Parallel Destructive Updating in Strict Functional Languages

A.V.S. Sastry, William Clinger
Department of Computer Science
University of Oregon
Eugene, OR, 97403
[sastry,will]@cs.uoregon.edu

Abstract

In our recent paper [22], we gave an efficient interprocedural update analysis algorithm for strict functional languages with flat arrays and sequential evaluation. In this paper, we show that the same algorithm extends to a parallel functional language with additional constructs for partitioning and combining arrays. Even with parallel evaluation, the complexity of the analysis remains polynomial. The analysis has been implemented and the results show that several numerical algorithms such as direct and iterative methods for solving linear equations, LU, Cholesky, and QR factorizations, multigrid methods for solving partial differential equations, and non-numeric algorithms such as sorting can be implemented functionally with all updates made destructive. We describe a new array construct to specify a collection of updates on an array and show that problems like histogram, inverse permutation, and polynomial multiplication have efficient parallel functional solutions. Monolithic array operators can be considered as a special case of this new construct.

1 Introduction

Although pure functional programming languages show great promise for parallel programming, their success is limited by two problems. One is the array update problem: modification of an array at an index, also called *incremental update* [17], in general requires a new copy of the entire array. If all updates are made by copying, the complexity of an algorithm degrades in proportion to the size of the array being updated. This inefficiency is being addressed by three avenues of research: one approach requires the programmer to write programs in a restricted style that ensures that all updates can be perfomed by side effect; another approach, closely related to the first, requires the programmer to assert that it is safe to update an array by side effect, leaving it to the compiler to verify that assertion if necessary [16, 19, 30]; the *optimization* approach leaves it to the compiler to detect updates that can be implemented by side effect [4, 3, 6, 7, 8, 9, 12, 13, 15, 18, 22, 21, 23, 25, 24].

The second problem is: how to express parallel updates on an array? Can parallel updates on distinct indices be performed destructively? Can parallel updates on non-distinct indices be performed destructively?

Specifying a collection of updates using the incremental update operator results in a sequential solution. Monolithic arrays [1, 17] were devised to express parallelism at the expense of creation of a new array. Consider the operation of multiplying a row of a matrix by a scalar. With monolithic arrays, a new copy of the entire matrix is required. With our update analysis, the incremental updates yield a sequential solution with no space overhead. Ideally, we would like to update the matrix in parallel without copying the matrix.

In this paper, we show that incremental updates can be used to specify parallel updates. We also present an extension of our recent algorithm [22], which we believe to be the first practical algorithm for interprocedural update analysis in first-order functional languages with flat arrays and parallel evaluation.

To handle parallel updates on indices that are not known to be distinct, we devise a new incremental update operator called an *accumulating update* and show that problems like histogram, polynomial multiplication and inverse permutation [1, 17, 28] can be expressed naturally and implemented efficiently using update analysis.

2 A Parallel Functional Language

The incremental update operator does not lend itself well for expressing parallel updates on a single array. Consider updating an array `a` at indices `i` and `j` with values 3 and 4. If these two updates are performed in parallel, we get two new arrays each containing only one update. These updates must be non-destructive because the first argument of each update operator is live when the update is performed. Moreover, it is not clear how to incorporate both updates in a single array subsequently. The only way to express these two updates is to choose a sequential order of updates, for example `upd(upd(a,i,3),j,4)`. This criticism of incremental update operator has already been made in [1]. In this paper we show that by defining new operations on arrays, one can express parallel updates using the incremental update operator and update analysis can determine whether these updates can be made destructively.

Our language is a first-order functional language with flat multi-dimensional arrays. We introduce a `let` expression `let` $[t_1 = e_1, \ldots, t_n = e_n]$ `in` e `end`. The scope of a `let` binding $t_i = e_i$ is the entire `let` expression except itself or any region shadowed by a nested `let` binding. We also assume that all t_i's are distinct. Bindings with cyclic dependencies are not permitted. The `let` bindings and the body of the `let` expression can be evaluated in parallel subject to dependency constraints.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

2.1 Partition and Combine operations

A multi-dimensional array $a : [l_1 : b_1, \ldots, l_d : b_d]$ has dimension d and for an index $[i_1, \ldots, i_d]$ to be valid, it must be the case that $l_1 \leq i_1 < b_1, \ldots, l_d \leq i_d < b_d$.

Intuitively, partitioning an array means dividing an array into two subarrays with disjoint index spaces whose union is the index space of the original array. Combining, the inverse operation, is the concatenation of two arrays. A more precise description follows.

2.1.1 Semantics

The operator `partition` takes an array $a : [l_1 : b_1, \ldots, l_d : b_d]$, a dimension k such that $1 \leq k \leq d$, and a partitioning index i such that $l_k < i < b_k$, and returns two new arrays $a_1 : [l_1 : b_1, \ldots, l_k : i, \ldots, l_d : b_d]$ and $a_2 : [l_1 : b_1, \ldots, i : b_k, \ldots, l_d : b_d]$. The value of a_1 or a_2 at index $[i_1, \ldots, i_d]$ is the corresponding value of the array a. By this definition, indices of a_1 and a_2 are disjoint.

The `combine` operator is the inverse of `partition`. It takes two arrays $a_1 : [l_1 : b_1, \ldots, l_k : i_k, \ldots, l_d : b_d]$ and $a_2 : [h_1 : u_1, \ldots, h_k : u_k, \ldots, h_d : u_d]$ that are compatible for combining, and a dimension k such that $1 \leq k \leq d$, and returns a new array

$$a : [l_1 : b_1, \ldots, l_k : (i_k + (h_k - u_k + 1)), \ldots, l_d : b_d]$$

. The two arrays are compatible for combination if the size of a_1 in dimension i, which is $b_i - l_i + 1$, is same as $u_i - h_i + 1$ for $1 \leq i \neq k \leq d$.

The value of a at index $[i_1, \ldots, i_d]$ is $a_1[i_1, \ldots, i_d]$ if $[i_1, \ldots, i_d]$ lies in the range of a_1. Otherwise, it is

$$a_2[h_1 + (i_1 - b_1), \ldots, (h_d + (i_d - b_d))]$$

2.2 An Example

Consider the problem of adding two vectors. In an imperative language, vector addition can be performed by a simple do or `for` loop in $O(n)$ time. In a functional language without update analysis, the corresponding loop takes $O(n^2)$ time.

The `partition` and `combine` operators can be used to write more efficient functional programs. The following program runs in $O(n \log n)$ time without update analysis, in $O(n)$ time with update analysis, and in $O(\log n)$ time with update analysis and parallel execution.

```
\* // is integer division. *\
\* dim(a,i) = no. of elements in dimension i. *\
\* help_vector_add(a,b,i) adds vectors *\
\* a and b from index i onwards. *\

vector_add (a,b) = help_vector_add(a,b,0);

help_vector_add (a,b,i) =
  if dim(a,1) = 1
    then upd(a,i,a[i]+b[i])
  else let midpoint = i + dim(v,1)//2;
           a_1, a_2 = partition (a, midpoint, 1);
           rv_1 = help_vector_add(a_1,b,i);
           rv_2 = help_vector_add(a_2,b,midpoint)
       in
           combine(rv_1,rv_2,1)
       end
  endif;
```

If the size of a is larger than 1, then a is partitioned into two vectors a_1 and a_2. The problem is solved recursively on each vector and the solutions are combined using the `combine` operator.

2.3 Implementation Choices

There are two choices of implementation for the `partition` and `combine` operators. In a copying implementation, the two partitions are created by copying data from the original array. In a sharing implementation, the new arrays share data with the original array. This can be achieved by creating new array headers with the new ranges for the two partitions while sharing the data with the original array. The overhead of creating a header is $O(d)$, proportional to the dimension of the array which is usually a small number.

The `combine` operator can also be implemented by copying. However, if arrays that are combined are adjacent partitions implemented by sharing, then `combine` can also be implemented by sharing. In such a case, it is required to create a new header for the resulting array. The `combine` operator cannot always be implemented by sharing even if `partition` is implemented by sharing. The reason is that one may be combining arrays that are not physically adjacent to one another.

Suppose we have established that all `partition` and `combine` operators in a program can be implemented by sharing. Can we also avoid creation of array headers thus making `partition` an identity operator and `combine` a synchronizing operator? The array headers are used for bounds checking and operations like determining the size of the array. If bounds checking is not performed and the programmer does not use any operation that needs the array header, a situation similar to programming in a language like C, then `partition` and `combine` become operators that return their first argument. The partition operation can even be performed at compile time.

3 Update Analysis

This section is very similar to our previous paper [22]. One of the key insights that led to our simple algorithm was that anonymous aggregates (new aggregates) need not be tracked. We defined four analyses called *propagation analysis*, *aliasing analysis*, *selects-updates analysis*, and *reference count analysis*. We have to define the flow equations for `partition` and `combine` operators for these analyses. We assume copying semantics for these operators. Since we have extended the source language with `let` expressions, we have to extend the analysis for them.

Since `partition` and `combine` create new aggregates, they do not propagate any of their arguments. Including the `let` expression in the source language allows the user to name an expression and use the name elsewhere one or more times, causing sharing. Consider a `let` binding $t_i = e_i$. If e_i returns an anonymous aggregate, it can be shared in the rest of the `let` expression through the name t_i. Therefore, an anonymous aggregate can always be identified with the name of the temporary variable to which it is bound. When an anonymous aggregate is returned as a result of the function, any local name that was associated with the result can

be discarded because the scope of the local name is limited to the `let` expression in which it is introduced.

In our previous analysis, we used the emptyset to denote an anonymous aggregate as well as a non-aggregate value. In this paper, we introduce a new value b to represent a non-aggregate value. The emptyset denotes an anonymous aggregate. The reason for this change is that it reduces the complexity of the algorithm.

Aliasing analysis is exactly the same as described in section 4 of [22]. The *selects-updates* analysis is also the same except that `partition` and `combine` neither select nor update any of their arguments.

The dependence graph of a `let` expression is updated by adding an edge between node t_i and t_j if t_j is not a predecessor of t_i and t_i updates any array that is selected by t_j. After adding these edges, the dependence graph remains a directed acyclic graph. In our previous paper, we added edges without checking for the existence of a path between the two nodes, which could result in a graph with cycles. The dependence graph is used to compute the set of syntactically live variables at any binding. Our current decision is to sacrifice parallelism in favor of destructive updating, whenever there is a conflict between the two.

3.1 Computing Live Variables

Given a dependence graph where each node represents an expression (and the corresponding temporary variable), we have to determine the variables that are live at each node. A variable x is live at t_i if x is neither t_i nor its successor and there exists another node $t_j \neq t_i$ that is not yet evaluated and uses x. The reason we don't need to consider t_i or its successors is that these variables are not defined before the evaluation of t_i. Since we are considering parallel evaluation, we conservatively assume that all the nodes that are not predecessors of t_i are yet to be evaluated.

To compute live variables, we need to know the intermediate form of our language. The source and intermediate forms are described in figures 2.a and 2.b. The `partition` operator in the source language returns two values, the left and right partitions. We will associate a unique label with each `partition` operator. In the intermediate form, we split a `partition` operator into two `left_part` and `right_part` operators which inherit the label from the corresponding `partition` operator.

Since `partition` is a single operator at the source level, the variables used in `left_part` operator with label `1` need not be considered as live at the node `right_part` with the same label. In general, variables in nodes with the same label as that of t_i need not be considered in determining the live variables at t_i. Therefore the set of live variables at t_i is given as

$$\begin{aligned} Live(t_i) \;=\; \bigcup \{ x \mid\; & x \in Vars(t_j), \\ & t_j \notin preds(t_i), \\ & x \notin (succs(t_i) \cup \{t_i\}), \\ & label(t_i) \neq label(t_j) \} \end{aligned}$$

Consider the intermediate form for the helper function in vector addition example.

```
help_vector_add(a,b,i) =
  let t_1 = dim(a,1);
      t_2 = (t_1 = 1);
      t_3 = if t_1 then
              let t_4 = a[i];
                  t_5 = b[i];
                  t_6 = t_4 + t_5;
                  t_7 = upd(a,i,t_6)
              in
                 t_7
              end
            else
              let  t_8  = t_1//2;
                   mid  = i + t_8;
                   a_1  = left_part (a,mid,1);
                   a_2  = right_part(a,mid,1);
                   rv_1 = help_vector_add(a_1,b,i);
                   rv_2 = help_vector_add(a_2,b,mid);
                   t_10 = combine(rv_1,rv_2,1)
              in
                 t_10
              end
  in
     t_3
  end
```

The live variables at a_1 and a_2 are {b,i,a_2} and {b,i,a_1} respectively. The domains and functions for propagation analysis are given in figures 3.a, 3.b, 3.c, and 3.d. Details of the rest of the analyses have been omitted as they are very similar to the ones in [22].

3.2 Complexity

We show that the complexity of the analysis remains polynomial as in our earlier algorithm. The parameters are m (the number of internal nodes in the parse tree of a program, k (the maximum function arity), n (the number of functions), and p (the maximum number of operators that return anonymous aggregates). The parameter p includes the `partition`, `combine`, and upd operators and all the function calls that return new aggregates.

Since temporary variables are discarded when considering the value propagated by any function, the number of fixpoint iterations needed for all the analyses are the same as in [22]. Recall that propagation analysis requires $O(nk)$ iterations. The maximum number of values that can be propagated by an expression is $k + p + 1$; 1 represents the non-aggregate value, although it will be a type-error to return an aggregate as well as non-aggregate value. Analyzing a function call, the most expensive operation, requires at most k unions of sets of size at most $k + p + 1$. The overall complexity of propagation analysis becomes $O(mnk^2(k+p))$. Recall that the worst case complexity of propagation analysis in [22] is $O(mnk^3)$. If we had not distinguished between non-aggregate values and anonymous aggregates, then the maximum size of a set would have been $O(m)$ instead of $O(k + p)$.

Similarly it can be shown that the worst case complexity of *aliasing analysis* is $O(mnk^4(k + p)$, although in practice it takes only a few iterations. *Selects-updates analysis* and *reference count analysis* take $O(mnk^2(k + p))$ time. For reference count analysis, we assume that the live variables are already computed as discussed in previous subsection. In the above estimates, we haven't included the complexity

of adding edges to the dependence graph and computing the live variables both of which can be shown to be of polynomial complexity.

The main intent of the complexity estimate is to show that the analysis runs in polynomial time, even with parallel evaluation. For typical cases, these worst case estimates do not reflect the actual running times. Our previous analysis, for example, runs in near linear time on typical programs.

3.3 Optimizing Combines

Suppose after the update analysis we discover that a `partition` operator can be converted into a non-copying `partition!`, as is the case if its first argument is not live. The question is: can we always convert a `combine` to a non-copying `combine`? The answer is no. Consider the function

```
f (x,y) = combine(x,y,1)
```

Since we don't know anything about the storage layout of `x` and `y`, `combine` cannot be known to be non-copying at compile time.

We can perform a simple analysis to detect if `combine` can be non-copying. We use an analysis similar to propagation analysis that determines addresses propagated by an expression. The flat domain of addresses is defined as

$$\begin{aligned} Add &= ((V + \{\top\}) \times P^*)^{\top}_{\bot} \\ P &= Part \times \{l, r\} \\ Part &= \{\text{Occurrences of Partition Operators}\} \end{aligned}$$

$\top$ represents an unknown address. An address $\langle v, s \rangle$ is the address of an array obtained by applying a sequence of s `left_part` or `right_part` operations on the array with address v. We define how the labelled primitive operators propagate these addresses:

$$\begin{aligned} \mathcal{K}[\![\texttt{upd!}]\!]\ x &= x \\ \mathcal{K}[\![\texttt{upd}]\!]\ x &= \top \\ \mathcal{K}[\![^a\texttt{left\_part}]\!]\ x &= \top \\ \mathcal{K}[\![^a\texttt{right\_part}]\!]\ x &= \top \\ \mathcal{K}[\![^a\texttt{left\_part!}]\!]\ x &= \textit{if } (x = \top) \\ &\quad \textit{then } \langle \top, a.l \rangle \\ &\quad \textit{else } \langle fst(x),\ snd(x).a.l \rangle \\ \mathcal{K}[\![^a\texttt{right\_part!}]\!]\ x &= \textit{if } (x = \top) \\ &\quad \textit{then } \langle \top,\ a.r \rangle \\ &\quad \textit{else } \langle fst(x),\ snd(x).a.r \rangle \\ \mathcal{K}[\![\texttt{combine}]\!]\ x\ y &= \textit{if } (x = \top \textit{ or } y = \top) \\ &\quad \textit{then } \top \\ &\quad \textit{else if } (fst(x) = fst(y), \\ &\qquad (snd(x) = s.b.l), \\ &\qquad (snd(y) = s.b.r) \\ &\quad \textit{then } \langle fst(x),\ s \rangle \\ &\quad \textit{else } \top. \end{aligned}$$

All operators are bottom strict. The functions *fst* and *snd* are projection functions. All operators except `combine` that return new aggregates return $\top$ as the address. We also assume that arithmetic operators return $\top$. The operators with a `!` are non-copying. If the two arguments to `combine` are addresses of the left and right partitions created by a single partition operator whose label is b then the result is obtained by removing the last two elements in the sequence of the first argument to `combine`. In all other cases, the result is $\top$. The interprocedural analysis for address propagation can be defined using the function $\mathcal{K}$. If a function returns an address $\langle v, s \rangle$ where s is non-empty, it is replaced by $\top$. In other words, information about the partition of an array created inside a function and returned as its result is forgotten outside the function as shown by the example below.

```
f x =
     let t1,t2 = partition(x,1,1)
     in
        t1
     end
```

Variable `t1` gets the value $\langle x, 1.l \rangle$ where 1 is the label of `partition`). Since the result is of the form $\langle x, s \rangle$ where s is not the empty sequence, it is immediately changed to $\top$. The address propagation function for `f` is $f\ x = \top$.

Now consider the helper function for vector addition described in section 2. After update analysis, the function is

```
help_vector_add (A,b,i) =
  if dim(A,1) = 1 then upd!(A,i,A[i]+B[i])
  else
    let midpoint = i + dim(a.1)//2;
        a_1 = left-part!(a,midpoint,1);
        a_2 = right-part!(a,midpoint,1);
        rv_1 = help_vector_add a_1 b i;
        rv_2 = help_vector_add a_2 b midpoint
    in
       combine(rv_1, rv_2,1)
    end
  endif;
```

For the purposes of address propagation, the flow equation is

$$\begin{aligned} vadd(a, B, i) &= a \sqcup \\ &\quad \mathcal{K}[\![\texttt{combine}]\!]\ vadd(\langle fst(a),\ snd(a).1.l \rangle, B, i) \\ &\qquad vadd(\langle fst(a),\ snd(a).1.r \rangle, B, \top)) \end{aligned}$$

The equation can be solved by fixpoint computation as

$$\begin{aligned} vadd^0(a, B, i) &= \bot \\ vadd^1(a, B, i) &= a \\ vadd^2(a, B, i) &= a \sqcup \\ &\quad \mathcal{K}[\![\texttt{combine}]\!]\ \langle fst(a),\ snd(a).1.l \rangle \\ &\qquad \langle fst(a),\ snd(a).1.r \rangle \\ &= a \end{aligned}$$

If any of the arguments to a `combine` operator is $\top$ or of the form $\langle x, s_1 \rangle$ and $\langle y, s_2 \rangle$ and either $x \neq y$ or s_1 and s_2 are not of the form $s.b.l$ and $s.b.r$ respectively then we cannot make that `combine` non-copying. The fixpoint can be computed in linear time (at most 2n iterations where n is the number of functions).

Interaction with update analysis

Given that a `combine` operator could be made non-copying, we can make it actually non-copying, provided the two arguments to combine are not live. We need this condition because our update analysis assumes that combine always returns a new array that is not live elsewhere.

Even if a `combine` is not known to be non-copying at compile time, it is not always the case that it requires copying. A simple test at run-time can check if the two arguments are actually contiguous and then avoid copying. The real advantage of a statically non-copying combine operation is that in the absence of bounds checking and operators that access the array header, both partition and combine operations can be converted into identity operations, thus avoiding the overheads of header creation.

4 Experimental Results

We have designed and implemented a compiler that takes a source program of our language and generates a Scheme or C program. The main phases of the compiler are alpha renaming, cycle detection among `let` bindings, flattening of `let` expressions and conversion to the intermediate form, common subexpression elimination, and update analysis. The compiler is implemented in Standard ML.

Our example programs are mostly taken from numerical computations [11, 10]. These examples can be classified as direct methods for solving linear equations, iterative methods, and miscellaneous. Direct methods include gaussian elimination with and without partial pivoting, and various matrix factorizations such as LU, QR, and Cholesky. Examples of iterative methods are point jacobi, red-black method, successive over-relaxation, conjugate gradient method and the multigrid method for solving partial differential equations numerically.

The miscellaneous examples include basic operations such as vector addition, matrix multiplication, prefix computation, and quicksort.

Table 1 shows the effectiveness of our algorithm. In these examples, all updates are made destructive. The program size is characterized by m: the number of nodes in the parse tree, n: the number of functions in the program, and k: the maximum function arity. The last column of the table shows the analysis time on a SPARCstation IPC. It does not include the time for the other phases of the compiler.

5 On Programming Style and Predictability

We have demonstrated that our update analysis algorithm is effective and efficient. We now address the question of whether programmers will be able to understand the update optimization well enough to write efficient code.

This is an important question because update analysis is a powerful optimization that can easily change the complexity of an algorithm by orders of magnitude. Programmers need to know whether the code they write is efficient. It would be disastrous for programmers to write functional programs that they mistakenly believe will be made efficient by update analysis.

For this reason we believe that functional languages should be equipped with two kinds of update operator: a copying update and a destructive update. These operators would both have the purely functional semantics of the copying update operator, but their pragmatics would differ: The compiler would refuse to accept any program that contains a destructive update that our update analysis algorithm cannot prove to be equivalent to a copying update. The efficiency of an update operation would then be clear to programmers: A destructive update executes in constant time, but a copying update must be assumed to be inefficient until proved otherwise (by changing it to a destructive update and passing it through the compiler).

Programmers would find this very frustrating if the compiler were unable to accept destructive updates that the programmer knows are safe, but our update analysis is so effective that this would hardly ever happen.

Programmers would also be frustrated if they were unable to understand why the compiler rejects a destructive update. We know, however, that programmers will be able to understand the outcome of update optimization because they are able to understand a similar but more difficult issue—not perfectly, but well enough: the problem of dropping all pointers to a data structure so it can be garbage collected. The garbage collection problem is dynamic, and it involves dropping all pointers, whereas the update problem is static and involves dropping all but one pointer; otherwise these two problems are the same.

Programmers do struggle with the garbage collection problem, and not always successfully, but they do well enough. The penalty for failing to drop all pointers to a structure is that the program is less efficient than it should be, and may catastrophically run out of space when it shouldn't. The penalty for failing to drop all but one pointer to an update structure is that the update operator will have to be changed to a copying update operator, and the program will be less efficient than it should be. At least the programmer will know the program is inefficient, which is not always true with the garbage collection problem.

Furthermore the compiler can *explain* why it thinks a destructive update is unsafe. If aliasing is the problem, then the compiler can report the variables that it fears may be aliased. If two expressions update the same array, the compiler will detect the problem while adding precedence edges to the dependence graph. Again, the compiler can indicate the expressions that interfere.

Our analysis is independent of the choice of order of evaluation, so long as there exists any order of evaluation for which the compiler can prove that all destructive updates are safe. Therefore the compiler would not be sensitive to the order in which formal parameters are declared. (This is a distinct improvement over previous algorithms [4, 18].)

The `partition` and `combine` operators can be used efficiently by following a few simple rules. When an array is partitioned, for example, it should not be live elsewhere. Neither partition of an array should be returned as a result of a function. Every function should have a matching number of `partition` and `combine` operators. The left and right arrays that result from a `partition` operation should be the left and right arguments to a subsequent `combine` operator within the same function body, and the proof of this should be obvious to the programmer (so it will also be obvious to the compiler). If these simple rules are followed, then most unnecessary copying can be avoided.

Sometimes it is possible to reduce copying in one part of a program by introducing copying in another part. This is very hard for the compiler to notice, but easy for the compiler to confirm once it is pointed out. We are led therefore

Program	Size (m,n,k)	No. of upds	Destructive upds	Analysis time (in secs)
LU	(78,3,7)	2	2	0.59
cholesky	(86,3,7)	4	4	0.71
QR	(181,13,7)	6	6	1.49
gauss	(126,6,7)	8	8	1.24
gauss (with pivoting)	(154,10,7)	12	12	1.91
jacobi	(32,3,7)	1	1	0.29
red-black	(68,6,8)	4	4	0.56
conjugate gradient	(84,9,6)	2	2	0.61
SOR	(78,6,9)	4	4	0.74
multigrid	(163,9,6)	4	4	0.86
matmul	(54,2,7)	1	1	0.41
prefix sum	(26,2,3)	1	1	0.12
quicksort	(42,5,4)	4	4	0.21
vadd	(14,1,3)	1	1	0.05
ram simulator	(59,1,6)	7	7	0.33

Figure 1: Performance of the Update Analysis Algorithm

to propose an explicit copy operation that has the semantics of the identity function but serves also as a declaration. Explicit copy operations declare not only the programmer's awareness that copying will be required, but they also declare the places in the program where the programmer believes copying should occur in order to obtain the most efficient results.

Since the copying update operation that we took as the starting point for our research is equivalent to a composition of the destructive update and copy operators, and the copy operator is more versatile than the copying update operator, we refine our proposal by suggesting that functional languages should replace the copying incremental update operator by the combination of an explicit copy operator and an explicitly destructive update operator—both of which, we hasten to add, have a purely functional semantics provided the compiler refuses to accept any destructive updates that cannot be proved to be equivalent to the traditional copying update.

It may fairly be said that we are advocating a more imperative approach to functional programming. We believe this is consistent with other recent research into the problem of state in functional languages. We suggest that recent research may even be leading toward a rejuvenating redesign of imperative languages from a functional perspective, which would not be a bad thing at all.

For predictability and portability, compilers should behave uniformly. Update optimization could be implemented just as uniformly across compilers as tail-recursion optimization and type-checking. Our update analysis is no more complicated than type-checking in ML, and we believe programmers will find update analysis at least as easy to understand as ML style type-checking.

6 Expressing Collection of Updates

In this section, we define a new operation on arrays to express a collection of updates. This operation has not yet been incorporated in our language. The partition operation is useful for expressing parallel updates when it is known at compile time that the updates are on distinct indices of the array. Our experience has been that for several numerical algorithms, the `partition` operator suffices for expressing parallelism. However, there are cases when either the updates are not known to be distinct at compile time or there are multiple updates at the same index. The histogram and polynomial multiplication problems require updating at the same index. For the inverse permutation problem, the updates are performed on distinct indices but this is not known at compile time. How does one express such multiple updates without losing deterministic behavior?

We define a new update operator called an *accumulating update*. Given a suitable operator $\oplus$, the corresponding accumulating update is written as

$$A\{i \oplus\!= v\}$$

. It returns a new array like A except that at index i its value is $A[i] \oplus v$. Analogously, one can also define another operator

$$A\{i =\!\oplus v\}$$

for which the new array has a value $v \oplus A[i]$ at index i. Using the accumulating update operator, now we can specify a collection of updates on an array as

$$A\{\langle e_{index}\rangle \oplus\!= \ e \mid e_1 \leq i \leq e_2\}$$

The index expression e_{index} and e can have i as a free variable. This expression describes a set of updates one for each value of i from e_1 to e_n. There can be more than one generators. We need to require a certain property of the update operator to ensure a deterministic result. Consider two updates v_1 and v_2 at index i. In order for the result to be the same at the end of the updates, we require that

$$A\{i \oplus\!= v_1\}\{i \oplus\!= v_2\} = A\{i \oplus\!= v_2\}\{i \oplus\!= v_1\}$$

. In other words, $(A[i] \oplus v_1) \oplus v_2 = (A[i] \oplus v_2) \oplus v_1$. Thus we require that $\oplus$ obey the following identity.

$$(a \oplus b) \oplus c = (a \oplus c) \oplus b$$

Any operator that is associative and commutative satisfies the above property.

Since the order in which the updates are performed does not matter as long as they are serialized, multiple updates

can be implemented on a shared memory multiprocessor using an extra array of locks. The overall space complexity is $O(n)$ where n is the size of the array.

An optimization useful for implementing a collection of updates is to avoid locks whenever it can be determined that the updates are all disjoint. One simple case that occurs very commonly is with updates of the form

$$A\{\langle i\rangle \oplus= \ e \mid l \leq i \leq b\}$$

We know by the nature of the generator that all values of i are distinct. Therefore all updates are on disjoint indices and no synchronization is required. In such a case we can even replace $\oplus=$ by $=$ if we know that the initial array contains the identity element of the operator. For example, vector addition can be written as

```
vadd(A,B) =  A{<i> += B[i] | 0<=i< dim(A,1)}
```

If A is not known to be live elsewhere, then it can be updated destructively in parallel. Compare this program with the program using partition and combine. Currently we are investigating the issues involved in the compilation of the collection of updates operator.

6.1 Examples

Given an array of n numbers ranging from 0 to $m-1$, the histogram problem is to compute the number of occurrences of each element in the array. A one line solution using a collection of updates is

```
hist(A,n,m) = array(m,0){<A[i]> += 1 | 0<=i<n }
```

Polynomial multiplication can be expressed naturally by a collection of updates with + as the accumulating operator.

```
pmult(A,B,m,n) =
  let a = array(m+n,0);
  in
    a{<i+j> += A[i]*B[j] | 0<=i< m,0<=j<n}
  end
```

The inverse permutation problem takes an array I that holds a permutation of 0 to n-1 and returns an array A such that A[I[i]] = i. The difficulty is that it is not known at compile time if I is a permutation. We define an operator $*$ and an identity element e such that $x * e = e * x = x$ and $x * y = n$ otherwise. The inverse permutation problem can then be written as

```
inv_perm(I,n) = array(n,e){<I[i]> *= i | 0<=i<n}
```

A monolithic array is a array all of whose elements are defined once [20]. A monolithic array construct takes f and n as arguments and returns a new array whose value at i is $f(i)$. It can be described by using the collection of updates as

```
marray(f,n) = array(n,0){<i> = f i | 0<=i<n }
```

We do not need any accumulating operator, because from the syntax we know that all the updates are on disjoint indices. One can also write functions to compute scan primitives such as prefix sum, array compaction, copying, enumerate, and distribute-sums used in data-parallel computing[2].

7 Related Research

There hasn't been much work on update analysis in parallel functional languages. The only reported work is that of Gopinath [13, 14] and SISAL [6, 5]. SISAL does not handle recursion. Gopinath's analysis has a worst case exponential complexity. Our analysis is simplified by the partition and combine operations. P. Wadler has proposed monads as an approach to destructive updating [29]. Monads sequentialize the execution to achieve destructive updating and therefore are not suitable for parallel execution. We have taken an orthogonal approach: instead of sequentializing all updates, we divide the array into semantically different arrays by the partition operator allowing the updates to be done in parallel. Guzman [15] and Swarup *et al* [27] also assume sequential evaluation.

Monolithic arrays were proposed because of the difficulty of expressing parallel updates [1, 17]. In this paper, we give a generalization of the incremental update to express a collection of updates on an array. Monolithic arrays are a special case of this operator. Wadler's new monolithic array construct [28] needs additional data structures for performing combining operations, whereas in our approach combining is done at the array itself. Another relevant work in the context of specifying a collection of operations is the xapping data structure of Connection Machine Lisp [26], which is based on the SIMD model of computing. The programming language Id [20], a non-strict language, provides accumulators as an extension of arrays. An accumulator is allocated as a new array with initial values and all accumulations are performed atomically by an accumulating operator. The accumulators of Id appear to have been derived from the monolithic array operator, whereas we have generalized the incremental update operator.

8 Conclusions

We have presented a strict functional language with incremental updates, partitioning and combining operations, and a collection of updates for expressing parallelism. We have described an efficient update analysis algorithm and its performance on typical numerical algorithms. Currently we are working on the efficient compilation of our language for a multiprocessor system.

We have considered the implications of our algorithm for language design, and have explained why we believe programmers will be able to write efficient programs that rely on update optimization.

Acknowledgements

We thank the LFP program committee for their useful comments and suggestions on an earlier draft of this paper. This research was supported in part by the University of Oregon Doctoral Research Fellowship. The first author would also like to thank Dr. Sanjay Rajopadhye for discussions of monolithic arrays.

References

[1] Arvind, R. Nikhil, and K. Pingali. I-structures : Data structures for parallel computing. In *Graph Reduction*, pages 336–369. Springer-Verlag, LNCS 279, 1986.

c	$\in$	*Cons*		Constants
x	$\in$	V		Variables
op	$\in$	*Prims*		Primitive operators (*i.e.* +, -, sel, upd, partition, combine, array,...)
f	$\in$	F		Defined functions
e	$\in$	*Exp*	::=	$c \mid x \mid op(e_1, \cdots, e_n)$ $\mid f(e_1, \cdots, e_n)$ $\mid$ if e_0 then e_1 else e_2 $\mid$ let *binds* in exp end
binds	$\in$	*Bindings*	::=	$(Ids = exp;)^*$
Ids	$\in$	*Identifiers*	::=	$x \mid x, x$
pr	$\in$	*Program*	::=	$\{f_1\ (x_{1_1} \cdots x_{k_1}) = e_1;$ $\vdots$ $f_n\ (x_{1_n} \cdots x_{k_n}) = e_n\}$

Figure 2.a: The Syntax of the Source Language

op	$\in$			{sel, upd, left_part, right_part, +, array, combine, ...}
t_i	$\in$	TV		Temporary variables
se	$\in$	SE	::=	$c \mid x \mid t_i$
e	$\in$	$IExp$	::=	$se \mid op(se_1, \ldots, se_n)$ $\mid f(se_1, \ldots, se_n)$ $\mid$ if se then e_1 else e_2 $\mid$ let $[t_1 = e_1, \ldots, t_n = e_n]$ in $t_i$end
pr	$\in$	$IProg$	::=	$\{f_1\ (x_{1_1} \cdots x_{m_1}) = e_1;$ $\vdots$ $f_n\ (x_{1_n} \cdots x_{m_n}) = e_n\}$

Figure 2.b: The Syntax of the Intermediate Language

V	=	program variables	
F	=	user defined function names	
D	=	$\mathcal{P}(V \cup \{b\})$	domain of abstract values
$VEnv$	=	$V \rightarrow D$	variable environments
$FEnv$	=	$F \rightarrow D^* \rightarrow D$	propagation function environments

Figure 3.a: Domains for Propagation Analysis

$\mathcal{O}[\![op]\!] = \lambda x_1, \ldots, x_n.\emptyset$ where $op \in$ { upd, `left_part`, `right_part`, `array`, `combine`,... }
$\mathcal{O}[\![op]\!] = \lambda x_1, \ldots, x_n.\{b\}$ where $op \in$ is { +, -, `sel`,... }

Figure 3.b: Propagation functions for primitive operators

$$
\begin{array}{lcl}
\mathcal{H}[\![\mathrm{c}]\!]\sigma\rho & = & \{b\} \\
\mathcal{H}[\![x]\!]\sigma\rho & = & \sigma[x] \\
\mathcal{H}[\![t_i]\!]\sigma\ \rho & = & \mathit{let}\ v = \mathcal{H}[\![\mathit{expr\_of}(t_i)]\!]\sigma\ \rho \\
 & & \mathit{in} \\
 & & \quad \mathit{if}\ (v = \emptyset)\ \mathit{then}\ \{t_i\}\ \mathit{else}\ v \\
\mathcal{H}[\![op]\!](se_1, \ldots, se_n)\sigma\rho & = & \mathcal{O}[\![op]\!](\mathcal{H}[\![e_1]\!]\sigma\ \rho, \ldots, \mathcal{H}[\![e_n]\!]\sigma\ \rho) \\
\mathcal{H}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\sigma\rho & = & \mathcal{H}[\![e_1]\!]\sigma\rho\ \cup\ \mathcal{H}[\![e_2]\!]\sigma\rho \\
\mathcal{H}[\![f_k(se_1, \ldots, se_n)]\!]\sigma\rho & = & \rho[f_k](\mathcal{H}[\![se_1]\!]\sigma\rho, \ldots, \mathcal{H}[\![se_n]\!]\sigma\rho) \\
\mathcal{H}[\![\ \mathtt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ \ t_i\ \mathtt{end}\ ]\!]\sigma\rho & = & \mathcal{H}[\![t_i]\!]\sigma\ \rho
\end{array}
$$

Figure 3.c: Propagation function $\mathcal{H} : IExp \rightarrow VEnv \rightarrow FEnv \rightarrow D$

$$\mathcal{H}_p[\![pr]\!] = fix(\lambda\sigma.\ \sigma[f_i \mapsto \lambda y_1, \ldots, y_k.\ (\mathcal{H}[\![exp_i]\!][x_{k:1} \mapsto y_1, \ldots, x_{i:k} \mapsto y_k]\sigma) \cap (\bigcup_{i=1}^{k} y_i)]$$

Figure 3.d: $\mathcal{H}_p : IProg \rightarrow FEnv$

[2] G. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[3] A. Bloss. *Path Analysis and Optimization of Non-strict Functional Languages.* PhD thesis, Yale University, Dept. of Computer Science, 1989.

[4] A. Bloss. Update analysis and efficient implementation of functional aggregates. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 26–38, 1989.

[5] D. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Dept. of Computer Science, 1989.

[6] D. Cann. Retire Fortran? a debate rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[7] A. Deutsch. On determining the lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *ACM Symposium on Principles of Programming Languages*, pages 157–168, 1990.

[8] M. Draghicescu and S. Purushothaman. A compositional analysis of evaluation-order and its applications. In *ACM Conference on Lisp and Functional Programming*, pages 242–249, 1990.

[9] P. Fradet. Syntactic detection of single-threading using continuations. In *Functional Programming Languages and Computer Architecture*, pages 241–258. Springer Verlag LNCS 523, 1991.

[10] G. H. Golub. *Matrix Computations.* The Johns Hopkins University Press, 1988.

[11] G. H. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, Inc., 1993.

[12] C. Gomard and P. Sestoft. Globalization and live variables. In *Proceedings of the Symposium on Partial Evaluation and Semantic Based Program Manipulation (PEPM)*, pages 166–176, 1991.

[13] K. Gopinath. *Copy Elimination in Single Assignment Languages.* PhD thesis, Stanford University, Computer Systems Laboratory, 1988.

[14] K. Gopinath. Copy elimination in functional languages. In *ACM Symposium on Principles of Programming Languages*, 1989.

[15] J. Guzman. *On Expressing the Mutation of State in a Functional Programming Language.* PhD thesis, Yale University, Dept. of Computer Science, 1993.

[16] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, 1990.

[17] P. Hudak. Arrays, non-determinism, side-effects, and parallelism: A functional perspective. In *Graph Reduction*, pages 312–327. Springer-Verlag, LNCS 279, 1986.

[18] P. Hudak. A semantic model of reference counting and its abstraction. In *ACM Conference on Lisp and Functional Programming*, 1986.

[19] P. Hudak. Mutable abstract datatypes or how to have your state and munge it too. Technical report, Department of Computer Science, Yale University, 1993.

[20] R. Nikhil. Id (version 90.0) reference manual. Technical Report CSG Memo 284-1, MIT, 545 Technology Square, Cambridge, MA 02139, August 1990.

[21] A. Sastry. *Efficient Array Update Analysis of Strict Functional Languages.* PhD thesis, University of Oregon, Dept. of Computer Science, June 1994. (in preparation).

[22] A. Sastry, W. Clinger, and Z. Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275, 1993.

[23] D. Schmidt. Detecting global variables in denotational specifications. *ACM TOPLAS*, 7(2):299–310, 1985.

[24] P. Sestoft. Replacing function parameters with global variables. Master's thesis, DIKU, University of Copenhagen, Oct. 1988.

[25] P. Sestoft. Replacing function parameters by global variables. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, 1989.

[26] G. L. Steele Jr. and W. Hillis. Connection machine lisp: fine grained parallel symbolic processing. In *ACM Symposium on Lisp and Functional Programming*, pages 279–297, 1986.

[27] V. Swarup, U. Reddy, and E. Ireland. Assignments for applicative languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 192–214. Springer Verlag LNCS 523, 1991.

[28] P. Wadler. A new array operation. In *Graph Reduction*, pages 328–335. Springer-Verlag, LNCS 279, 1986.

[29] P. Wadler. Comprehending monads. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 61–78, 1990.

[30] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods.* North Holland, 1990. Presented at IFIP TC2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, April 1990.