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 (919)-962-1913
{prins,palmerd}@cs.unc.edu

Abstract

Fully-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, development of parallel programs often 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 and assess its behavior. Actual parallel execution on a parallel machine, or class of machines 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. The transformation and translation are managed by the KIDS [Smith90] interactive program development system. In this fashion, a high-level prototype parallel computation can be experimentally developed and subsequently evolved 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 (sets or sequences) and, in particular, include the ability to apply a function in parallel to all elements 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 can be applied in a data-parallel fashion, requires *nested* aggregates, in which elements may themselves be aggregates. A nested array

[†] This work supported in part by DARPA/ISTO Contract N00014-91-C-0114.

[1]The *Proteus* language is a component of the DARPA CPL (Common Prototyping Language) effort.

foundation for APL was described by [More79], and can be found in NIAL, APL2, J, SETL and FP [Back78]. Although data-parallel programs are conveniently specified in these languages, they can only be executed sequentially due to the complex and fine-grain synchronization requirements in a parallel implementation of general data-parallelism. 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. 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 any parallel divide-and-conquer sorting algorithm. 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 fully-parallel execution.

A major step in this direction was developed 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 partitioned (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.

Related work

Many researchers have addressed the problem of deriving parallel programs by transformation. 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 also 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 nested parallel operations may be applied, 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 is 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 yield vector model code. 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 a more general definition of the iterator construct (it includes reference indexing) than that found in Paralation LISP. A key contribution of this paper, consistent with our aims for parallel program development by refinement, is the transformational approach to the translation.

The remainder of this paper is organized as follows. The next section gives a subset of *Proteus* notation. Section 3 describes the representation of nested sequences in the vector model and their basic operations. Section 4 gives the transformation and translation rules. Section 5 applies the rules to a simple example. We conclude with implementation status.

2. Data-Parallel Expression Language

We describe a subset **P** of *Proteus* that can be transformed for fully-parallel execution. This subset is restricted in two ways. First, the types of values and operations in *Proteus* are restricted to simplify the exposition. For example, 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 these restrictions is straightforward. Second, the subset is purely functional; hence the *Proteus* notion of state is not addressed. To achieve programs that meet this restriction, prior directed transformation steps may be required. These restrictions, however, are not overly limiting because the subset is highly expressive.

The types of **P** consist of scalar types, arbitrarily nested sequences, tuple types and function types. More precisely the types in **P** are generated by the following CFG:

$$T \ ::= Int \mid Bool \mid Seq(T) \mid (T \times \ldots \times T) \mid (T \to T)$$

In contrast to data-parallel languages that do not distinguish between tuples and sequences, the sequences in **P** are *homogeneous*. This homogeneity allows expressions in the *Proteus* subset to be fully typed up to the level of parametric polymorphism and the translation of nested parallelism. Expressions in **P** are composed using the constructs in Table 1.

construct	meaning
$(e_1)(e_2)$	application of function value e_1 to argument e_2
fun $(x)\ e$	λ–abstraction of body e with parameter x
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 **P**

A small number of *basic functions* predefined in **P** are given in Table 2; other functions can be constructed in terms of these basic 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	$e.i$	$(\alpha_1 \times ... \times \alpha_n) \times Int \rightarrow \alpha_\iota$
seq_cons	$[e_1, ..., e_n]$	$\alpha \times ... \times \alpha \rightarrow Seq(\alpha)$
seq_extract	$e_1[e_2]$	$Seq^k(\alpha) \times Int^k \rightarrow \alpha$
seq_upd	`update`(e_1, e_2, e_3)	$Seq^k(\alpha) \times Int^k \times \alpha \rightarrow Seq^k(\alpha)$
length	$\#e_1$	$Seq(\alpha) \rightarrow Int$
range	$[e_1 \,..\, e_2]$	$Int \times Int \rightarrow Seq(Int)$
restrict	`restrict`(e, b)	$Seq(\alpha) \times Seq(Bool) \rightarrow Seq(\alpha)$
combine	`combine` (b, e_1, e_2)	$Seq(Bool) \times Seq(\alpha) \times Seq(\alpha) \rightarrow Seq(\alpha)$

Table 2. Basic functions of **P**

The index origin for sequences is 1, hence V`[1]` is the first element in 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 elementwise complement of M).

Global function definitions are written **`fun`** `f(`x`)` = e. A function definition with multiple parameters such as **`fun`** `f(`x_1`, ...,`x_n`)` = e is a syntactic abbreviation for the definition

```
fun f(x) = let x1 = x.1, ..., xn = x.n in e
```

where `x` has the appropriate tuple type. Note that the data-parallel version of a multiple parameter function also expects a tuple as input. The tuple is made of sequences of the proper types. To simplify issues with non-local references, we restrict function values to be fully parameterized λ-abstractions.

The remaining constructor in **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 $Seq(\alpha)$, and e is an expression of type β with the assumption that free occurrences of identifier x have type α. The result is a value of type $Seq(\beta)$ defined as follows:

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

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

$$\begin{array}{l}[x \leftarrow d \mid b : e] \equiv \\ \quad \textbf{let } T = \texttt{restrict}(d, [x \leftarrow d : b]) \textbf{ in } [t \leftarrow T\colon e^{x}_{t}]\end{array}$$

The subset **P** is a flexible and comprehensive notation that can be used to express arbitrary functions, data-parallel functions and generalized data-parallel functions:

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

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)] ]

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)
```

3. Vector-Model Representation and Operations

Expressions formed using the data-parallel notation **P** of the previous section will be transformed and then 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 **V** as a flat, low-level data-parallel notation with the following types and operations.

The types of **V** are scalar types, flat sequence types, tuple types and function types that are generated by the following CFG:

$$T ::= Int \mid Bool \mid Seq(Int) \mid Seq(Bool) \mid Seq(T \to T) \mid (T \times \ldots \times T) \mid (T \to T)$$

The basic constructors of **V** include the constructors given in Table 1. Note that **V** does not contain the iterator construct. The predefined functions of **V** include the following:

- elementwise arithmetic and logical operations (which we denote as the primed version of the scalar function, e.g. +' for elementwise addition).
- reduction operations over vectors using associative scalar primitives with types of the form $\alpha \times \alpha \to \alpha$ such as logical or, addition and maximum.
- parallel prefix versions of the reduction operations above that compute a reduction of each of the *n* prefixes of a vector of length *n*.
- segmented parallel prefix operations that perform parallel prefix operations on each component of an arbitrarily segmented vector.
- parallel permutation and other rearrangement operations on vector values.

All values of **P** with the exception of sequences of tuples and nested sequences can be directly represented in **V**. For the remaining values, we use the following representations.

A sequence value in **P** with type $Seq^d(\alpha)$ for some scalar type α in **P** and $d \geq 1$ is said to have *depth d* and is represented in **V** as follows. A *value* vector is used to hold the actual values and *d descriptor* vectors are used to represent the nesting structure of the sequence. This *nesting tree* representation is shown in figure 1 in the appendix. Note that empty sequences and sub-sequences are easily handled in this representation with a zero index at the appropriate level.

A sequence of tuples in **P** is represented in **V** by a tuple of (same length) sequences. For example, a value of type $Seq(Int \times Seq(Int))$ is represented as a value of type $Seq(Int) \times Seq(Seq(Int))$. In this fashion all values can be reduced to tuples whose components have scalar types and types of the form $Seq^d(\alpha)$ for some scalar type α that are represented as shown above.

Operations on Nested Sequence Representations

The transformed code will use three fundamental operations on the representation of nested sequences. To summarize informally the meaning of these operations, let V, R, and T denote representations of a nested sequences of depth $d+k$, $1+k$, and $d+m$ respectively, while M and L represent flat sequences.

The $Extract(V,k)$ operation flattens the top d nesting levels of V, so that all depth k items appear consecutively in the resulting sequence of depth k+1. The $Insert(R,T,d)$ operation is the inverse operation that restructures R into a depth $d+k$ sequence using the top d levels of nesting structure provided by T. Both of these operations can be implemented by simply associating and dissociating the index levels appropriately. The $Expand(M,L)$ operation replicates the elements of M according to the corresponding values in L. This operation can be implemented using a single operation from **V** . The three operations are illustrated in figure 2 in the appendix using the nesting tree representation.

4. Translation to Vector Model

An overview of the complete translation is shown in the figure 3 in the appendix. The high-level transformations and translations are being implemented using the KIDS system to yield C code with calls to the CVL library. The resulting program can be compiled for execution on any of a number of different platforms.

The power of the iterator construct comes from its ability to give a per-element recipe for the computation of a result value. In this view, the iterator sits at the head of a syntax tree that is repeatedly evaluated with different bindings for the iterated variable. To extract data-parallelism, we distribute the iterator through the constructs of the language towards the leaves of the syntax trees using some simple identities. With the iterators closer to the leaves, the values manipulated by the transformed expression are sequences rather than elements of a sequence, and this corresponds to data-parallel execution.

Consider, for example, the expression

```
        [i ← [1..N]: g(i)]
```

The iterator specifies that for each choice of *i*, an arbitrary function *g* should be applied to *i*. To introduce data-parallelism, we distribute the iterator through the application of *g* to yield:

```
        g'([i ← [1..N]: i])
```

For this transformation to be meaning-preserving we require that *g'* be a data-parallel version of *g,* that is, a function that returns a sequence of results given a sequence of arguments to *g*.

If g is a basic function of **P** , then g' has to be explicitly implemented in terms of operations of **V** . The data-parallel versions of the scalar arithmetic functions are immediately provided by the corresponding elementwise scalar operations of **V**. For each remaining function f in Table 2 with signature $\alpha \rightarrow \beta$, we must provide a function f' with signature $Seq(\alpha) \rightarrow Seq(\beta)$ using the operations provided by **V** operating directly on the representation of the arguments in **V** . We state without further proof that this can be done for the set of operations in Table 2.

If g is a defined function **fun** g(x) = e, with arbitrary body e, the following definition specifies the semantics of g' operating on a sequence V of arguments to g to yield a sequence of results:

```
fun g'(V) =if #V==0 then []:seq(β) else [x ← V: e]
```

The body of g' is precisely the body of g enclosed by an iterator that specifies that e should be evaluated for each item in the argument sequence. The transformations described below can now be applied to distribute the iterator through the syntax tree of e to introduce data-parallelism.

Transformation Rules

We now present a set of five transformation rules that can be used to rewrite any **P** expression e into an equivalent **P** expression $\hat{e}$ such that iterators only encompass simple scalar constants or an occurrence of a variable bound in the surrounding iterator. A subsequent translation replaces the iterator subtrees in expressions with operations to create sequence representations of the result using the operations of **V** . The resultant program, expressed in **V** , can be executed in parallel.

We first transform a program so that each iterator is in a *canonical* form. An iterator is in canonical form if the bound variable ranges over an index set that is a range of integers starting with 1. An arbitrary iterator can be placed in this form using the rule

```
[x ← e1: e2] ≡                                        (R1)
      let  V = e1
      in
             [i ← [1..#V]: (e2)^x_V[i]]
```

To distribute the iterators through expressions we give a transformation rule to distribute an iterator through each of the basic constructors of **P** given in Table 1.

To distribute an iterator through an application of function f, we replace f by its data-parallel version f' .

```
[x ← V: f(e)] ≡  f'([x ← V: e])                      (R2)
```

Note that if two iterators are distributed through a function application, then the result calls for the application of a function f'' . In fact, if k iterators are distributed through a function application the result calls for the application of a function f^k. In the translation section we show that all f^k for $k \geq 1$ can be implemented with f' . 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'), so that the correct version can be used in a given context.

To distribute an iterator through a **let** construct we must create a sequence of bound values that are indexed in the result expression

```
[i ← d:   let t = e1 in e2] ≡                                   (R3)
          let  T = [i ← d: e1]
          in
                [i ← [1..#T]: (e2)^t_T[i]]
```

To distribute an iterator through a conditional expression, we partition the index set into indices for which the conditional is true and indices for which it is false. The two arms are evaluated with the corresponding index set and the results are merged:

```
[i ←[1..n]:   if b then e1 else e2] ≡                           (R4)
   let
      M = [i ← [1..n]: b]
      T = let J = restrict([1..n], M)
          in
               [i ← J: e1]
      E =  let J = restrict([1..n], not'(M))
          in
               [i ← J: e2]
   in
      combine(M,T,E)
```

Since all function definitions are fully parameterized, a function definition is independent of any surrounding iterators. Therefore, an iterator surrounding a function definition is the case of an iterator surrounding a simple constant, and no transformation is needed to distribute the iterator through the construct. Additionally, since all functions have a single tuple parameter, a transformation for an iterator surrounding a tuple is needed. This rule changes a sequence of tuples into a tuple of sequences:

```
[i ← d: (x,y)] ≡  ([i ← d: x], [i ← d: y] )                     (R5)
```

Translation Rules

A single function f with type $\alpha \rightarrow \beta$ may be called in various contexts that differ in the number of surrounding iterators. In the transformation rules, we generated an application of function f^k with signature $seq^k(\alpha)\rightarrow seq^k(\beta)$ in a context with k surrounding iterators. Since the surrounding iterators simply enumerate a depth k "frame" of arguments in such a context, it suffices to generate f' the simple data-parallel version of f, to be used in all contexts. To achieve the effect of $f^k(e)$, we flatten the frame around values in e, apply f', and restore the frame around the result of this application. A pictorial representation is located in figure 4 in the appendix.

```
f^k(e) ≡                                                        (T1)
   let  V = e
   in
         Insert(f' (Extract(V,(depth(V)-(k-1)) ),V,k-1)
```

When the Extract operation is applied to a tuple, it is actually applied to each member of the tuple individually, yielding a tuple of sequences of lesser depth.

Since function sequences can be constructed in **P**, there may exist function applications in an expression to be translated in which a sequence of functions are to be applied to a sequence of arguments. This case can not be executed in a fully parallel fashion in the vector model, since the functions in the sequence may be arbitrarily different. However, we can be assured that there are at most a bounded number of distinct functions in such a sequence. Hence the translation is to apply the data-parallel version of each distinct function in turn to all the arguments corresponding to its occurrence in the function sequence.

The final step in translation is to replace all iterator expressions directly by values of **V** . In general, we have the expression

$[i_1 \leftarrow [1..e_1]\text{: } [i_2 \leftarrow [1..e_2(i_1)]\text{: } \ldots [i_n \leftarrow [1..e_n(i_1,\ldots,i_{n-1})]\text{: } e] \ldots]]$

where e is a constant or i_k for $1 \le k \le n$. Each successively nested iterator requires that we repeat the values taken on by the outer iterator. The translation becomes

```
let                                                              (T2)
   I1  = [1..e1]
in
   let  L  = e2'(I1)
        I2 = extract(range1'(L),1)
        I1 = extract(expand(I1,L),1)
   in
      ...
        let  L    = en'(I1,...,In-1)
             In   = extract(range1'(L),1)
             In-1 = extract(expand(In-1,L),1)
           ...
             I1   = extract(expand(I1,L),1)
        in
             τ(e,I1,...,In)
```

where $\tau(e,I_1,\ldots,I_n)$ yields I_k if e is i_k and yields $\#I_n$ copies of e if e is constant. The function `range1`(n) yields `[1..`n`]` and `range1'` is its data-parallel version. As applied in this rule, `extract` is flattening nested sequences for use in the next step of the translation.

5. Example

We will give a specific example of the transformations applied to a function. Only portions of the functions that are modified by the application of a rule are shown.

```
[i ← [1..5]: sqs(i)]
```

with the definition for *sqs* from section 2.

```
fun sqs(n) = [i ← [1..n]: i*i]
```

1. Rewriting top-level expression

```
    [i ← [1..5]: sqs(i)]
```

≡ {R2 applied to `sqs`}

```
    sqs'([i ← [1..5]: i])
```

2. Definition of `sqs'`.

```
fun sqs'(V) = if #V == 0  then []:seq(int)
                          else [i ← V: [j ← [1..n]: j*j]]
```

≡ {R1: applied V} { for simplicity, only the **else** clause will be displayed }

```
   else let  R = V
         in
              [i ← [1..#R]: ([j ← [1..n]: j*j])$^{n}_{R[i]}$ ]
```

≡ {Results of substitution and switching to infix notation using only the **in** clause}

```
         in
            [i ← [1..#R]:[j ← [1..R[i]] :mult((j,j)) ]]
```

≡ {R2: applied to mult}

```
   [i ← [1..#R]:mult'([j ← [1..R[i]]:(j,j)])]
```

≡ {R2: applied to mult'}

```
   mult''([i ← [1..#R]:[j ← [1..R[i]]:(j,j)]])
```

≡ {R5: applied twice to the tuple}

```
   mult''([i ← [1..#R]:[j ← [1..R[i]]:j]],
             [i ← [1..#R]:[j ← [1..R[i]]:j]])
```

≡ {T1 applied to mult''}

```
   let  K = ([i ← [1..#R]:[j ← [1..R[i]]:j]],
              [i ← [1..#R]:[j ← [1..R[i]]:j]] )
   in
      insert (mult'(extract (K,1),K,1))
```

≡ {T2 applied to one of the nested iterators}

```
   let  K = ( let  I1 = [1..#R]
              in
                   let  L = index'(R,I1)
                        I2 = extract(range1'(L),1)
                        I1 = extract(expand(I1,L),1)
                   in
                        τ(j,I1,I2) , ... )
```

Now the data-parallel version of *sqs* has been fully defined in terms of operations in **V**, and can be executed in parallel.

6. Discussion

Status

An early, problem-specific form of the transformations was implemented in KIDS and was used to transform a simple data-parallel computation [MNP+92], although the final translation step was performed manually. We are currently implementing the general transformation rules and constructing CVL versions of the basic data-parallel operations.

Optimizations

Because they are so frequently applied, it is critically important that the insert/extract operations have minimal overhead. The selection of the tree vector structure for nested sequences was chosen specifically because those operations on this representation can be implemented with a pointer re-assignment.

Through early evaluation of expressions, collisions normally incurred by using expanded (replicated) indices as fetch targets, can be avoided. Each expression within a generator that depends on bound variables can be evaluated as soon as those values are bound and the results can then be expanded, making them readily available for subsequent operations.

Conclusions

We have generated a simple transformational framework for achieving fully parallel execution of a surprisingly large class of high-level, data-parallel programs. The efficiency of our approach remains to be established, but with careful choice of nested sequence representation, we believe that efficient parallel execution is achievable. In view of the simplicity of the transformation rules, we also believe it will be easy to use the rules in conjunction with other, possibly orthogonal, directed transformations.

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.

[CBZ90] Chatterjee, S., Blelloch, G., Zagha, M., "Scan Primitives for Vector Computers", *Proceedings Supercomputing '90*, IEEE , 1990.

[HQ91] Hatcher, P., Quinn, M., *Data-Parallel Programming on MIMD Computers*, MIT Press, 1991.

[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.

[Magó79] Magó, G., "A Network of Computers to Execute Reduction Languages." *International Journal of Computer and Information Sciences*, 1979.

[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) pp.38-50.

[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 Function Programming* ACM SIGPLAN/SIGACT/SIGART, 1986.

Appendix

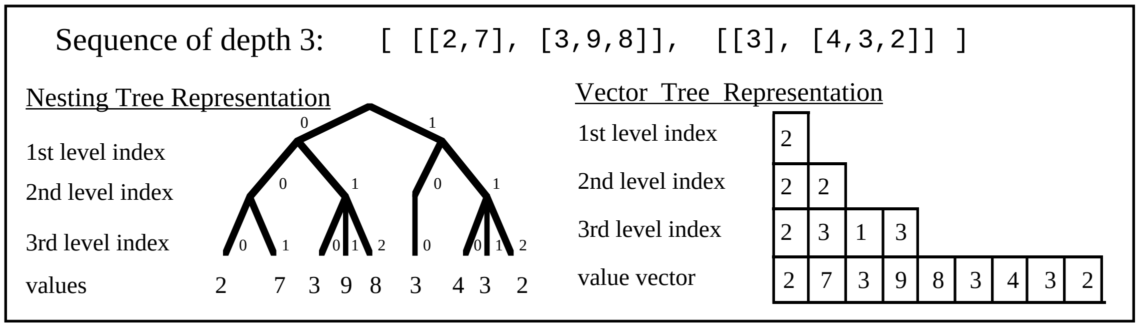

Figure 1.

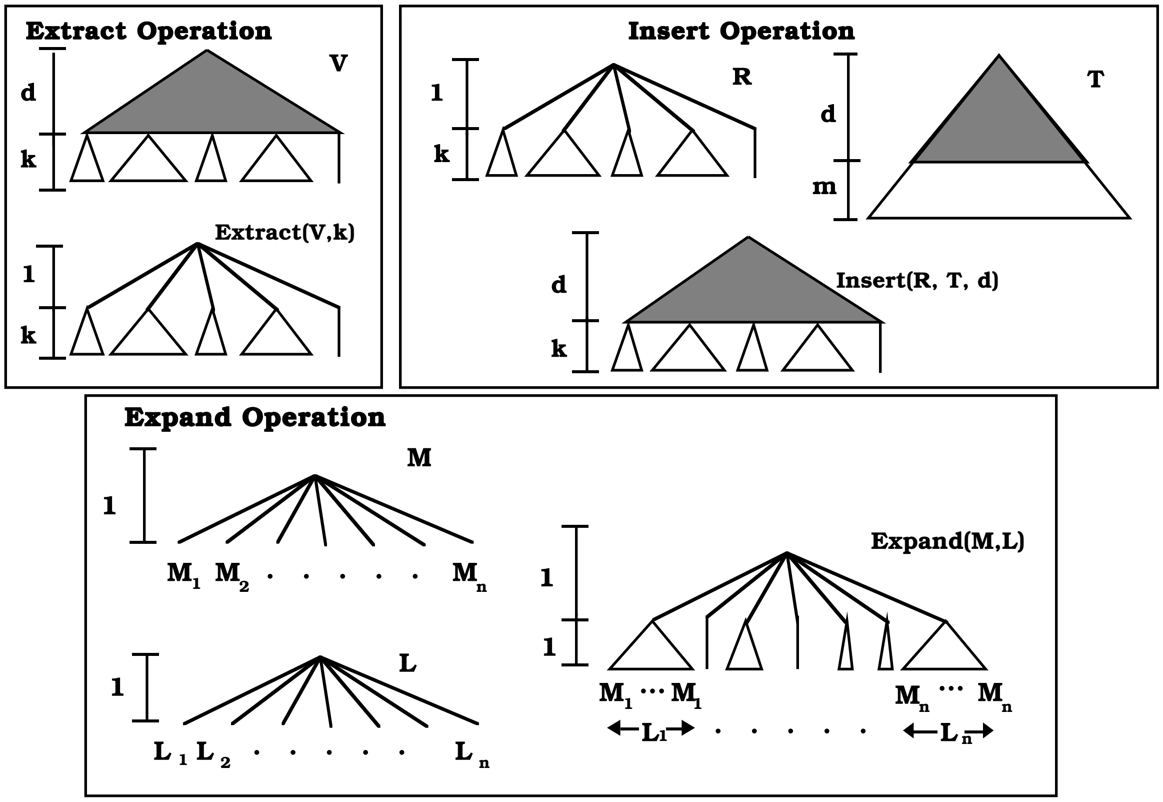

Figure 2.

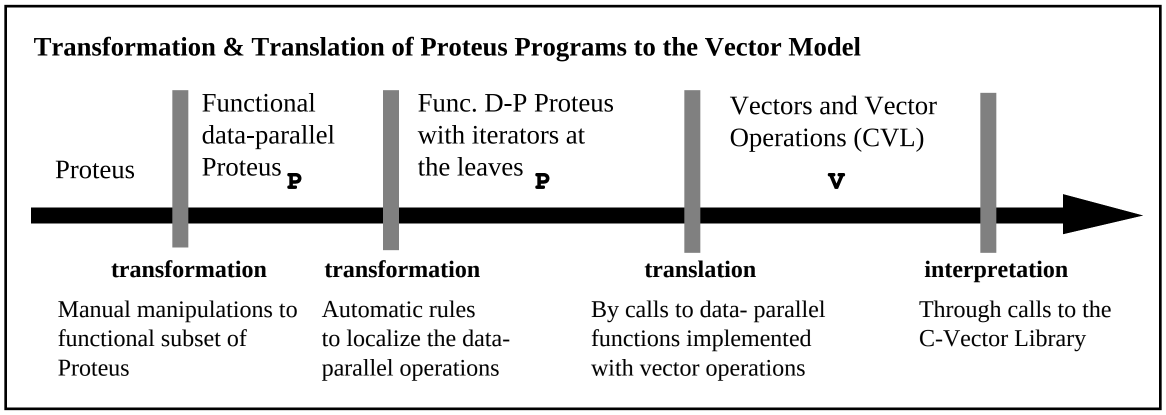
Transformation & Translation of Proteus Programs to the Vector Model
Proteus
Functional data-parallel Proteus P
Func. D-P Proteus with iterators at the leaves P
Vectors and Vector Operations (CVL) V
transformation
Manual manipulations to functional subset of Proteus
transformation
Automatic rules to localize the data-parallel operations
translation
By calls to data- parallel functions implemented with vector operations
interpretation
Through calls to the C-Vector Library

Figure 3.

$$\begin{tikzcd}
V \arrow[r, "f^{k}(V)"] \arrow[d, "\texttt{Extract}(V,k\texttt{-1})"'] & S \\
W \arrow[r, "f'(W)"'] & R \arrow[u, "\texttt{Insert}(R,V,k\texttt{-1})"']
\end{tikzcd}$$

Figure 4.