomputer" programs– separate programs for each processor that communicate by explicit message-passing. Implementing these languages is easy, because the programmer bears the entire burden of optimization for parallelism, namely data and process decomposition, load balancing, message optimization, *etc.* However, programming and debugging in this style is notoriously difficult and non-portable [24]. A minor aspect of non-portability is simply incompatible message-passing libraries from different vendors, but this issue is receding, with the advent of systems such as PVM [39] and MPI [29]. However, the major reason for non-portability is that optimizations for parallelism are often specific to a machine, and these are simply too tedious to rewrite manually for different machines or machine configurations.

[1] By "successful", we mean useful application programs that achieve performance gains through parallelism.

Author's address: Digital Equipment Corporation, Cambridge Research Laboratory, One Kendall Square, Bldg. 700, Cambridge MA 02139, USA. Email: `nikhil@crl.dec.com`

HPF (High Performance Fortran) [16, 25] is a recent development that is an excellent abstraction of a certain common paradigm in message-passing programs, the so called SPMD (Single Program, Multiple Data) or "data parallel" paradigm. Interestingly, HPF is a retraction back to implicitly parallel programming and away from the explicit parallelism of message-passing programs. In fact, an HPF program may be understood (and debugged) using sequential semantics, a deterministic world that we are comfortable with. Once again, as in traditional programming, the programmer works with a single address space, treating an array as a single, monolithic object, regardless of how it may be distributed across the memories of a parallel machine. The programmer does specify data distributions, but these are at a very high level and only have the status of hints to the compiler, which is responsible for the actual data distribution and other optimizations for parallelism (it is even free to ignore the programmer's hints, just as modern C compilers sometimes ignore `register` hints).

Although HPF is an excellent solution for certain regular, grid-like, parallel computations on arrays, it is not a general-purpose parallel language. Programs that involve parallel recursive control structures and/or dynamic linked data structures, such as trees, graphs and sparse arrays, are outside the scope of current HPF parallelization capabilities.

Id is also an implicitly-parallel, single address-space language, but it is not restricted to regular, grid-like parallel computations. Like HPF, the programmer, to first approximation, does not worry about processes, data distribution, messages and the like. Programming in Id is comparable to programming in SML [28], Haskell [19] or Scheme [11]. The programmer may provide hints to the implementation about how to distribute functions and data but, like HPF, these only have the status of hints and do not directly affect the semantics of the program. Unlike HPF, Id is also capable of expressing unstructured, dynamic parallelism:

- The Id programmer can define a variety of parallel data structures, including trees, graphs, hash tables, queues, *etc.*
- The shape and size of data structures and computations may be highly data dependent.
- Id programs may obtain much parallelism from producer-consumer situations, where a data structure is consumed by one process even as it is being built by another.

In this paper, we provide an overview of Id, illustrating all these points. We begin in Section 2 with a description of the layered structure of Id: the purely functional core and its two non-functional extensions.

In Section 3 we describe the functional core of Id, and contrast it with other functional languages such as Scheme, SML and Haskell. The major novel features of Id relative to these languages are *array comprehensions*, a purely functional notation for building arrays, and Id's *lenient* evaluation order, which is a non-strict, but not lazy, evaluation order.

In Section 4 we describe the *I-structure* layer. I-structures are not functional– we concede referential transparency– but they are a limited form of side-effects that preserve determinacy (the Church-Rosser property), which is invaluable in parallel programs because it provides a certain degree of repeatability for debugging. I-structures are closely related to logic variables in logic programming.

In Section 5 we describe the *M-structure* layer. These are true side-effecting operations, and so may introduce non-determinacy into a program, but they are novel in the way they integrate synchronization with data access.

In Section 7 we outline our implementation of Id, emphasizing the treatment of locality, which is essential for good performance in a parallel machine.

In Section 8 we compare Id to other parallel languages. After a few more remarks regarding Id and HPF, we contrast our approach to other parallel functional languages, parallel logic programming languages and parallel object-oriented programming languages.

Finally, in Section 9, we conclude with a description of results, current implementation status and plans.

The original Id language was designed by Arvind and his colleagues at University of California, Irvine [3]. The primary influence was machines, not languages, since it was originally conceived as a textual notation for dataflow graphs, a parallel machine language. Work on this language continued under Arvind's direction at MIT until 1985.

The author joined MIT in 1984 (with no background in dataflow machines or dataflow graphs), and designed a new language called Id/83s for the 1985 MIT summer course 6.83s [32]. The primary influence was languages, not machines, in particular the language ML with which the author had extensive experience. This language evolved over the years with various names (such as Id Nouveau), but is now, once again, just called Id.

The core of Id is a non-strict functional language which, semantically, is closely related to Haskell, the pure, non-strict functional language [19] that is now gaining wide acceptance. Recognizing this, a group of researchers, including the author, met in August 1993 and decided to transform Id into *pH* (for parallel Haskell), using Haskell notation for the functional core. This move has attracted much interest from functional programming researchers who were not previously in the Id community. The Id compilers at Digital CRL and MIT are both being modified to accept pH syntax instead of Id. Since the concepts are identical, this paper could have been written with pH notation, except that at time of writing, pH notation has not yet been fully settled. This situation may change in time for the final version of this paper.

2 The Layers of Id

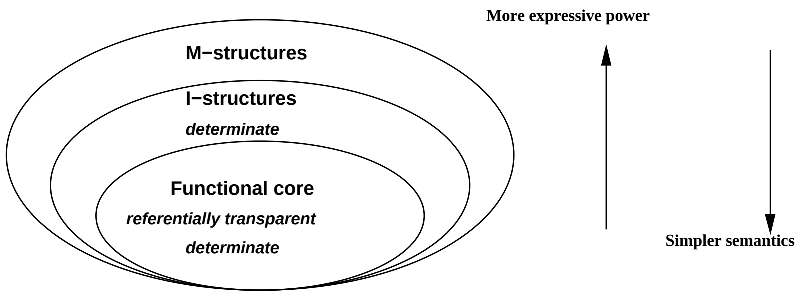


Figure 1: The layers of Id

Figure 1 depicts the layers of Id. Going outward from the purely functional layer increases expressive power, but at the cost of a lesser ability to reason about programs. The inner layer is a purely functional, non-strict language. Being referentially transparent, it supports equational reasoning, which is a powerful tool for programmers in proving programs correct, and for compilers in transforming programs [8].

The I-structure layer loses referential transparency and, hence, the power of equational reasoning. However, it preserves determinacy (the Church-Rosser property), which is a very powerful tool for the programmer, especially the *parallel* programmer, because it gives a degree of repeatability during debugging. I-structures are data structures whose fields may be assigned only once. Formally, the value of a field can only be refined from $\perp$ to a value, and hence they are related to logic variables. An Id program restricted to this layer is guaranteed to have the same result (including the same errors) when run with different schedules in the runtime system (*e.g.,* on different machines, or even the same machine under different conditions).

M-structures allow reads and writes on data structures, and so they are general side-effects, but they are novel in the way they integrate synchronization with data access. The M-structure layer concedes determinacy. The principal use that we have found for M-structures is in codes where non-determinacy at the program level can be exploited usefully (for more parallelism, or for less storage) in expressing an algorithm that is deterministic in at the application level. For example, M-structures permit us to implement a hash table in which non-deterministic inserts allow us to build it with much parallelism; although the exact structure of the final table is non-deterministic, the semantics of lookups on the final table is unaffected.

When we say that I- and M-structures add expressive power, this is, of course, in a subjective sense, since even the functional core is Turing complete. By increased expressive power we mean that the programs are more perspicuous, more obviously parallel, and more obviously storage-efficient. We hope our examples in Sections 4 and 5 will bear this out. For the functional purist who remains unconvinced, we hope that these examples can be taken as a challenge to be matched, in readability and in performance, by purely functional solutions (which we would enthusiastically embrace).

The I-structure and M-structure layers are distinguished by syntax and by types, just as the side-effecting operations of SML have special notation, functions and types. Hence, like SML, it is possible to have a compiler flag that forces a program to be within a particular layer, *e.g.,* the purely functional subset. Unfortunately, like SML, this is non-trivial to integrate with separate compilation– it is impossible to tell, purely from a function's interface, whether it is restricted to a particular layer or not. Current research indicates that there is some hope that it will be possible to encapsulate I-structure-using functions as pure-functions, but this is still speculative.

3 The Functional Core of Id

Ignoring differences in concrete syntax, the functional core of Id is very similar to SML and Haskell. It has a lambda-calculus core, a Hindley-Milner polymorphic type system, curried application by juxtaposition, user-defined algebraic types (which are disjoint unions of product types), and pattern-matching notation both to discriminate on disjoint unions and to bind variables to components of data structures. Here is the obligatory definition of polymorphic lists and the `map` function on lists:

```
type list *0 = Nil | Cons *0 (list *0);

def  map  f  Nil           = Nil
|    map  f  (Cons x xs)   = Cons  (f x)  (map  f  xs);
```

"`*0`" is a type variable, for polymorphic lists.

As in other functional languages, the list constructor `Cons` can also be written with an infix colon `(x:xs)`, and there is also notation for "list comprehensions", which capture certain common forms of nested maps and filters. We assume the reader is familiar with SML or Haskell, and do not elaborate on this further.

A novel bit of notation in Id is the "array comprehension", for constructing entire arrays. For example:

```
  {2d_array  ((1,N),(1,N))  of
   | [i,j] = 1   || i <- 2 to N, j <- 1 to (i-1)     % upper triangle
   | [i,i] = 0   || i <- 1 to N                      % diagonal
   | [i,j] = -1 || i <- 1 to N-1, j <- (i+1) to N  % lower triangle}
```

constructs an $N \times N$ matrix with 1, 0 and -1 in the upper triangle, diagonal and lower triangle, respectively. Haskell borrowed this idea from Id and adopted a similar notation.

Non-strictness

Id has non-strict semantics. Technically, this means it is possible that $f\bot \neq \bot$, *i.e.,* it is possible for a function to produce an answer even if its argument diverges.

Consider a function to compute the maximum number in a list of numbers, as well as all "large" numbers, *i.e.,* numbers within 10 percent of that maximum. Here is how it might be written in a strict functional language, such as SML:

```
def f xs = {  m  = loop1  xs  -inf
              ys = loop2  m  xs  Nil
```

```
          In
            (m, ys) } ;

def   loop1  Nil     xm    = xm
|     loop1  (x:xs)  xm    = { xm'  = if (xm > x) then xm else x;
                             In
                               loop1  xs  xm' } ;

def   loop2  m  Nil     ys = ys
|     loop2  m  (x:xs)  ys = { ys' = if (x > 0.9*m) then (x:ys) else ys
                             In
                               loop2  m  xs  ys' };
```

The notation "{ *decl*;..;*decl* In *e*}" is a *block*, analogous to `letrec` blocks in Scheme. `Loop1` traverses the input list `xs`, carrying a running maximum `xm`, initially `-inf`. At the end of the list, it returns `xm`, which is bound to `m`. `Loop2` traverses the list a second time, collecting the large numbers (initially `Nil`).

In Id, and other non-strict functional languages, the two list traversals can be reduced to one:

```
def f xs = {  (m, ys) = loop  m  xs  -inf  Nil
          In
            (m, ys) };

def   loop  m  Nil     xm  ys = (xm, ys)
|     loop  m  (x:xs)  xm  ys = { xm'  = if (xm > x) then xm else x;
                                  ys'  = if (x > 0.9*m) then (x:ys) else ys
                                In
                                  loop  m  xs  xm' ys' };
```

Note that `m` is not only a result of `loop`, but also an argument. Non-strictness allows us to invoke `loop` even though `m` is not yet defined. Recursively, the entire recursion can unfold, computing `m`, which is then fed back, which in turn allows `ys` to be computed.

Another example defines an array of the first N Fibonacci numbers:

```
fib_array = {  A = {array (1,N) of
                    | [1] = 1
                    | [2] = 1
                    | [j] = A[j-1] + A[j-2] || j <- 3 to N}
            In
               A };
```

Note that `A` is both defined and used in the first equation.

This kind of non-strict evaluation has been found to be invaluable in a number of settings, such as writing parsers using higher-order combinators [20] (naturally captures the mutual recursion of grammars), expressing attribute grammars [22] (naturally captures the recursion between synthesized and inherited attibutes), and Successive Over-relaxation in arrays [26] (naturally captures the "wavefront" recursion, just like `fib_array` above).

Lenient Evaluation Order and Parallelism

Except for Id, every implementation of non-strict functional languages that we are aware of uses lazy evaluation. Id uses an evaluation order called "lenient evaluation" [42]. A precise definition may be given in terms of a redex selection strategy, as in [2] but, briefly, it amounts to evaluating *all* redexes in parallel except for those inside conditionals and lambdas. Since this subsumes the normal-order evaluation strategy, *i.e.,* it includes all redexes that a normal-order evaluator would reduce, it has the same termination properties as normal-order (and lazy evaluation). Unlike a lazy evaluator, however, it may "waste" work, *i.e.,* it may reduce redexes that are not necessary for the result.

Of course, a real implementation, due to limited resources, must choose a subset of available redexes. As long as it chooses fairly, *i.e.,* it never indefinitely postpones reduction of the normal order redexes, it still has the same termination properties.

Id has non-strict semantics, and uses lenient evaluation for several reasons: for expressive power as discussed above; for parallelism, and to give meaning to its non-functional constructs. We discuss parallelism in this section, and the relationship of evaluation order to non-functional constructs later as we introduce them.

Non-strict, lenient evaluation can have *exponentially* more parallelism than strict (call-by-value) evaluation. Consider the following program to compute a list containing the leaves of a binary tree.

```
type  tree =  Leaf  int | Node  tree  tree ;

def  leaves  t  =  aux  t  Nil ;

def aux  (Leaf j)    lvs  =  j:lvs
|   aux  (Node L R)  lvs  =  aux  L  (aux  R  lvs) ;
```

The type declaration trees specifies that a tree is either a leaf containing an integer or a node with two sub-trees. Leaves simply calls an auxiliary function, passing it the tree and an empty list. Aux, given a tree and a list of numbers, concatenates a list of the tree's leaves to the given list. Given a tree that is a leaf with number j, it just attaches j to the given list of leaves. Otherwise, it is given a tree that is an internal node with two subtrees L and R. It concatenates the leaves of R onto the given list, and concatenates the leaves of L onto that list, which gives us the result list.

In a strict implementation, in the second clause of aux, the recursive call of aux on R must complete before the recursive call of aux on L can begin. Thus, the computation becomes a sequential, right-to-left, depth-first traversal of the tree, building the list as it goes. In Id, on the other hand, the recursive call of aux on L can begin even though the list produced by the recursive call on R is not available yet. Thus, the traversal of both subtrees can proceed in parallel, producing exponentially more parallelism than the strict version.

This is clearly borne out in Figure 2, which shows parallelism profiles for two versions of this program when run on a full binary tree of depth 10. The parallelism profiles were collected as follows. The program was compiled into a fine-grain dataflow graph, where the vertices represent machine-level instructions and the arcs represent the transmission of data from one instruction to the next. The compilation method produces dataflow graphs that ensure that no execution occurs inside conditionals and lambdas. The dataflow graph was executed on a simulator that, at each time step, executed all "ready" instructions, *i.e.,* instructions whose inputs were available, *i.e.,* it faithfully follows lenient evaluation. The parallelism profiles are a record of how many instructions were executed at each time step. The upper profile is the normal one for Id, and the lower profile is for a version that was made artificially strict (but still parallel). The non-strict version could be completed in 250 time steps, with a maximum parallelism of 1776 (total instructions: 66,533), while the strict version took 26650 time steps, with a maximum parallelism of 4 (total instructions: 58,349).

Of course, parallelism profiles like these represent "ideal" executions and are not physically realizable. Nevertheless, they are useful to show upper bounds on the available parallelism in a program.

4 The I-structure layer

I-structures are a departure from pure functional semantics. A data structure in Id may have I-structure components. An I-structure component may be left *empty* when the data structure is created, and may

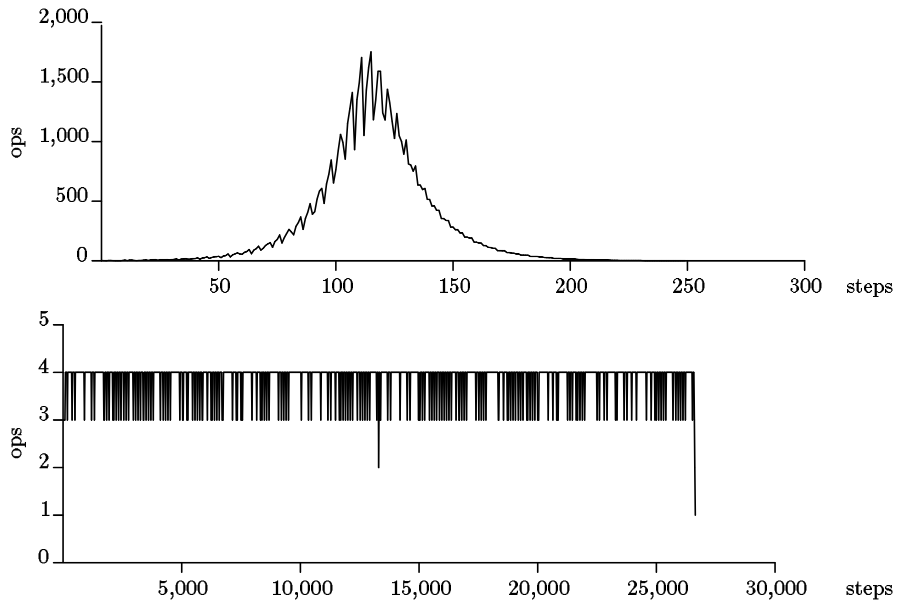


Figure 2: Parallelism profile for lenient and strict executions of `Leaves` function (tree of depth 10)

subsequently be filled using an assignment, at which point it becomes *full*. An I-structure component may only be assigned once. More formally, the component initially has the value $\perp$, which may then be refined to a value $v > \perp$.

An attempt to read an empty component simply suspends until a value is available. Formally, such a read-expression is simply $\perp$ until it gets a value $v > \perp$.

I-structures are useful for expressing certain computations for which we know no efficient purely functional solution. Consider the following problem:

> Given an array A of numbers, produce a new array B that represents a crude sort– B contains all the negative numbers from A followed by all the non-negative numbers from A.

We know of no functional solution to this problem that does not first create some other intermediate data structure as large as the arrays (try it!). Here is a solution using I-structures.

```
{ B = I_array (1,N);

  def loop  j  L  R  =  if (j > N) then
                            ()
                        else
                          { (L',R',k) = if (A[j] < 0) then (L+1,R,L)
                                                      else (L,R-1,R);
                            B[k] = A[j]
                          In
                            loop  (j+1)  L'  R' }

  loop  1  1  N;
In
  B }
```

The first statement creates the result I-structure array `B` of size N, with all slots empty. The second statement defines the function `loop` (Id allows function definitions inside blocks, with the usual static scope rules). The third statement invokes `loop`, and finally the array `B` is returned. The loop traverses the array `A` for `j` going from 1 to N. It carries with it two "cursors" `L` and `R` with which it fills the array `B`, from the left for negative numbers and from the right for non-negative numbers. At each `j`, it decides whether `A[j]` should be copied into the lower or upper part of `B` (the index `k`), and computes the updated value of the cursors for the loop call. The second statement in the body of the loop copies the `j`'th component of `A` into the chosen slot of `B`.

I-structure components may also be used in algebraic types. Our second example illustrates "open lists".

```
type olist *0  =  ONil | OCons  *0  .(olist *0);
```

The "." in the type declaration specifies that it has I-structure semantics, meaning that when the `OCons` constructor is applied, the second component may be left empty, as follows.

```
    OCons  e1  _
```

The component may subsequently be filled in, as follows:

```
    oc.OCons_2  =  e2
```

Pattern-matching works as usual on I-structure slots, so it is possible, for example, to write a mapping function on `olists` exactly like the functional `map` function in Section 3. However, it is possible to write a *tail-recursive* mapping function on `olists`, which was not possible in the functional case:

```
def  omap  f  ONil         =  Nil
|    omap  f  (OCons x xs) =  { ys           = OCons  (f x)  _ ;
                                ys'          = loop  f  xs  ys;
                                ys'.OCons_2  = ONil
                              In
                                ys };

def  loop  f  ONil          ys  =  ys
|    loop  f  (OCons x xs)  ys  =  { ys1         = OCons  (f x)  _ ;
                                     ys.OCons_2 = ys1
                                   In
                                     loop  f  xs  ys1 } ;
```

When omap finds a non-empty list, it constructs the first cell of the result list (`ys`), with an initially empty tail, and it returns `ys` as the result. In parallel, it invokes `loop`, which traverses the input list, growing the tail of the list `ys` with new elements. `loop` returns the last cell of the list, `ys'` (which may be the same as `ys`), which, by the invariant for `loop`, always has an empty tail. `Omap` plugs this last empty tail with `Onil`.

Discussion of I-structures

Admittedly, omap is by no means as pretty (or simple) as the functional `map`. Our point here is to demonstrate that I-structures often admit tail-recursive, constant-space solutions where the functional core does not. Interestingly, the distinction between functional data structures and I-structures is completely an artifact of the type system; there is no difference at all in implementation. In fact, in the implementation of Id's standard libraries (where type rules may be bent), the functional `map` is expressed in this tail recursive manner.

The "pipelined parallelism" illustrated in Figure 2 may also be seen in omap. Since all components of a block evaluate in parallel, omap can return `ys` as soon as it has been created– it does not have to wait for `loop` to complete and for the tail of `ys'` to be plugged. In particular, if `(omap f xs)` takes time t (which is $O(n)$, n

being the length of the list), then the composition (omap g (omap f xs)) can execute in time $t+\delta$, instead of $2t$, because the outer omap can begin work immediately on the initial prefix returned by the inner omap. In other words, the inner omap produces its output list continuously, like a stream, which is continuously consumed by the outer omap. There is no danger of the outer omap overtaking the inner, because in this case it will simply encounter an empty tail, and block until it is full. Thus, except for the fact that the outer omap must follow the inner by a small delay (the δ, above), the two omaps can operate in parallel.

I-structures also permit us to use a technique similar to one that is popular in logic programming languages, namely "difference lists". A difference list is a pair of references: one to the first cell of a list, and the other to the last cell, whose tail component is always empty. Figure 3 shows how appending two different lists can be done in constant time. This is accomplished using the following code:

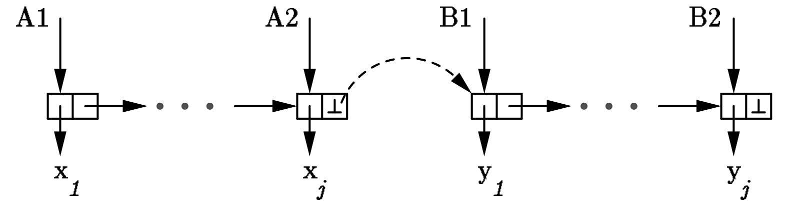


Figure 3: Appending two difference lists takes constant time

```
type Dlist *0 =  DNil | D2 (olist *0) (olist *0);

def  dappend  DNil          DNil         =  DNil
|    dappend  (D2 A1 A2)    DNil         =  D2 A1 A2
|    dappend  DNil          (D2 B1 B2)   =  D2 B1 B2
|    dappend  (D2 A1 A2)    (D2 B1 B2)   = { A2.OCons_2 = B1
                                             In
                                               D2 A1 B2 } ;
```

Readers familiar with parallel logic programming languages (such as KL1, [43]), will perhaps by now have seen a similarity between I-structures and logic variables and their use as a way to communicate data from a producer to multiple consumers. In fact, the connection is deep, and has been explored in detail by Jagadeesan and Pingali [21]. If I-structure assignment is viewed as unification, *i.e.,* a slot is refined to the *unifier* of its current contents and the value being assigned, then the correspondence is exact. With this viewpoint, multiple assignments into an I-structure slot are perfectly all right, provided all the assigned values are "consistent" with each other. An inconsistent assignment refines the value to $\top$, the inconsistent value.

For practical reasons, however, we take a more conservative approach, just like Prolog implementations which typically omit the "occurs" check during unification because it is so expensive. Apart from the occurs-check, there is the additional problem that in Id we have higher-order functions and abstract data types, for which unification is undecidable. Thus in Id, we take the position that multiple assignments into an I-structure slot always take the whole program to $\top$. The whole program must go to $\top$, and not just the multiply-assigned slot, in order to preserver determinacy. Intuitively, consider some other part of the program that reads that slot. If it reads before the second assignment, it would read a proper value, whereas if it reads after the second assignment, it would see the error. Determinacy is restored by insisting that the whole program goes to $\top$. This may seem a rather drastic step, but in practice it is not. Multiple assignments should be regarded in the same category as indexing an array out of bounds— it cannot be statically prevented, but it is usually possible to avoid it and, when it does happen, the error is always caught and reported.

5 The M-structure layer

As mentioned earlier, M-structures are unrestricted side effects that are are novel in the way data access is integrated with synchronization. An M-structure component of a data structure is either *empty* or *full* (in this sense they are similar to I-structures). Reading and writing an M-structure component are called *take* and *put* operations, respectively. A take operation suspends while the component is empty; when full, it returns the value, *leaving the component empty again.* A put operation is only allowed on an empty component, and it leaves it full. If there are multiple takes blocked on an empty component, only one of them succeeds when the component becomes full (due to some put operation); we do not specify which one succeeds. Thus, given an M-structure array A, the assignment:

```
A![3] = A![3] + 1
```

is an *atomic* increment of component 3. If there are multiple instances of this statement active simultaneously, only one of the *takes* will succeed, and the others will block until the incremented value is *put* back. The increments at location 3 will thus be serialized. Note, there is no serialization with respect to accesses at other indices (`A![4]`, `A![25]`, ...).

With M-structures, it is trivial to write a *parallel* program that computes a histogram of the numbers in binary tree, for example:

```
type  tree  =  Leaf int | Node tree tree;

def histogram t = { H = {m_array (1,N) of
                         | [j] = 0 || j <- 1 to N};

                    traverse  t  H
                  In
                    H };

def traverse  (Leaf j)    H  =  { H![j] = H![j] + 1  In () }
|   traverse  (Node L R)  H  =  { traverse  L  H;
                                  traverse  R  H
                                In
                                  () };
```

The `histogram` function creates an M-structure array for the histogram, initialized to 0. It calls `traverse` to accumulate the histogram, and finally returns H. The traverse function fans out exponentially (going down left and right branches of each interior node in parallel). At each leaf, it atomically increments the appropriate slot in the histogram.

It is of course possible to write a purely functional solution to this problem (or even one using I-structures). However, all such solutions typically use extra intermediate data structures which are difficult to eliminate in a parallel language. Here is one such functional solution:

```
def histogram t ={ H = {array (1,N) of
                         | [j] = 0 || j <- 1 to N};
                 In
                    traverse  t  H };

def traverse  (Leaf j)    H  =  incr_array_slot  H  j
|   traverse  (Node L R)  H  =  traverse  R  (traverse  L  H);
```

Here, `traverse` essentially performs a depth-first walk of the tree; at each leaf, `incr_array_slot` produces a new array with the `j`'th slot incremented. This solution is very storage-inefficient— it builds as many arrays as there are leaves. There are some compiler analyses that will recognize that the array is being used in a "single-threaded" fashion [9], which can be used to update the array *in situ*, but these analyses are limited– first, they are not very effective when dealing with higher-order functions and nested data structures, and more seriously, they typically assume a strict, sequential evaluation order, whereas our goal is to maximize parallelism.

An important point to note is that in the M-structure solution, the increments occur in an unpredictable order, *i.e.,* the sequence of intermediate histogram arrays is *non-deterministic*. However, the final histogram is, of course, unique, depending only on the set of integers in the tree. This is an illustration of a common feature of Id programs that use M-structures– local non-determinism is used to provide increased parallelism, in order to compute a determinate result. The disadvantage, of course, is that determinacy is now a property only of individual programs and not a property of the language.

Graph traversals

Another common use for M-structures is in graph traversals, which are notoriously hard to express efficiently in purely functional languages (I-structures do not help, either). Consider the following problem: given a node `root` of a directed graph (which may have shared subgraphs and cycles), count the nodes reachable from `root`.

A traditional imperative program would use a mark bit in each node and perform a depth-first search from `root`, retreating every time it encountered a previously marked node.

A traditional functional program would also do a depth-first search from `root`, but since it cannot use mark bits, it would carry along a table of visited nodes, retreating every time it encountered a node already in the table. This table has to be "threaded" through the traversal, making the program completely sequential.

It is possible to write a solution using M-structures that mimics the imperative solution with mark bits but, curiously, that is not the typical M-structure solution. In particular, it precludes multiple traversals of the graph in parallel, since only one traversal at a time can use the mark bits. Instead, the best M-structure solutions mimic the functional solution in carrying along a separate table of visited nodes; since each traversal has its own table, multiple traversals can occur concurrently without interfering with each other. However, it differs from the functional solution in that the table is not threaded into a sequential traversal; instead, nodes are entered into the table non-deterministically.

```
type  gnode  =  GNode  int  info  (list  gnode);

def  gcount  root  =  {  table = mk_empty_table ();
                      In
                         dfs  table  root };

def  dfs  table  (GNode  id  x  nbs)  =  if  (test_and_insert  id  table)  then
                                            0
                                         else
                                            1 + sum (map  (dfs  table)  nbs) ;
```

The `dfs` function performs the depth-first traversal. It tests if the current node has already been visited, retreating with value 0 if so. Otherwise, it counts 1 for this node, plus the count obtained by recursively invoking `dfs` on each of its neighbours `nbs`.

Note that the table is no longer threaded through the traversal. The `map` function invokes `dfs` *in parallel* on all the neighbors, passing `dfs` to all these invocations. The same table is accessed by all invocations of `dfs`. Thus, it is possible that `dfs` arrives at some shared node simultaneously *via* two separate paths. One of them will record the node in the table and continue, whereas the other, seeing the node in the table, will retreat. This is non-deterministic, but it does not matter, since the final count of graph nodes does not depend on this order– it depends only on the actual graph.

One subtlety is the `test_and_insert` function, which does two thing *atomically*— not only does it test whether the node identifier `id` is in the table, but it also inserts it in the table if not. This is important— if it were not atomic, two invocations of `dfs` on a node may simultaneously test and not find the node in the table; then, both would insert the node, count it, and perform the recursive traversal. By making it atomic, we ensure that only one of them succeeds.

For lack of space, we omit the implementation of the table and its operations; in [7], we show a parallel hash table that will serve.

This example illustrates both the disadvantages and the advantages of M-structures. To recognize that we needed an atomic `test_and_insert` rather than separate `test` and `insert` functions, is subtle, and may be regarded as an abstraction violation, since the implementor of tables may not have predicted this in isolation (fortunately, there are ways to create such atomic actions *ex post facto*, but we do not explore this further here). Further, it is once again up to the programmer to recognize that despite the local non-determinism, the resulting `gcount` function is still determinate.

On the other hand, the M-structure program is transparently parallel and space-efficient, which cannot be said for the functional solution. Further, we doubt that any compiler can be smart enough to produce such a parallel and space-efficeint solution from a functional program.

Curiously, this last observation contradicts conventional wisdom. Many people (including the author) have promoted functional languages as "naturally parallel" languages, claiming that imperative languages are unlikely to have compilers that are smart enough to uncover the parallelism that is obvious in functional programs. Here, on the other hand, it is the imperative (M-structures) program that has obvious parallelism, which will be hard to match by compilers for functional languages.

6 Lenient Evaluation Order, revisited

In Section 3 on the functional core, we introduced lenient evaluation in order to give the reader a model for parallelism in Id programs. Now, having covered I- and M-structures, we return to this topic to explore further subtleties.

For the functional core, lenient evaluation is simply a matter of parallelism and resource management. Lenient evaluation implements non-strictness, just like a lazy evaluator, but it gives us more parallelism. For those exploring lazy implementations, the goal is for automatic analysis, such as strictness analysis, to identify computations that may be evaluated eagerly in parallel. Strictness analysis is very hard in the presence of higher-order functions and non-strict data structures, so many researchers have resorted to programmer-specified annotations for eager, parallel evaluation. There is not much difference between lenient evaluation and programmer-annotations for eager evaluation in a lazy evaluator— in both cases, controlling termination may be up to the programmer. In Id, the programmer has explicit control in the opposite direction, *i.e.,* he can explicitly request lazy evaluation in order to program with infinite structures such as streams

A lenient evaluator, by reducing unnecessary redexes, may use extra computational resources (both processor and memory), but it is not obvious that this is more costly than a lazy evaluator. A lazy evaluator uses both processor and memory resources simply to suspend and reactivate computations; historically, these resource demands have been much heavier than implementations of strict languages, which perform unnecessary computations eagerly as in Id.

For the functional core, the evaluator still has some latitude in choosing redexes. In particular, except for the normal order redexes, it can safely ignore all other redexes. In fact, once the top-level term has been reduced to normal form, the evaluator can safely declare termination with that answer.

Once we introduce I- and M-structures, however, this latitude is no longer available. Now, the evaluator must reduce all and exactly all redexes that we specified earlier, namely, all redexes except those inside conditionals and lambdas. The reason is that once we have side-effecting constructs, we must be precise about exactly which side-effects will or will not occur during execution. Because of side-effects, there is no longer any way to decide which redexes are "needed" and which are not (in the functional core, the normal order redexes, and none other, are needed).

7 Implementation

Until 1989, implementation of Id was targeted almost exclusively towards dataflow machines. The first compiler was written by Ken Traub[2] [41], and was targeted to the MIT Tagged Token Dataflow Architecture [4], a machine that was studied extensively through simulation (the parallelism profiles in Figure 2 were gathered on a TTDA simulator). This compiler was later retargeted to the Monsoon dataflow machine [33], which was built as a collaborative effort between MIT and Motorola, Inc.

Starting in 1989, there were two separate, but similar projects to implement Id on stock hardware. At Berkeley, David Culler and his group retargeted the MIT Id dataflow compiler to TAM (Threaded Abstract Machine) which, in turn, was compiled to native code for stock hardware [38]. Simultaneously, the author began implementing a new Id compiler that abandoned the original dataflow basis in favor of *P-RISC Graphs*, which are fine-grain, parallel control-flow graphs with a model of locality [31, 30].

We now outline the implementation of Id used in the author's P-RISC compiler, which we expect to release by December 1993. Figure 4 shows a model of the runtime system. The runtime system takes a strong

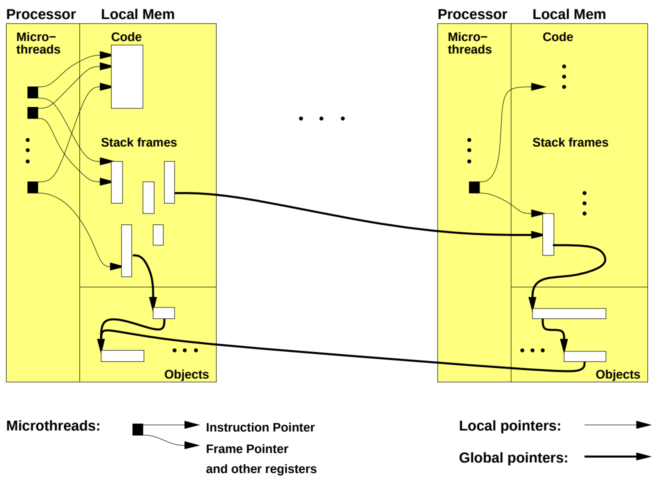


Figure 4: Runtime system of the P-RISC implementation of Id

position on locality by assuming a distributed memory model. Our initial implementations are for so-called "multicomputers" that directly fit this model, such as workstation farms and the CM-5. However, we believe that this approach is beneficial even if implemented on a shared-memory parallel architecture such as the KSR-1, DASH or Alewife, because modern shared-memory architectures typically have an underlying distributed memory system which involve similar locality concerns.

An Id program is compiled into fixed code that is available to all processors (replicated on multicomputers). As the computation unfolds, each invocation of a function or loop iteration gets its own *frame* for local variables. Because of parallel invocations, these frames logically form a tree, not a stack, although for familiarity we still refer to them as stack frames. Being part of a tree structure, a sequence of frames may not occupy contiguous memory like a stack; frames are allocated individually and linked by pointers.

Frames are the unit of coarse-grained parallelism, *i.e.,* the frames of a caller and a callee may be on different processors. However, a frame never crosses a processor boundary. Since the fixed code is available to all processors, a frame may be allocated anywhere. In each processor, code is always executed relative to a frame, and accesses to frame variables are always ordinary (local) loads and stores. The combination of an instruction pointer and frame pointer, with perhaps additional temporary registers, is called a *microthread*. Microthreads are always local to a frame– microthreads do not even cross frame boundaries, let alone

[2] Vinod Kathail had previously written a compiler for the pre-1985 predecessor language which was also called Id.

processor boundaries. Conceptually, a processor, and even a single frame, may have many microthreads executing simultaneously although, in practice, these are multiplexed.

P-RISC differs from traditional thread models, which are also based on tree-structured stacks ("cactus stacks"), primarily in its granularity and locality assumptions. Cactus stacks typically have many frames between branches, whereas any P-RISC frame may have multiple children. In cactus stacks, there is typically exactly one thread between branches, whereas in P-RISC even a single frame may have multiple microthreads. Cactus stack implementations typically are designed for a shared-memory model, so a stack (and its associated thread) may cross processor boundaries, whereas a P-RISC microthread is always local to a processor.

Objects are allocated on a segment of the global heap that is local to a processor. An object cannot cross a processor boundary. A source-level data structure may be larger than a local heap, but such abstractions must be implemented by the compiler using a heirarchy of smaller objects, each of which fits within a local heap.

Conceptually, microthreads communicate using messages. For example, during a function call, the caller issues one or more messages which trigger microthreads in the callee (the caller may continue doing other work). Similarly, the callee issues one or more result messages which trigger threads in the caller. For a heap access, one microthread issues a message that triggers a heap access handler microthread on the processor where the heap object resides. The handler issues a response message which triggers another microthread on the original processor. These are sometimes called *split-phase transactions.*

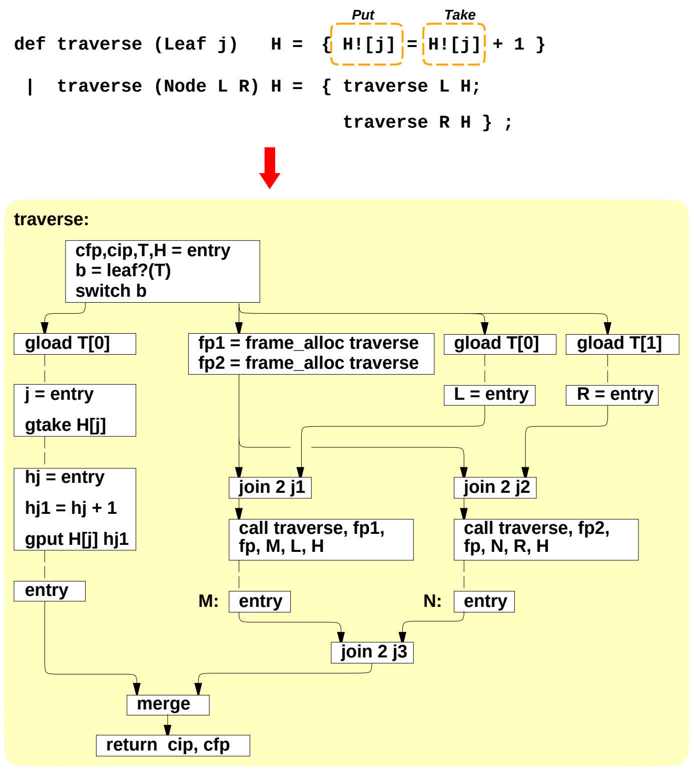
Thus, P-RISC microthreads never suspend. Even synchronization is handled as part of the split-phase model. Suppose a microthread on processor P1 issues a read request to an I-structure slot that resides on processor P2. The handler on P2 tests the slot and, if empty, queues the request on the slot. Later, when there is a write to the slot, the corresponding write handler dispatches the response to the waiting request. Thus, the synchronization is handled by the heap read handler on P2, not by the accessing microthreads on P1.

Id programs are first translated into P-RISC graphs, which are based on this runtime model. Figure 5 shows a translation of the histogram-building `traverse` function from Section 5. The `entry` instruction at the top receives a message, from which it loads four local variables. `T` and `H` are references to the tree and histogram array, respectively. The other two variables represent the *continuation* for this function (*i.e.,* return information), namely, a pointer `cfp` to the frame of the callee, and the label `cip` of a microthread that is expecting the function's result. After testing whether it is a leaf, the `switch` does a conditional jump. On the *true* side (on the left), the `(gload T[0])` requests a global load from the heap– the contents of leaf of the tree. The dashed line below it indicates that this is a split phase operation. The result, when it arrives, triggers the microthread below it, which loads the result from the message into the local variable `j`. The `(gtake H[j])` performs a *take* operation on the histogram slot, which is another split phase operation. When that response arrives, the incremented value is *put* back using `gput`.

On the *false* side, we fork off three microthreads which can execute concurrently. Thus, we allocate two frames for the recursive calls, and we perform the two heap loads for the left and right subtrees, in parallel. Each `join` instruction ensures that all its incoming threads have arrived before continuing below the join. This is a very cheap operation: the local variable `j1`, initially zero, is incremented and compared with the terminal count, "2". The first thread increments to 1 and dies. The second thread increments to two and continues. Note, the first thread simply dies– there is no suspension involved.

In each recursive call to `traverse`, note that we pass the current frame pointer `fp` and a return label (`M` or `N`), representing the return continuations. The calls occur in parallel because after issuing a message for one call, the processor can execute other microthreads, including one that issues a message for the other call.

The runtime state involves a collection of microthreads on each processor. Microthreads are initially triggered by the arrival of a message. A microthread always runs to completion, during which it may fork other microthreads and issue other messages. This is mapped to stock hardware as shown in Figure 6. Messages arrive at a processor in a central message queue. A message always begins with an instruction pointer and a global pointer (which refers either to a frame or to a heap object). The top-level action of a processor, thus, is to repeatedly dequeue a message and execute the corresponding microthread. As the microthread



```
def traverse (Leaf j)   H = { H![j] = H![j] + 1 }
 |  traverse (Node L R) H = { traverse L H;
                              traverse R H } ;
```

Figure 5: An example P-RISC graph

executes, any messages it issues are sent, and any microthreads it forks are simply pushed on the scheduling stack. When a microthread dies, it simply pops a microthread from the scheduling stack and executes it. We arrange it so that the microthread at the bottom of the scheduling stack always points at the code to dequeue the next message. Thus, microthread multiplexing is done very cheaply by the runtime system (there is no operating system involvement).

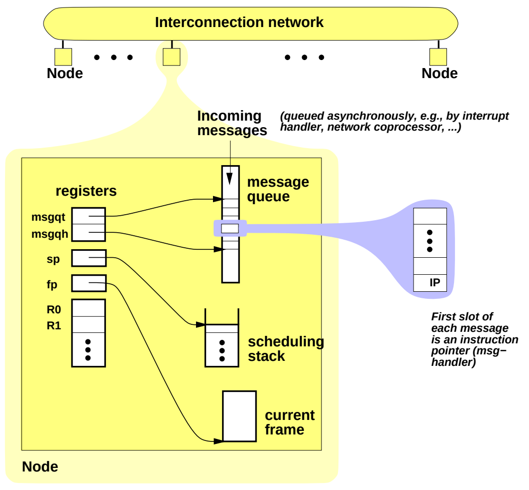

Interconnection network
Node
Node
Incoming messages
(queued asynchronously, e.g., by interrupt handler, network coprocessor, ...)
registers
msgqt
msgqh
sp
fp
R0
R1
message queue
IP
First slot of each message is an instruction pointer (msg-handler)
scheduling stack
current frame
Node

Figure 6: State in each node of a distributed memory multiprocessor

Allocation of frames and heap objects is done by calls to the runtime system, which currently uses simple round-robin strategies to distribute frames and objects amongst the processors (much work remains to be done in experimenting with different strategies). We also allow user-specified annotations in the source code to control frame and heap object placement, including co-location of objects and frames.

Our model of messaging is very lightweight. It assumes a minimal transport layer to actually deliver messages, with no further interpretation. Such a transport layer is often available directly in hardware. All message composition and message-handling is done entirely within the P-RISC runtime system. In addition, our compiler includes both static and dynamic optimizations that elide message-passing completely when the source and destination are known to be on the same processor.

Our description here merely gives the flavor of the implementation. For more detail, the reader is referred to [30].

8 Comparison with other parallel languages

In this section we compare Id only to other "shared memory" parallel programming languages because we believe that message-passing languages (such as Occam or CSP) are fundamentally not viable for general-purpose parallel programming. Message-passing languages require the programmer explicitly to partition work and data, which is difficult to do even once, let alone having to redo it for different machine configurations.

High Performance Fortran

HPF is a prominent recent development in parallel programming languages [16, 25]. It is a superset of Fortran 90, and Its design is based on experience with languages like Fortran D [17] and Vienna Fortran [44], and commercial offerings are expected from severral vendors.

HPF has sequential semantics, except for certain array statements and loops which may be regarded as a (large) collection of simultaneous assignments that may be executed in parallel. This "sequential" basis is, of course, very valuable in debugging. In HPF, the programmer specifies hints to the compiler regarding the distribution of data across processors, and the compiler determines not only the final data distribution but also all the work distribution (namely, which processors execute which iterations of a parallel loop). The data distribution hints have the advantage of being architecture-independent— the emphasis is more on relative placement and alignment of multiple arrays rather than absolute distributions.

Current compilation technology for HPF is only capable of handling static, "regular" parallel computations that fit nicely into the SPMD (Single Program Multiple Data) mold [17, 45]. In particular, parallelization is focused on large, static, dense rectangular arrays, for computations where the communication is very regular, such as nearest-neighbor shifts and exchanges, grid communications, stencil computations, *etc.*; we sometimes refer to this as *marching band* parallelism. Even matrix multiplication, whose communication pattern, while regular, does not fit easily in this mold, is typically beyond the scope of today's compilers (and hence is provided as a built-in primitive in HPF). There is some research work on more irregular computations involving sparse matrices (see, *e.g.,* the *inspector-executor* techniques of [37]), but the effectiveness and generality of these techniques are as yet unclear.

Thus, we believe that HPF is an excellent solution for a limited class of applications. We think that Id is more appropriate for applications outside this class, namely, dense matrix computations with irregular computation and communication patterns, sparse matrix computations, computations involving more complex data structures including trees, graphs and, in general, computation where the parallelism is not evident statically. Of course, for regular computations, there is nothing to prevent HPF-like compilation techniques to be applied to Id programs as well, but there is no major effort in this direction at present.

Parallel Functional Languages

Functional languages have long been advocated as naturally parallel languages, and Backus' influential Turing lecture [6] spurred much further work in this area. We share this enthusiasm, although we are not functional language purists. As explained earlier, we depart from pure functions for two reasons. The first reason is purely an expressive power issue: non-deterministic programs are sometimes shorter and easier to understand than their determinate counterparts which often *overspecify* the solution. The second reason is pragmatic: much research remains in order to make purely functional programs as parallel and storage-efficient as programs with I- and M-structures.

Researchers in strict functional languages have typically started with sequential semantics, adding annotations to introduce parallelism and synchronization. In Multilisp [13], sequential Scheme is extended with the construct `(future` e`)`, which is a hint to the system to evaluate e and the future's continuation in parallel. Synchronization is implicit in attempting to access the value of the expression, which blocks until e has terminated with a value. Qlisp [12] includes a parallel `let` construct in which all the right-hand sides of the bindings may be evaluated in parallel. CD-Scheme [35] has a `pcall` construct which allows all components of a function call to be evaluated in parallel, and a `remote` construct which allows a function call to be invoked on another processor. In contrast to these languages, Id does not have a sequential basis. In our experience, explicit annotations for parallelism are not only difficult to use, but the method is not modular– a good placement of parallelism annotations cannot be decided by looking at a function or module in isolation.

Concurrent ML [36] is based on SML, but departs further from SML's sequential core than Id, Multilisp or Qlisp in that it introduces new semantic objects specifically for parallelism, such as threads, channels and events. This reintroduces some of the difficulty of CSP-like message-passing programming languages. CML was not motivated by concerns of general-purpose parallel programming; rather, it was designed to express simpler and coarse-grained virtual concurrency, such as event loops in interactive, window-based systems.

Finally, the implementations of all the above languages are based on a coarse-grain, cactus stack model. This works well for shared memory multiprocessors of small size (*e.g.,* less than 20 processors), but it is not clear how to implement them and to exploit locality on large scale parallel machines. In contrast, the Id implementation technique addresses the locality problem in large machines directly, and its lenient evaluation order is the source of abundant fine-grain parallelism which is necessary in order to overlap computation with communication.

Except for Id, researchers in non-strict functional languages uniformly use lazy evaluation as the basis. The long-term goal is this: the sequentiality of lazy evaluation is relaxed by recognizing *strict* arguments to functions and constructors, and invoking them eagerly in parallel. Unfortunately, automatic recognition of strict arguments is still not practical in the presence of higher-order functions and nested data structures. So, implementations today leave it to the programmer to insert *spark* annotations [14] which are similar to `future` annotations in Multilisp. Spark annotations have the same defect as futures, namely that it is difficult to decide where to place them, and this decision is not modular. Finally, implementations of lazy functional languages are typically based on compiled graph reduction (see, *e.g.,* [34]), and it is not clear how to evolve them to run on distributed memory parallel machines.

Parallel Logic Programming Languages

Logic programming languages, like functional programming languages, have also long been regarded as naturally parallel languages. Unfortunately, it is not so easy to parallelize the implementation of a logic programming language because of the so-called shared environment problem: two OR-parallel branches may attempt to update (as part of unification) a variable in their shared environment in inconsistent ways, so that the environment, or a part of it, has to be copied. Thus, most researchers have chosen to limit their logic languages to elimintate this problem; in particular, they use so-called *committed choice* languages which restrict the OR choice to a small, deterministic conditional test. Further, some languages use mode declarations (programmer- or compiler-supplied) to specify the direction of data flow during unification, so that unification can be simplified substantially. An example is the language KL1 [43] that is used in the Japanese Fifth Generation project.

We believe that such restricted logic languages are, in fact, not very different from functional languages. Further, the simple, one-way data flow use of logic variables is very similar to the use of I-structures in Id. Thus, at an abstract level, these languages are close relatives of Id with I-structures. However, by staying within the functional framework, we are able to provide familiar facilities such as a polymorphic static type system, higher-order functions, arrays, *etc.* which are not usually found in logic languages. We do not know of any counterpart to M-structures in logic languages.

Parallel Object Oriented Languages

Although object-oriented (OO) languages originated in the sequential world (Simula, Smalltalk), there is a natural extension to parallelism that has attracted many researchers, namely that "message-sends" can be made non-blocking— a method can continue executing in one object after it has sent a message to another object, whose invoked method therefore executes concurrently [1]. Examples of these languages include Concurrent Smalltalk (CST) [18], Cantor [5], Acore [27], ABCL [40], CHARM [23] and COOL [10].

Many of the implementation concerns for these languages are almost identical to those for Id. So, the difference is mainly in the source language semantics and programming style, *i.e.,* the functional *vs.* object-oriented style. These differences are largely orthogonal to parallelism issues.

Our understanding of OO languages is still immature relative to functional languages, from both a theoretical and a practical standpoint: the semantics of inheritance and static types for OO languages is an active field of research, as is the search for efficent implementations of method-dispatch in the presence of inheritance. As these topics become better understood, it is possible that Id will incorporate some of these more expressive features of OO languages.

9 Results, and future work

To date, it has been difficult the compare Id performance with that of other parallel languages, because good Id implementations on stock hardware are not yet available. The most detailed performance study is described in [15], which presents a detailed comparison of four Id programs running on an 8 processor Monsoon dataflow machine [33] versus the corresponding Fortran and C programs running on a single MIPS R3000 RISC processor. Although Monsoon has only 8 processors, each processor runs 8 interleaved threads, so that it effectively runs a total of 64 simultaneous threads. Monsoon was built between 1989 and 1991 by MIT and Motorola, Inc. The four programs include matrix multiplication, Simple (a hydrodynamics kernel), Gamteb (a Monte Carlo photon particle simulation), and Paraffins, a symbolic program to enumerate all unique paraffin molecules upto a given size. The first two are regular, dense matrix oriented scientific computations, whereas the latter two are highly recursive and irregular.

First, the experiments confirm the viability of Id's implicit parallelism– exactly the same programs ran, *without rewriting or recompiling*, on 1 to 8 Monsoon processors, achieving speedups from 6.25 to 7.74. The programmer did not have to worry about data distribution, load balancing, communication patterns, or any such low-level concern.

Regarding absolute performance, the comparison is not straightforward. The Id programs running on one Monsoon processor executed from 1.5 to 3 times the number of cycles of corresponding Fortran and C programs running on a single MIPS R3000. The Id cycle count is likely to decrease by going to the P-RISC model described in Section 7, because the Monsoon architecture has a very poor register model, and is unable to exploit sequential threads when they are available. On the other hand, the Fortran/C cycle counts are likely to increase when the codes are parallelized. Our current research should throw some quantitative light on these trends.

We expect to release our first P-RISC-based implementation of Id, with source codes, by December 1993. It is already running in-house on several test programs on uniprocessors and clusters of workstations, but it is not yet ready for general use. This first release is not going to be a high-performance release, but we hope to interest others in joining the effort by giving them something to experiment with.

Parallel computing was once exclusively the domain of expensive supercomputers. However, parallel machines are now becoming affordable for the average university department. While the hardware is available, the software, particularly programming languages, lags far behind. It is our hope and goal that languages like Id (and its imminent reincarnation as pH) will demonstrate a feasible way to exploit parallel machines for routine computing.

Acknowledgements: The design and implementation of Id has been a collaborative work involving a number of researchers. The center of Id activity has been the Computation Structures Group at MIT, led by Professor Arvind, where the author worked from 1984 to 1991. Keshav Pingali (now at Cornell University) and Vinod Kathail (now at Hewlett-Packard Laboratories) contributed to the early design of Id, particularly I-structures. Ken Traub and James Hicks were the main compiler implementors. Shail Aditya contributed to Id's incremental type system and overloading extensions. Paul Barth contributed to the M-structure effort. Harry Hochheiser wrote large parts of the first mockup of the author's P-RISC compiler, and Derek Chiou implemented the backend for workstation clusters. K. Ekanadham from IBM Research contributed to Id language design. Professor David Culler and his group at University of California, Berkeley (including Klaus Schauser, Thorsten von Eicken and Seth Goldstein) implemented the TAM back end for Id, running on workstations, the nCUBE and the CM-5. Olaf Lubeck and Bob Hiromoto from Los Alamos National Lab, and Wim Bohm from Colorado State University have contributed to Id application programming and performance measurements.

References

[1] G. Agha. Concurrent Object-Oriented Programming. *Comm. of the ACM*, 33(9):125–141, September 1990.

[2] Z. M. Ariola. *An Algebraic Approach to the Compilation and Operational Semantics of Functional Languages with I-structures.* PhD thesis, Harvard University, June 1992.

[3] Arvind, K. P. Gostelow, and W. Plouffe. The (preliminary) Id report. Technical Report 114, Department of Information and Computer Science, University of California, Irvine, CA, 1978.

[4] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[5] W. C. Athas and C. L. Seitz. Cantor User Report; Version 2.0. Technical Report 5232:TR:86, Department of Computer Science, California Institute of Technology, Pasadena, CA 91125, January 1987.

[6] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Comm. of the ACM*, 21(8):613–641, August 1978.

[7] P. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proc. Functional Programming and Computer Architecture, Cambridge, MA, Springer Verlag LNCS 523*, pages 538–568, August 28-30 1991. Also: CSG Memo 327, March 1991, MIT Laboratory for Computer Science 545 Technology Square, Cambridge, MA 02139, USA.

[8] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, New York, 1988.

[9] A. Bloss. Update Analysis and Efficient Implementation of Functional Aggregates. In *Proc. Fourth Intl. Conf. on Functional Programming Languages and Computer Architecture, London*, pages 26–38, September 1989.

[10] R. Chandra, A. Gupta, and J. L. Hennessy. Integrating Concurrency and Data Abstraction in the COOL Parallel Programming Language. *IEEE Computer*, February 1994.

[11] W. Clinger and J. Rees (eds.). Revised^4 Report on the Algorithmic Language Scheme. Technical report, MIT AI Laboratory, November 2 1991.

[12] R. Goldman and R. P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, pages 51–59, July 1989.

[13] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[14] K. Hammond and S. P. Jones. Some Early Experiments on the GRIP Parallel Reducer. Technical report, Department of Computing Science, University of Glasgow, September 2 1990.

[15] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance Studies of the Monsoon Dataflow Processor. Technical Report CSG Memo 345-2, MIT Lab. for Computer Science, Computation Structures Group, October 12 1992.

[16] High Performance Fortran Forum. *High Performance Fortran: Language Specification, Version 1.0*, May 3 1993. Anonymous ftp: `titan.cs.rice.edu`.

[17] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D. *CACM*, 35(8):66–80, August 1992.

[18] W. Horwat, A. A. Chien, and W. J. Dally. Experience with CST: Programming and Implementation. In *Proc. ACM SIGPLAN 89 Conf. on Programming Language Design and Implentation*, 1989.

[19] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[20] G. Hutton. Higher-order Functions for Parsing. *J. Functional Programming*, 2(3), July 1992.

[21] R. Jagadeesan, P. Panangaden, and K. K. Pingali. A Fully Abstract Semantics for a Functional Language with Logic Variables. In *Proc. 4th. IEEE Symp. on Logic in Computer Science, Asilomar, CA*, June 5-8 1989.

[22] T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, (Springer-Verlag LNCS 274).*, February 1987.

[23] L. V. Kale. Parallel Programming with CHARM: An Overview. Technical Report 93-8, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1993.

[24] A. H. Karp. Programming for Parallelism. *IEEE Computer*, pages 43–57, May 1987.

[25] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1993.

[26] D. Luenberger. *Linear and Non-Linear Programming (2nd ed.).* Addison Wesley, MA, 1984.

[27] C. R. Manning. A Peek at Acore, an Actor Core Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 84–86. ACM Press, April 1989. Published in SIGPLAN Notices, 24:4.

[28] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 990.

[29] MPI Forum. Message Passing Interface Standard, 1993. `mpi-comm@cs.utk.edu`.

[30] R. S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. In *Proc. Sixth Ann. Wkshp. on Languages and Compilers for Parallel Computing, Portland, Oregon*, August 12-14 1993.

[31] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. *Intl. J. of High Speed Computing*, 5(2):171–223, 1993. Presented at Workshop on Massive Parallelism, Amalfi, Italy, October 1989.

[32] R. S. Nikhil and Arvind. Id/83s. Technical report, Computation Structures Group, MIT Lab. for Computer Science, Cambridge, MA 02139, USA, July 1985.

[33] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA*, May 1990.

[34] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-Machine. *J. Functional Programming*, 2(2), April 1992.

[35] C. Queinnec. A Concurrent and Distributed Extension of Scheme. In *Proc. PARLE 92*, 1992.

[36] J. H. Reppy. CML: A Higher-order Concurrent Language. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation, Toronto (SIGPLAN Notices 26:6)*, pages 293–305, June 26-28 1991.

[37] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and Run-time Compilation. *Concurrency: Practice and Experience*, 3(6), December 1991.

[38] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture, Cambridge, MA (Springer-Verlag LNCS 523)*, pages 50–72, August 1991.

[39] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[40] K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers. In *4th. ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 218–228, May 19-22 1993.

[41] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[42] K. R. Traub. *Implementation of Non-Strict Functional Programming Languages.* MIT Press (Research Monographs in Parallel and Distributed Computing), 1991.

[43] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer J.*, 33(6):494–500, 1990.

[44] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran– A Language Specification, Version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, University of Vienna, September 1992.

[45] H. Zima and B. M. Chapman. Automatic Restructuring for Parallel and Vector Computers. Technical Report ACPC/TR 90-4, Austrian Center for Parallel Computation, University of Vienna, 1990.