To appear in Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993.

Transforming High-Level Data-Parallel Programs into Vector Operations†

Jan F. Prins and Daniel W. Palmer
Department of Computer Science
University of North Carolina
Chapel Hill NC 27599-3175
{prins,palmerd}@cs.unc.edu

Abstract

Efficient parallel execution of a high-level data-parallel language based on nested sequences, higher order functions and generalized iterators can be realized in the vector model using a suitable representation of nested sequences and a small set of transformational rules to distribute iterators through the constructs of the language.

1. Introduction

Currently, the development of parallel programs primarily takes place in low-level machine-specific programming languages since these are typically the only languages supported on parallel machines. In this setting, prototyping is a painful process, since small changes in the high-level approach precipitate a flood of changes in low-level details. To make things worse, little or none of this effort may be portable to other settings.

Proteus[1] is a high-level language designed for the prototyping of parallel computations [MNP+91, NP92]. A *Proteus* program specifies parallelism in a high-level and machine-independent fashion. The parallel semantics of such a program can be simulated sequentially, to observe its behavior and make measurements of machine-independent characteristics such as total work and available concurrency. Actual parallel execution on a parallel machine can be obtained by *directed transformation* of the prototype to place it in a restricted form that can then be translated directly to a low-level (possibly machine specific) programming language. We use the KIDS [Smith90] interactive program development system to manage the transformation and translation process. In this fashion, a high-level prototype parallel computation can be experimentally developed and subsequently refined into a parallel program executing on a parallel machine.

In this paper, we are concerned with the directed transformation of *data-parallel Proteus* programs. The data-parallel constructs of *Proteus* permit the construction and manipulation of aggregate values (e.g. sets and sequences) and, in particular, include the ability to apply an arbitrary function to each element of an aggregate value to yield an aggregate result.

High-level programming languages like APL [Iver62] and SETL [Schw70] pioneered the inclusion of data-parallel constructs to gain expressive power by bringing the languages closer to familiar and powerful mathematical notations. The aggregate in the original APL language was the *flat* array, an array whose elements are all scalar values. To obtain fully general data-parallelism, in which *any* function including data-parallel functions can be applied in a data-parallel fashion, requires *nested* aggregates, in which elements may themselves be aggregates. A nested array foundation for APL was described by [More79], and can be found in NIAL, APL2, J, SETL and FP [Back78]. Although fully general data-parallel programs are conveniently specified in these languages, they are not executed in parallel. This is due to implicit serial dependencies in the specification of apply-to-each operations and the apparent requirement for complex and fine-grain MIMD control of execution. Thus these languages are not parallel programming languages.

Languages in which data-parallelism is the mechanism used to specify actual parallel computation such as *Lisp, MPL, and DAP-Fortran have historically targeted specific SIMD parallel computers. More recent languages like CMFortran and C* are portable across various SIMD and MIMD machines. The aggregates in these languages are restricted to flat arrays distributed in a regular manner over processors in an effort to predict and minimize communication requirements in execution [KLS+90, Prin90]. Because aggregates are flat, only a limited class of arithmetic and logical operations may be applied in a data-parallel fashion.

Consequently, using these languages, it is not possible to directly express *nested parallelism* –the data-parallel application of a function which is itself data-parallel.

†This work supported in part by DARPA/ISTO,monitored under ONR Contract N00014-91-C-0114

[1] The *Proteus* language is a component of the DARPA Prototech effort.

For example, a data-parallel sort function can not be applied in parallel to every sequence in a collection of sequences. Yet this is the key step in several parallel divide-and-conquer sorting algorithms. Indeed, there is extensive evidence that nested data-parallelism is an important component in the compact expression of efficient parallel computations [Blel90, Skil90, MNP+91]. The difficulty is not in the languages, since general data-parallel languages can easily express nested parallel computations. Rather the problem lies in the difficulty of translating nested parallelism to achieve efficient parallel execution.

A major step in this direction was taken in [Blel90] where it was shown that for nested sequence aggregates subject to a restricted set of operations, an equivalent *vector model* program operating on segmented flat sequences can be derived. The vector model is efficiently executed on a wide class of parallel machines. Building on these techniques, the transformations presented in this paper give a simple mechanism to transform the fully general data-parallelism available in *Proteus* programs into the vector model.

1.1 Related work

The problem of deriving parallel programs by transformation has received wide attention. In this paper, we are concerned specifically with the translation of data-parallelism, so we restrict our review of related work to that concerned with the implementation of nested parallelism.

CM Lisp [SH86] and Paralation Lisp [Sabo88] are fully-general data-parallel languages implemented as high-level programming languages for the Connection Machine. However, implementations of these languages apply nested data-parallel operations in a serial fashion. McCrosky [McCr87] describes a way to represent the nested arrays of APL and gives implementations for APL primitives on a SIMD execution model, but nested parallel execution is not addressed. Philippsen [PTH91] describes an implementation of nested parallelism in Modula-2* for a SIMD computer, but Modula-2* has no data-parallel nested aggregates. So while data parallel operations may be nested the programmer must orchestrate their parallel access to the appropriate portions of a shared global variable. This requires extensive bookkeeping and use of low-level facilities; thus, we believe that the expressive utility of nested parallelism in this setting is limited.

In [BS90] it was shown how to compile a subset of Paralation LISP into vector model code. More recently, the nested vector model language NESL [Blel92] has used similar techniques to target the portable vector model intermediate language CVL. Compared to these approaches, the translation of *Proteus* includes translation of function values (which are critical elements of the higher-order data-parallel style), and starts from a more general definition of the data-parallel construct (the iterator in *Proteus*) based on a shared memory view. A key contribution of this paper, consistent with our aims for parallel program development by refinement, is the transformational approach to the translation problem.

The remainder of this paper is organized as follows. The next section describes a subset of the *Proteus* notation. Section 3 details language transformations to remove iterators from a *Proteus* expression. Section 4 describes the vector model representation of nested sequences and a translation rule that replaces parallel functions operating on nested sequences with simple vector model functions. Section 5 demonstrates the rules on a simple example. We conclude with implementation status, and a discussion of results.

2. Data-Parallel Expression Language

We describe a subset $\mathcal{P}$ of *Proteus* that can be transformed for vector-model execution. This subset is a restriction of *Proteus* in three ways. First, we restrict ourselves to the expression language; thus the Proteus notion of state is not considered. Second, we require that the types of all expressions be static and monomorphic; *Proteus* expressions in general may exhibit dynamic and polymorphic types. Since overloading is permitted in $\mathcal{P}$, a polymorphic *Proteus* function can be instantiated with several different monomorphic argument types. Finally, the types of values and operations available are restricted to simplify the exposition. For example, function values must be fully-parameterized, set-valued aggregates are not considered, the set of scalar types is limited, and only a small number of operations on sequences are provided. Extension of this last restriction should be relatively simple. To achieve programs that meet all of these restrictions, prior directed transformation steps may be required. The restrictions are not overly limiting because the subset is highly expressive.

The types of $\mathcal{P}$ include scalar types, arbitrarily nested sequences, tuple types and function types generated by the following CFG:

$$T ::= Int \mid Bool \mid Seq(T) \mid (T \times ... \times T) \mid (T \rightarrow T)$$

In contrast to data-parallel languages that do not distinguish between tuples and sequences, the sequences in $\mathcal{P}$ are *homogeneous*. This requirement is needed to statically type all expressions.

Expressions in $\mathcal{P}$ are composed using the constructs in Table 1. A small number of *basic functions* predefined in $\mathcal{P}$ are given in Table 2; other functions can be constructed in terms of these basic functions. We note that we have deviated somewhat from the usual *Proteus* syntax.

construct	meaning
$(e_f)(e_1,...,e_n)$	application of function value e_f to arguments $(e_1,...,e_n)$
fun $(x_1, ... ,x_n)$ e	λ–abstraction of body e with parameters $x_1, ... ,x_n$
let $x = e_1$ **in** e_2	value of e_2 with x bound to value of e_1

if b
then e_1
else e_2
	yields e_1 if b is true else yields e_2

Table 1. Basic constructors of $\mathcal{P}$

The index origin for sequences is 1, hence V`[1][2]` is the second element of the first sequence in the nested sequence V. The operation `restrict`(V,M) with $\#V$ $\#M$ yields V with all elements corresponding to false values in M removed. If R = `combine`(M,V,U) where $\#M = \#V + \#U$, then `restrict`$(R,M) = V$ and `restrict`$(R, \sim M) = U$ (here $\sim M$ denotes the element-wise complement of M).

The `dist` operation replicates values in the first sequence by the corresponding value in the second sequence. For example, `dist`([3,4,5],[3,2,1]) yields [[3,3,3],[4,4],[5]].

The remaining construct in $\mathcal{P}$ is the *iterator* which is the source of all data-parallelism. Its form is:

$$[x \leftarrow d\colon e]$$

where x is an identifier, d is a sequence-valued expression of type $Seq(\alpha)$ called the domain of the iterator, and e is an expression of type β under the assumption that free occurrences of identifier x have type α. The iterator yields a value of type $Seq(\beta)$ defined as follows (e_y^x denotes the syntactic replacement of every free occurrence of x in e by the expression y):

$$\forall k \in 1..\#d\colon \quad [x \leftarrow d\colon e][k] \equiv (e)_{d[k]}^{x}$$

This definition gives the value of an arbitrary element of the result independent of the values of the other elements, hence a natural implementation is to evaluate all elements of the result in parallel.

It is often convenient to restrict the set of values for which e will be evaluated to elements from d satisfying a predicate b in which there are free occurrences of x. Thus we define

```
[x  ← d | b : e]  ≡
  let
    T = restrict(d,[x  ← d : b])
  in
    [t  ← T: e_t^x]
```

The subset $\mathcal{P}$ is a flexible and comprehensive notation that can be used to express scalar functions,

```
fun odd(a) = (1 == (a mod 2))
```

data-parallel functions,

```
fun sqs(n) = [i ← [1..n]: i*i]

fun concat(V,W) =
  [i ← [1..(#V+#W)]:
      if (i≤#V)
      then V[i]
      else W[i-(#V)] ]
```

higher-order data-parallel functions,

name	notation	signature
scalar functions	`+,-,==`, etc.	$\alpha \rightarrow \beta$ (where $\alpha, \beta \in \{Bool, Int\}$)
tuple_cons	$(e_1, ..., e_n)$	$\alpha_1 \times ... \times \alpha_n \rightarrow (\alpha_1 \times ... \times \alpha_n)$
tuple_extract	$e_t.i$	$(\alpha_1 \times ... \times \alpha_n) \times Int \rightarrow \alpha_i$
seq_cons	$[e_1, ..., e_n]$	$\alpha \times ... \times \alpha \rightarrow Seq(\alpha)$
seq_index	$e_s[e_1][e_2] ... [e_k]$	$Seq^k(\alpha) \times Int^k \rightarrow \alpha$
seq_update	$(e_s;\ [e_1] ... [e_k]: e_v)$	$Seq^k(\alpha) \times Int^k \times \alpha \rightarrow Seq^k(\alpha)$
length	$\#e_s$	$Seq(\alpha) \rightarrow Int$
range	$[e_1\ ..\ e_2]$	$Int \times Int \rightarrow Seq(Int)$
restrict	`restrict`(e_s, e_b)	$Seq(\alpha) \times Seq(Bool) \rightarrow Seq(\alpha)$
combine	`combine` $(e_b\ e_1, e_2)$	$Seq(Bool) \times Seq(\alpha) \times Seq(\alpha) \rightarrow Seq(\alpha)$
distribute	`dist`(e_1, e_2)	$Seq^k(\alpha) \times Seq^k(Int) \rightarrow Seq^{k+1}(\alpha)$

Table 2. Basic functions of $\mathcal{P}$

```
fun reduce(f,V) =
  if (#V = 1)
  then V
  else
    let
      W = [i ← [1..(#V) div 2]:
             f(V[(2*i)-1],V[2*i])]
    in
      if (odd(#V))
      then reduce(f,
           concat(W,[V[#V]]))
      else reduce(f,W)
```

and nested data-parallel functions,

```
fun oddsq(n) =
  [i ← [1..n]| odd(i): sqs(i)]

fun flatten(V) =
  reduce(concat,V)
```

We now turn to the transformation of $\mathcal{P}$.

3. Transforming $\mathcal{P}$ to data-parallel form

The expressive utility of the iterator construct comes from its ability to give a per-element specification of an aggregate result. In this view, an iterator sits at the head of a syntax tree that is repeatedly evaluated with different values for the bound variable to yield successive elements of the result. This per-element view of the computation is, however, rather different from a data-parallel computation in which operations are applied to aggregates of data, instead of single elements.

To place an arbitrary expression involving iterators into a form suitable for data-parallel execution, we use transformation rules that distribute the iterators through the constructs of $\mathcal{P}$ toward the leaves of the syntax tree. When fully distributed through all the internal nodes of a syntax tree, iterators enclose leaves that are simple constants or references to variables. These iterators can then be replaced by sequence-valued expressions that directly generate the equivalent result. An expression transformed in this manner is data-parallel: it contains no iterators and operates on sequence aggregates rather than individual elements.

In order to simplify the presentation of the transformations, we introduce some definitions. Let α be an arbitrary type of $\mathcal{P}$, let d be some non-negative integer. A *depth d frame* of elements of type α is a nested sequence of type $Seq^d(\alpha)$. For a given function

$$g : \alpha_1 \times \ldots \times \alpha_n \rightarrow \beta,$$

the *depth d parallel extension* of g is defined to be the function

$$g^d : Seq^d(\alpha_1) \times \ldots \times Seq^d(\alpha_n) \rightarrow Seq^d(\beta)$$

that applies g to each element in a depth d frame of arguments, yielding a depth d frame of results. Note that under this definition, $g^0 \equiv g$. All arguments of g^d must be *conformable* at depth d, meaning that each argument exhibits the same nesting structure within the depth d frame.

If g is defined as **fun**□g(x_1, ... , x_n) = *e*, for some arbitrary body *e*, then g^d can be derived from g by enclosing *e* within d iterators that enumerate the elements of the arguments at depth d. That is,

```
fun□g^d(V_1, ... , V_n) =                    (R0)
  [i_1 ← [1..#V_1]:
    [i_2 ← [1..#(V_1[i_1])]:
      ...
      [i_d ← [1..#(V_1[i_1]..[i_(d-1)])]:
        let
          x_1 = V_1[i_1]..[i_d],
           ...
          x_n = V_n[i_1]..[i_d]
        in
          e
      ]
    ...
    ]
  ]
```

The conformability requirement permits selection of corresponding elements from all arguments based on the nesting structure of one argument (the first argument in this case).

Definitions of g^d will be overloaded to permit particular arguments to omit the depth d frame. Semantically, such arguments are considered to be replicated into a depth d frame to be conformable with the remaining arguments. This is a generalization of scalar extension found in the operations of many data-parallel languages.

To explain the transformation, consider placing the expression

```
[i ← [1..N]: g(i)]                 (3.1)
```

into data-parallel form where **fun**□g(x) =*e* is some arbitrary function with type *Int* → *Int*. The iterator can be distributed through the function application of g in (3.1) by replacing g with the depth 1 parallel extension g^1 to obtain

```
g^1([i ← [1..N]: i])               (3.2)
```

where

```
fun□g^1(V) =
  [j ← [1..#V]:  let
                   x = V[j]
                 in
                   e
  ]
```

The iterator introduced in g^1 can in turn be eliminated by application of the transformation rules to the body of g^1.

The iterator `[i← [1..N]: i]`remaining in expression (3.2) encloses a simple reference to the bound variable *i* and can be replaced by the sequence valued expression `[1..N]` to yield

$$\mathtt{g}^{1}\mathtt{([1..N])} \tag{3.3}$$

and this is the form of (3.1) that is suited for data-parallel execution (after g^1 is transformed).

The most challenging problem is the construction of sequence-valued expressions to replace simple references inside nested iterators. Each enclosing iterator increases the combinations of values of the bound variables for which the expression can be evaluated simultaneously. Iterators enclosing a constant or a free occurrence of a variable may be replaced directly by the constant or variable since we rely on parallel extensions of functions to replicate such single values to the appropriate depth. For an occurrence of a variable bound in one of the enclosing iterators (variables introduced in iterators or in **`let`** statements enclosed within iterators), the sequence-valued expression will depend on the iterators between its definition and its use.

If we restrict the form of the iterator domain to [1..*e*] where *e* may be an arbitrary expression, then the basic operations `range1` and `dist` and their parallel extensions suffice to build all required sequence-valued expressions. The definitions of these two functions are

```
range1(n) = [1..n]
dist(c,r) = [i ← [1..r]: c]
```

The general form of a collection of iterators enclosing a bound occurrence of a variable is

$$\begin{array}{l}[\mathtt{i}_1 \leftarrow [\mathtt{1..e}_1]\mathtt{:} \\ \qquad \dots\ [\mathtt{i}_d \leftarrow [\mathtt{1..e}_d]\mathtt{:}\ \mathtt{i}_k]\ \dots\ ]\end{array}$$

where i_k is bound in iterator *k*. The expression replacing the nested iterators must yield elements arranged in a depth *d* frame. Generally speaking, the transformation of the upper bound expression e_k of the domain in iterator $k \le d$ yields a depth *k*-1 frame of upper bounds $\tau(e_k)$, hence the values assumed by i_k are given by $V = range1^{k-1}(\tau(e_k))$. To expand this value to a depth *d* frame, we must replicate the values in *V* according to the domain sizes of the iterators *k*+1 ... *d*. Since the size of the domain in iterator *j* on each nested invocation is given by the upper bound expression e_j, this can be accomplished with the expression

$$\begin{array}{l}\mathtt{dist}^{d-1}(\mathtt{dist}^{d-2}(\ \dots \\ \quad \mathtt{dist}^{k+1}(\mathtt{dist}^{k}(\mathtt{V},\ \tau(e_{k+1})), \\ \qquad \tau(e_{k+2})),\ \dots\ ,\tau(e_{d-1})),\ \tau(e_d))\end{array}$$

For example, in the following two iterators

```
[i ← [1..N]: [j ← [1..i]: i]]
[i ← [1..N]: [j ← [1..i]: j]]
```

we have $\tau(e_1) = \mathtt{N}$ and $\tau(e_2) = \mathtt{range1}^{0}\mathtt{(N)} = \mathtt{[1..N]}$. These iterators can be replaced by the expressions

$$\begin{array}{l}\quad \mathtt{dist}^{1}(\mathtt{range1}^{0}(\tau(e_1)),\tau(e_2)) \\ = \mathtt{dist}^{1}(\mathtt{range1}^{0}\mathtt{(N)},\mathtt{range1}^{0}\mathtt{(N)})\end{array}$$

and

$$\begin{array}{l}\quad \mathtt{range1}^{1}(\tau(e_2)) \\ = \mathtt{range1}^{1}(\mathtt{range1}^{0}\mathtt{(N)})\end{array}$$

respectively. In the syntax-directed transformation rules that follow, the expressions replacing nested iterators are constructed incrementally, with the depth of all bound variables increased each time an iterator is encountered.

We now outline the transformations to eliminate iterators from arbitrary programs in $\mathcal{P}$. A program in $\mathcal{P}$ consists of a set of function definitions. The transformations are applied to the body of each function *f* defined in the program as well as to each of the parallel extensions of *f* introduced as a result of the transformations. The number of parallel extensions of *f* that are introduced is a static property of the program. A subsequent translation step provides a single flat-vector implementation that can be used for all depth *d* ≥ 1 parallel extensions of functions.

3.1. Iterator canonical form

We first transform a program so that each iterator is in a *canonical* form. An iterator is in canonical form if its domain is a range of consecutive integers starting with 1. An arbitrary iterator can be placed in this form using the rule

$$\begin{array}{l}[x \leftarrow e_1\mathtt{:}\ e_2] \equiv \\ \qquad \mathbf{let} \\ \qquad\quad \mathtt{V} = e_1 \\ \qquad \mathbf{in} \\ \qquad\quad [\mathtt{i} \leftarrow [\mathtt{1..\#V}]\ \mathtt{:}\ (e_2)^{x}_{\mathtt{V[i]}}]\end{array} \tag{R1}$$

3.2 Iterator elimination

For each function definition $\mathtt{f(x}_1\mathtt{,...,x}_n\mathtt{)\ =\ e}$, we eliminate all iterators from the body of the definition by replacing *e* with the syntax-directed transformation τ(*e*, 0) defined below. In the rules that follow, angle brackets enclose the syntactic category in $\mathcal{P}$ on which τ is being defined. Thus τ(«*id*», *j*) = *id* indicates that τ translates all identifiers to themselves. Refer to the section 5 for an example illustrating the application of the rules.

$$\begin{array}{l}\tau(«id», j) = \\ \quad id\end{array} \tag{R2a}$$

$$\begin{array}{l}\tau(«const», j) = \\ \quad const\end{array} \tag{R2b}$$

```
$\tau$(«$e_f$ ($e_1$, ... , $e_n$)», $j$) =                    (R2c)
  ($\tau$($e_f$, $j$))$^j$ ($\tau$($e_1$, $j$), ... , $\tau$($e_n$, $j$))
```

```
$\tau$(«[$i$ ← [1..$e_1$]: $e_2$]», $j$) =                     (R2d)
  let
    $ib$ = $\tau$($e_1$),
    $i$  = range1$^j$($ib$)
    $v$  = dist$^j$($v$, $ib$)
              for all $v$ that occur in $e_2$ and are
              bound in enclosing iterators
  in
    $\tau$($e_2$, $j$+1)
```

```
$\tau$(«if $e_1$ then $e_2$ else $e_3$», $j$) =                (R2e)
  let
    M = $\tau$($e_1$, $j$)
  in
    let
      R2 =
        if restrict$^j$(M,M) ≠
                empty_frame$^j$(M)
          then
            let
              v = restrict$^j$(v,M)
                for all $v$ that occur in $e_2$
                bound in enclosing iterators
            in $\tau$($e_2$, $j$)
          else
            empty_frame$^j$(M)
      R3 =
        if restrict$^j$(M,not$^j$(M)) ≠
                empty_frame$^j$(M)
          then
            let
            v = restrict$^j$(v,not$^j$(M))
                for all $v$ that occur in $e_3$
                bound in enclosing iterators
            in $\tau$($e_3$, $j$)
          else
            empty_frame$^j$(M)
    in
      combine$^j$(M,R2,R3)
```

```
$\tau$(«let $v$ = $e_1$ in e2», $j$) =                         (R2f)
  let
    $v$ = $\tau$($e_1$, $j$)
  in
    $\tau$($e_2$, $j$)
```

```
$\tau$(«fun ($x_1$, ... ,$x_n$) $e$ », $j$) =                  (R2g)
  fun ($x_1$, ... ,$x_n$) $e$
```

where empty_frame^j(V) creates a depth d frame that has the same structure as V but contains no elements at depth d. Since all function definitions are fully parameterized, a function definition is independent of any surrounding iterators and reduces to the case of an iterator surrounding a simple constant.

The transformations have the potential to create a large number of function definitions, since each function may be called in a variety of iteration depths d and with a variety of depth 0 and depth d argument frames. However, as we shall see in the next section, all depth 0 argument frames can be converted to depth d argument frames in a uniform way, and all depth d >1 parallel extensions of a function can be implemented by using the depth 1 parallel extension.

4. Translation of $\mathcal{P}$ to $\mathcal{V}$

Iterator free data-parallel expressions in $\mathcal{P}$, as produced by the transformations of the previous section, can be translated to an implementation of vector-model parallelism [Blel90] such as C with the C Vector Library of [BCS+90]. Here, we characterize such an implementation, $\mathcal{V}$, as a flat, low-level data-parallel notation.

The types of $\mathcal{V}$ are scalar types, flat sequence types, tuple types and function types that are generated by the following CFG:

$$\begin{array}{lll} T & ::= & Int \mid Bool \mid Seq(Int) \mid Seq(Bool) \mid \\ & & Seq(T \rightarrow T) \mid (T \times \ldots \times T) \mid (T \rightarrow T) \end{array}$$

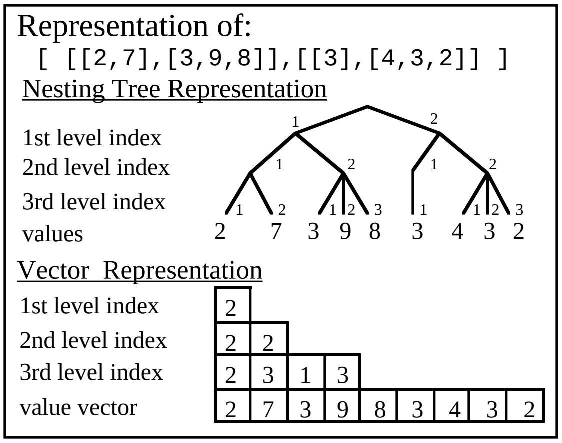
The basic constructors of $\mathcal{V}$ include the constructors given in Table 1. Note that $\mathcal{V}$ does not include the iterator construct. The operations of $\mathcal{V}$ are the operations given in table 2, and the depth 1 parallel extensions of each of these functions.

4.1 Representation of Nested Sequences

All values of $\mathcal{P}$, with the exception of nested sequences and sequences of tuples, can be directly represented in $\mathcal{V}$. A sequence value in $\mathcal{P}$ with type $Seq^d(\alpha)$ for some type α in $\mathcal{P}$ and $d \geq 1$ has a *vector representation* in $\mathcal{V}$ as follows. A collection of k vectors $V_1, \ldots, V_k$ are used, where $V_1,..,V_d$ are *descriptor* vectors, V_1 is always a singleton vector and $V_{d+1},..,V_k$ are *value* vectors, If α is a scalar type then $k = d+1$, but if α is a tuple type then $k > d+1$. A vector representation and its equivalent *nesting tree* representation are shown in figure 1. Each descriptor vector indicates the partitioning for the vector on the level below. An important characteristic of this representation is that:

$$\forall\ i\!: 1 \leq i \leq d \ \ \#V_{i+1} = \Sigma\ V_i$$

Only adjacent descriptor vectors are directly related to each other, so maintaining a consistent representation when performing sequence operations is relatively simple. Note that empty sequences at the leaves of the nesting tree are represented by a zero index in the lowest-level descriptor vector.



1st level index	2								
2nd level index	2	2							
3rd level index	2	3	1	3					
value vector	2	7	3	9	8	3	4	3	2

figure 1

4.2 Operations on Vector Representation of Nested Sequences

There are two operations that directly manipulate the representations of nested sequences: *extract* and *insert*. For V a sequence of depth $d+k$, *extract*(V,d) flattens the top d nesting levels (see figure 2) and can be implemented by replacing the top d descriptors by the singleton vector $V_1 = [\ \Sigma V_d]$.

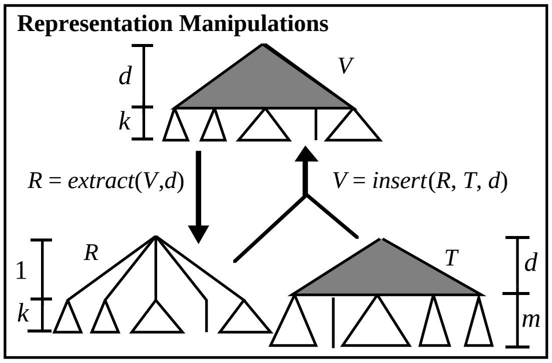


figure 2

The *insert* operation forms a depth $d+k$ sequence from a sequence of depth $k+1$ and another sequence of depth greater than d. In our translation, the second sequence is always the same as the sequence used in an *extract* operation, but this is not required. *Insert*(R,V,d) removes the top descriptor from R and replaces it with the top d descriptors from V (see figure 2). We require that $\#T_d = \mathrm{R}_1[1]$. This insures that the result of an *insert* operation is a consistent nested sequence representation. Note that $V = insert(extract(V,d),V,d))$ for any $d \leq$ depth of V.

4.3 Translation of Functions on Nested Sequences

A depth d parallel extension of f, operates on values at depth d without altering the frame, hence it suffices to use f^1, the simple depth 1 parallel extension of f, to be used in all contexts. To achieve the effect of $f^d(e)$, we flatten the frame around values in e, apply f^1, and restore the frame around the result of this application.

This is accomplished by using the *extract* operation to remove the nesting frame of a sequence so the simple data-parallel function can be applied. The result is then re-attached to a frame using the *insert* instruction as shown in figure 3.

$$\begin{tikzcd}
V \arrow[r, "f^d"] \arrow[d, "\mathtt{extract}(V,d\ )"'] & S \\
W \arrow[r, "f^1"'] & R \arrow[u, "\mathtt{insert}(R,V,d\ )"']
\end{tikzcd}$$

figure 3

This principle is the basis for our translation rule. It eliminates the need for the higher depth parallel extensions of functions. The translation rule for application of f^d with $d > 1$ is:

```
f^d (e1, e2, ...,en)  ≡                       (T1)
   let V1=e1, V2=e2,...,Vn=en
   in
     Insert
       (f′ (Extract(V1,d),
            Extract(V2,d),...
            Extract(Vn,d)),
     V1,d)
```

If f is a function-valued parameter to a function g, it is necessary to pass f in invocations of g as a pair (f, f^1), so that the correct version can be used in a given context.

4.4 Implementation of Functions of $\mathcal{P}$

After application of the translation rule, all expressions in $\mathcal{P}$ are in terms of the basic functions of $\mathcal{V}$. To realize parallel execution, $\mathcal{V}$ is in turn translated to some executable notation. In our case this is C with the vector operations provided by CVL. The details of the implementation in CVL of the operations in table 2 and their depth 1 parallel extensions is beyond the scope of the current paper. With these implementations and the translation rule, we can claim that all possible expressions in $\mathcal{P}$ can be represented and executed using only constructs in $\mathcal{V}$.

4.5 Vector Level Optimizations

Because they are so frequently applied, it is critically important that the *insert* and *extract* operations have minimal overhead. The selection of the tree vector representation for nested sequences was chosen specifically because those operations can be implemented inexpensively on this representation.

Certain functions may have parameters that should not be extracted and inserted. Consider the function *seq_index*. If the source parameter is fixed relative to the surrounding iterators, there is no need to replicate it when using iterator transformations. If the replication is applied in these cases, the result is that each set of index values would retrieve from their own copy of the source sequence, clearly a waste of time and space. We can avoid such waste by not always replicating depth 0 argument frames.

Clearly it would be advantageous to increase the set of predefined functions in $\mathcal{V}$ since their direct implementation can be much more efficient than their evaluation as a user-defined function. Consider, for example, the function *flatten* defined in section 2. *Flatten* can be implemented simply by creating a new descriptor vector for the values rather than by creating a new value using the *reduce* and *concat* function definitions.

5. Example

We illustrate the transformations and translations described in the previous two sections on the simple expression

```
[k ← [1..5]: sqs(k)]
```

using the function *sqs* defined in section 2:

```
fun sqs(n) = [j ← [1..n]:mult(j,j)]
```

The top level expression is transformed to

```
let
  kb = 5
  k  = range1^0(kb)
in
  sqs^1(k)
```

Since the resultant expression has introduced `sqs`^{1}, we must define this function using `sqs` and transform it to remove iterators.

```
    {R0}
fun sqs^1(V) =

  τ(«[i ← [1..length(V)] :
     let
       n = seq_index(V,i)
     in
       [j ← [1..n]: mult(j,j)] ]»,0)
```

```
    {R2d}
τ(«[i ← [1..length(V)] :
  let
    n = index(V,i)
  in
    [j ← [1..n]: mult(j,j)] ]»,0) =

  let
    ib = τ(«length(V)»,0)
    i = range1^0(ib)
  in
    τ(«let
        n = seq_index(V,i)
      in
        [j ← [1..n]
          :mult(j,j)]»,1)
```

```
    {R2c}
τ(«length(V)»,0) =

  τ(«length»,0)^0(τ(«V»,0))
```

```
    {R2a}
τ(«length»,0)^0(τ(«V»,0)) =

  length^0(V)
```

```
    {R2f}
τ(«let
    n = index(V,i)
  in
     [j ← [1..n]:mult(j,j)]»,1) =

  let
    n = τ(«seq_index(V,i)»,1)
  in
    τ(«[j ← [1..n] :mult(j,j)]»,1)
```

```
    {R2c}
τ(«seq_index(V,i)»,1) =

  τ(«seq_index»,1)^1(τ(«V»,1),
                     τ(«i»,1))
```

```
    {R2a}
τ(«seq_index»,1)^1(τ(«V»,1),τ(«i»,1))=

  seq_index^1(V,i)
```

```
    {R2d}
τ(«[j ← [1..n] :mult(j,j)]»,1) =

  let
    jb = τ(«n»,1)
    j = range1$^1$(jb)
  in
    τ(«mult(j,j)»,2)
```

```
  {R2a}
τ(«n»,1) = n
```

```
  {R2d}
τ(«mult(j,j)»,2) =

  τ(«mult», 2)$^2$ (τ(«j»,2), τ(«j»,2))
```

```
  {R2a}
τ(«mult», 2)$^2$ (τ(«j»,2), τ(«j»,2)) =

  mult$^2$(j,j)
```

Combining all the results yields the transformed version of the function sqs^1:

```
fun sqs$^1$(V) =
  let
    ib = #V
    i  = rangel$^0$(ib)
  in
    let
      n = seq_index$^1$(V,i)
    in
      let
        jb = n
        j  = range1$^1$(jb)
      in
        mult$^2$(j,j)
```

The translation rule must be applied to both the top level expression and the transformed function. The expression remains unchanged, but the invocation of *mult* is translated.

```
  {T1}
mult$^2$(j,j) =

  insert(mult$^1$(extract(j,1),
               extract(j,1),j,1) )
```

C code can be generated directly from this final transfomed *sqs*^{1} program by the KIDS system. The code produced for this example is:

```
nseq sqs_1(nseq s)
  { nseq q =
        range1_1(
          index_1(s,
            range1_0(length(s))
          )
        )
    return
      insert(mult_1(extract(q,1),
                    extract(q,1)
                 ),q,1);
  }
```

With CVL implementations of the primitves of $\mathcal{V}$ this code can be directly executed on a wide variety of parallel computers.

6. Discussion

Implementation Status

Most of the primitives of $\mathcal{V}$ have been implemented in CVL at the time of this writing. The transformations have been implemented by our colleagues at Kestrel Institute on the KIDS system. Combining these efforts gives us an end-to-end system that can automatically transform and execute *Proteus* programs. We are currently executing using the sequential version of the vector library, but attaining actual parallel execution only requires recompilation on parallel hardware. We are currently implementing the remaining language primitives and investigating improvements to the transformations that yield more efficient code.

Implications for sequential execution

In addition to providing a route to parallel execution of high-level data-parallel programs, the transformations can also be of use in a sequential setting. One of the objections often raised to the iterator construct is that it incurs substantial overhead in the repeated evaluation of the iterator body. The transformation rules suggest, however, that by replacing the iterators with vector primitives, the overhead of repeated calls can be eliminated.

Conclusions

We have described a simple but comprehensive transformational framework to reduce arbitrary data-parallel programs to vector operations. The class of data-parallel expressions that can be transformed in this manner includes irregular parallel computations (as found in the parallel application of a function to each of a collection of sequences of different length), recursive parallel computations (as found, for example, in parallel divide-and-conquer algorithms), and high-order parallel function application (as found in the parallel reduction of a sequence of values using an arbitrary function). In each case the resultant CVL program operates on simple vectors and can be executed with excellent load-balance on a wide class of parallel machines. The transformations were implemented using the KIDS system and, together with other transformations, form the basis of our strategy to achieve parallel execution of prototype programs by directed transformation.

Acknowledgments

We would like to thank Stephen Westfold of the Kestrel Institute for his implementation efforts and improvement of the transformation rules.

Bibliography

[Back78] Backus, J., "Can Programming be Liberated from the VonNeumann Style? A Functional Style and its Algebra of Programs", *Communications of the ACM*, 1978.

[BCS+90] Blelloch, G., Chatterjee, S., Sipelstein, J., Zahga, M., "CVL: A C Vector-Library", Draft Technical Note, Carnegie Mellon University, 1990.

[Blel90] Blelloch, G., *Vector Models for Data-Parallel Computing,* MIT Press, 1990.

[Blel92] Blelloch, G., “NESL: A Nested Data-Parallel Language”, Technical Report CMU-CS-92-103, Carnegie Mellon University, January 1990.

[BS90] Blelloch, G., Sabot, G., "Compiling Collection-Oriented Languages onto Massively Parallel Computers", *Journal of Parallel and Distributed Computing,* 8(2), February 1990.

[Iver62] Iverson, K., *A Programming Language*. Wiley, New York, 1962.

[KLS+90] Knobe, K., Lukas, J., Steele, G., "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines", *Journal of Parallel and Distributed Computing* 8, 1990.

[McCr87] McCrosky, C., "Realizing the Parallelism of Array-based Computation", *Parallel Computing 10* 1989.

[MNP+91] Mills, P., Nyland, L., Prins, J., Reif, J., Wagner, R.,"Prototyping Parallel and Distributed Programs in *Proteus*", *Proceedings Symposium on Parallel and Distributed Processing 92.* 1992.

[MNP+92] Mills, P., Nyland, L., Prins, J., Reif, J., "Prototyping N-body Simulations in *Proteus*", *Proceedings IPPS 92*, IEEE, 1992.

[More79] More, T., "The Nested Rectangular Array as a Model of Data" *APL79 Conference Proceedings.* ACM 1979.

[NP92] Nyland, L., Prins, J., "Prototyping Parallel Programs", Proceedings *1992 Dartmouth Institute for Advanced Graduate Studies in Parallel Computing Symposium*, 1992.

[Prin90] Prins, J., "A Framework for Efficient Execution of Array-Based Languages on SIMD Computers", *Proceedings Frontiers 90,* IEEE 1990.

[PTH91] Philippsen, M., Tichy, W., Herter, C., "Modula-2* and its Compilation", *Proceedings Austrian Conference on Parallel Computing*, 1991.

[Sabo88] Sabot, G., *The Paralation Model : Architecture-Independent Parallel Programing,* MIT Press, 1988.

[Schw70] Schwartz, J., "Set Theory as a Language for Program Specification and Programming", Technical Report Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1970.

[Skil90] Skillicorn, D., "Architecture-Independent Parallel Computation", *IEEE Computer 11*, Vol.23 No. 12 (Dec.), 1990.

[Smit90] Smith, D., "KIDS - A Semi-automatic Program Development System", IEEE *Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering* Vol. 16, No. 9, 1990.

[SH86] Steele, G. L., Hillis, W., "Connection Machine LISP: Fine-grained Parallel Symbolic Processing ", *Proceedings 1986 ACM Conference on Lisp and Functional Programming* ACM, 1986.