Compiling Collection-Oriented Languages onto Massively Parallel Computers

Guy E. Blelloch
Carnegie Mellon University
and
Gary W. Sabot
Thinking Machines Corporation

July 17, 1989

Abstract:

This paper introduces techniques for compiling the nested parallelism of collection-oriented languages onto existing parallel hardware. Programmers of parallel machines encounter nested parallelism whenever they write a routine that performs parallel operations, and then want to call that routine itself in parallel. This occurs naturally in many applications. Most parallel systems, however, do not permit the expression of nested parallelism. This forces the programmer to exploit only one level of parallelism or to implement nested parallelism themselves. Both of these alternatives tend to produce code that is harder to maintain and less modular than code described at a higher-level with nested parallel constructs. Not permitting the expression of nested parallelism is analogous to not permitting nested loops in serial languages.

This paper describes issues and techniques for taking high-level descriptions of parallelism in the form of operations on nested collections and automatically translating them into flat, single-level parallelism. A compiler is presented that translates a subset of a collection-oriented language, PARALATION LISP, into the instruction set of a flat virtual machine. The instructions of the virtual machine are simple instructions on vectors of atomic values, and can be implemented on broad class of target architectures, including vector machines, single instruction parallel machines, and multiple instruction parallel machines. We have implemented the instructions on the Connection Machine computer (CM-2), a massively-parallel, single-instruction computer.

As an illustration of the compiler techniques, the paper presents a quicksort example. The example has been tested on the CM-2. The speed of the compiled sort is only a factor of three slower than that of the fastest CM-2 sort.

Keywords: Paralation Lisp, Scan, Compiler, Connection Machine, Parallel Programming, Nested Parallelism, Vector Models.

1 Introduction

Most currently implemented parallel programming models allow only a single level of parallelism [10]. For example, a user could define a parallel `draw-line` routine that given two endpoints determine all the pixels that lie on the line between the points. This routine, however, cannot itself be called in parallel; the user cannot draw multiple lines in parallel.

This paper is concerned with programming models that allow multiple levels of parallelism: *nested parallelism*. In the line drawing example, this would allow the user to define a procedure, `draw-object`, that draws in parallel all the lines that make up an object by calling `draw-line` in parallel. The `draw-object` procedure would exploit two kinds of parallelism: one within the subroutine `draw-line`, and the other from executing many `draw-lines` in parallel. The procedure `draw-object` might further be called in parallel by a routine `draw-scene`.

Nested parallelism occurs naturally in many problems, and a programming model that can express nested parallelism allows a very concise description of such problems. This paper presents techniques that transform nested parallelism into flat, one-level parallelism. This makes it straightforward to map programming models that allow nested parallelism, onto hardware that naturally only accommodates a single level. Such hardware includes vector computers and homogeneous parallel computers.

The techniques considered are useful for a class of languages we call *collection-oriented* languages. Collection-oriented languages, such as Paralation Lisp [12], CM-Lisp [15], SETL [13], NIAL [11], SQL [5], APL [8,9] or APL2 [1], are based on data structures which represent collections of elements, and operations for manipulating the collections as a whole, such as multiplying two arrays or taking the intersection of two sets. This paper is concerned with the subclass of these languages that allow nested collections and the expression of nested collection operations. In such languages, each element of a collection can itself be a collection, and any function defined to operate on a collection can be applied over the elements of a nested collection. Of the above mentioned languages, all but APL and SQL allow such nesting. Many of the *nested collection-oriented* languages were not designed for execution on parallel computers, but their constructs often map naturally onto parallel architectures.

Although the techniques discussed in this paper could be used for many nested collection-oriented languages, for the sake of concreteness, the techniques will be presented in the framework of a compiler for a particular nested collection-oriented language, Paralation Lisp [12], and will compile to a particular flat collection-oriented instruction set, SV-Lisp. Paralation Lisp consists of a new data structure

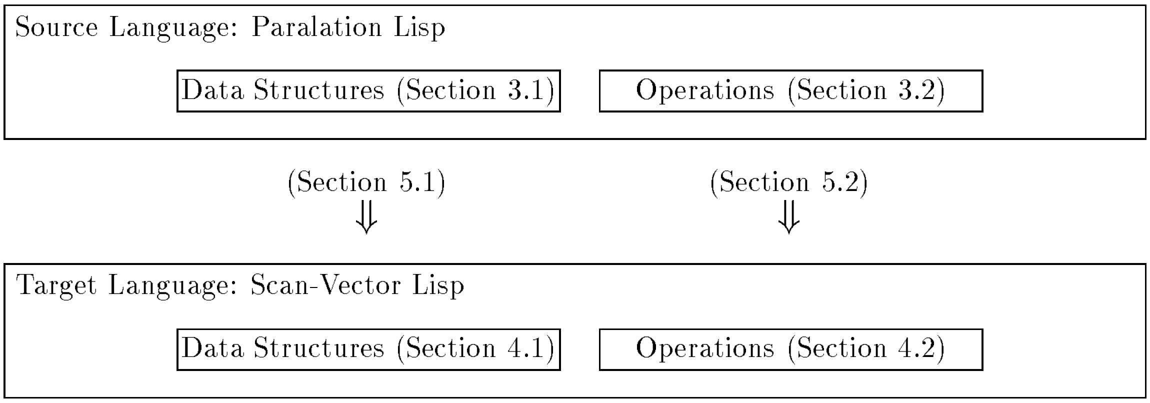

Figure 1: Organization of the compiler. To translate PARALATION LISP onto scan-vector lisp both the data structures and the operations must be translated.

and a small set of operators that are added to COMMON LISP. The data structure, the *field*, is an ordered collection much like an array, which can be nested. The operators include the `elwise` form that applies any code, including another `elwise`, over all the elements of a field, and a handfull of operators that are used to rearange the elements of a field. SV-LISP consists of a subset of COMMON LISP with the addition of a small set of instructions for manipulating flat (non nested) vectors of atomic values. These instructions, for example, include permuting the elements of a vector or elementwise adding two vectors.

Figure 1 illustrates the organization of the compiler discussed in this paper. The discussion of the compiler is broken into two main parts: the mapping of PARALATION LISP data structures onto SV-LISP data structures and the translation of PARALATION LISP code onto SV-LISP code. Figure 2 illustrates how the compiler fits into a larger system that compiles into the instruction set of an actual target machine, the CM-2. Translating PARALATION LISP to SV-LISP rather than directly onto a parallel machine, such as the CM-2, allows us to separate the novel techniques from standard compiler techniques, and makes it relatively easy to port the compiler to new machines. The vector instructions of SV-LISP were chosen because they can be further translated to a large variety of architectures, including vector machines such as the CRAY or Convex machines, single instruction parallel machines such as the CM-2, and multiple instruction parallel machines such as the Alliant.

The purpose of implementing this compiler was to test our techniques for flattening nested parallelism. The compiler is therefore by no means complete: it only supports a small, but important, subset of PARALATION LISP. The compiler does, however, produces efficient output code. The CM-2 code generated by the quicksort

$$\boxed{\text{PARALATION LISP}}$$
$$\Downarrow \text{ This paper}$$
$$\boxed{\text{SV-LISP with Segmented Instructions}}$$
$$\Downarrow \text{ See [3]}$$
$$\boxed{\text{SV-LISP without Segmented Instructions}}$$
$$\Downarrow \text{ A COMMON LISP compiler}$$
$$\boxed{\text{Vector and Host Machine Instructions}}$$
$$\Downarrow \text{ See [3]}$$
$$\boxed{\text{CM-2 Instructions}}$$

Figure 2: This figure shows the organization of the full compiler that translates PARALATION LISP into CM-2 instructions. This paper focuses on the first step.

example described in Section 2 and Section 6 is only a factor of three slower than the fastest existing sorting routine for the CM-2 (an optimized radix sort written in PARIS, the parallel instruction set of the Connection Machine). The generated code is also faster than a version of quicksort implemented in *LISP [16] (one of the two supported languages for the CM-2) and compiled using the *LISP compiler. *LISP does not allow nested constructs, so the code is also significantly more complex (the description of segments and the manipulation of these segments must be included in the code).

The compilation of a quicksort algorithm will be used as the central example in this paper. After introducing the quicksort example, the paper defines the source (PARALATION LISP) and target (SV-LISP) languages in detail, describes the transformations performed by the compiler, and then returns to the quicksort example to illustrate the actual transformations.

The most interesting and important elements of the discussed compiler are:

- The representation used for the PARALATION LISP collections.

- The creation of two versions of each function, one to be called at top level and one to be called nested.

- The stepping-up and stepping-down manipulations used at the entry and exit of the PARALATION LISP apply-to-each form.

- The implementation of nested conditionals—perhaps the most elusive aspect of flattening nested parallelism.

2 Quicksort Example

This section introduces a quicksort example [7]. This parallel quicksort example illustrates the use of nested parallelism, gives a basic idea of the constructs of PARALATION LISP, and introduces some of the techniques described in the remainder of the paper. Section 6 returns to this example and shows the actual code generated by translating the nested PARALATION LISP quicksort into a flat SV-LISP quicksort.

The basic data structure is a sequence, an ordered collection of values. If the sequence is already sorted, quicksort returns the sequence. Otherwise, a pivot element is picked at random, and the sequence is partitioned into two disjoint sequences. One contains elements that are smaller than the pivot, and the other contains elements that are greater-or-equal to the pivot. `qsort` is called recursively to sort the two partitioned sequences, and the results are appended to produce the result.

To take a closer look, we can step through the execution of the actual PARALATION LISP quicksort code in Figure 3. Only the results of the program will be discussed in this example; the language constructs are explained from a more general point of view in the next section. The program is written in a verbose style to facilitate line by line analysis (that is, temporary results are saved in named variables even though they are only needed once).

The `not-sorted-p` function tests if the keys are already sorted and returns `t` if they are not. It can be implemented by shifting the keys over by one, executing a comparison and taking an or of the results. This is the base case (the termination condition) of the recursive `qsort` function.

The first step of `qsort` is to check for its base case in line 2. If the keys are sorted line 16 returns `keys` as the result. In general, the test for the base case fails, and a random `pivot-value` is then picked as the second step of the algorithm. For example, if `qsort` is called on:

keys = `[7 9 2 11 19 6 12]`

`not-sorted-p` would return `t` and perhaps 9 would be selected as the random pivot value. Next, lines 5–6 calculate a flag value, `side`. Each value compares itself to the pivot value, returning nil if it is less than the pivot and t otherwise:

```
0 (defun qsort (keys)
1   ;; If not base case, do sort step, else return keys
2   (if (not-sorted-p keys)
3       (let* ((pivot-value (elt keys (random (length keys))))
4              ;;each element picks a side of the pivot partition to go to
5              (side
6                (elwise ((key keys)) (< key pivot-value)))
7              ;; create ordered collection of two ordered subcollections
8              ;; based on the value of side
9              (sub-data
10               (collect keys :by (collapse side)))
11             ;; recursively sort the subcollections
12             (sorted-sub-data
13               (elwise (sub-data) (qsort sub-data))))
14        ;; append the sorted subcollections
15        (expand sorted-sub-data))
16      keys))
```

Figure 3: Quicksort in PARALATION LISP

side = `[nil t nil t t nil t]`

In the next step, lines 9–10, paralation library functions are used to split the keys into two collections based on the two distinct values (nil and t) in `side`. Each collection is nested in a larger collection:

sub-data = `[[7 2 6] [9 11 19 12]]`

At this point, although a nested collection is being used, they do not seem necessary. A flat vector of the split numbers, along with some representation of the lengths of the subcollections, captures the state information equally well. In fact, that is the basis of our compiler's representation of nested collections. Simply put, a collection of collections can be represented by two collections: one containing the data and another containing the lengths of the subsets. For example, `sub-data` can be represented with the following two flat vectors:

```
[7 2 6 9 11 19 12]
[3 4]
```

In the next step, it becomes clear why it is desirable to hide this representation from the user: Lines 12–13 contain a parallel recursive call that creates yet another level of collection nesting. For example, the recursive `qsort` call that operates on the left half might cause a split based on a pivot value of 7 while the other call might split based on a pivot value of 12:

call 1's sub-data = `[[2] [7 6]]`
call 2's sub-data = `[[9 11] [19 12]]`

Viewed together from the macroscopic viewpoint, the two parallel recursive calls operate on a collection of collections of collections, and the nesting of collections grows a level deeper with each recursion, even as the number of `qsort` invocations that are executing in parallel doubles. A programmer using a collection oriented language does not have to be concerned about such details, any more than a C programmer must worry about allocating automatic (local) variables on a stack. Our compiler, and programmers using vector models directly, must deal with these issues explicitly.

Inductively assuming that `qsort` works, we can return to line 15 of the top level call to `qsort`. It uses a library function to transform a collection of sorted subcollections:

sorted-sub-data = `[[2 6 7] [9 11 12 19]]`

into the final result by flattening the collection of collections into a single collection.

qsort result = `[2 6 7 9 11 12 19]`

2.1 Two Kinds of Parallelism

We now consider how this quicksort can be implemented to run in parallel. We consider two types of parallelism. First, the `value-count` routine, the comparison of the `pivot-value`, the `collect` and the `collapse` can all be implemented in parallel as discussed later in Section 5.2. We call this the *intraroutine parallelism.* This type of parallelism seems natural for vector or SIMD architectures. The second kind of parallelism occurs in line 13, where quicksort is called recursively twice. Each of these calls can run in parallel. We call this the *interroutine parallelism.* It seems more suited to coarse-grained MIMD architectures.

If we took advantage of intraroutine parallelism but ignored interroutine parallelism, the code would execute rapidly in the first stages, where the vectors of data to be sorted are large, but would become very inefficient in the final stages of quicksort. There would be many invocations of quicksort that would have to be run separately, and each would be operating on a vector that was small compared to the number of processors available. The original PARALATION LISP compiler and the CM-LISP compiler both took this inefficient, but easy to implement, approach.

On the other hand, one can imagine an implementation on a coarse grained machine that only took advantage of the interroutine parallelism. Such an approach would be efficient in the later stages of quicksort, but it would be inefficient in the early stages. Small numbers of processors would be responsible for operating on relatively large vectors.

An important goal of the compiler discussed in this paper is to take advantage of both kinds of parallelism in a efficient and simple way and map them onto a strictly SIMD model. The compiler does this and compiles the quicksort into a routine which has an expected complexity of $O(\lg n)$ calls to a set of simple data parallel operations such as permute, scan, and elementwise arithmetic and logical operations.

3 Source Code: Paralation Lisp

This section summarizes the PARALATION LISP language; for more details the reader should see [12]. PARALATION LISP consists of a new data structure, three primitive operators, and a set of other operators built on the primitive operators, all added to COMMON LISP.

3.1 Data Structures

The data objects permitted in PARALATION LISP are all the standard COMMON LISP data objects with one additional object, the *field*. The *field*, is a linear-ordered collection of elements. A field can be heterogeneous and the elements can be any PARALATION LISP value—including another field—thus allowing nested collections. Here are some examples of fields.

A homogeneous field: `#F(7 9 2 11 19 6 12)`

A nested homogeneous field: `#F(#F(4 8 3) #F(9 1 12 7) #F(2 9))`

A heterogeneous field: `#F(7 #F(4 Nil 3) T "horse")`

A structure field: **#F** $\left(\boxed{\begin{matrix} u : u_{00} \\ v : v_{00} \end{matrix}} \; \boxed{\begin{matrix} u : u_{01} \\ v : v_{01} \end{matrix}} \; \boxed{\begin{matrix} u : u_{02} \\ v : v_{02} \end{matrix}} \right)$

3.2 Operators

The three additional primitive operators are an iteration operator and two field operators. The iteration operator, elwise, is used to iterate any PARALATION LISP code, including another elwise, over all elements of a field. The two primitive field operators, match and <-, perform communication among the elements of fields: match encapsulates a communication pattern into a *mapping*, and <- transfers a field according to a *mapping*. Several other operations are supplied by PARALATION LISP but can be defined in terms of match and <-. All the COMMON LISP sequence functions can also be used on fields.

The operators of PARALATION LISP that are needed for the compilation examples are outlined below. The ideas behind paralations and mappings, which are both important concepts of the language, are not discussed because they are not germane to a discussion of compiler issues.

Elwise: The elwise operator is used to apply a body over each element of a field, or set of fields.

```
(elwise bindings
   body)
```

executes the body for each element of each field in the bindings and execute the body. So, for example, the form:

```
(elwise ((a A)
         (b B))
   (+ a b))
```

will pairwise add the elements of A and B and return a field of the results (the body in this example is simply a function call to +). Each binding of an elwise must be from the same paralation; the result returned by elwise is a new field in that same paralation. The elwise body can include any valid PARALATION LISP form.

Match and Move: The match operation takes two key fields as arguments; one from a source paralation and one from a destination paralation. It returns a *mapping*. A mapping can be thought of as a bundle of one-way arrows that connect certain sites of the source paralation to certain sites of a destination paralation. Two sites are connected if their key field values are equal. A mapping is an encapsulated communication pattern.

The <- (move) function accepts a mapping and a field from the source paralation of the mapping as its arguments. <- simply pushes this source data field into the tails of the mappings arrows, causing a field in the destination paralation to pop out at the other end of the mapping. When several arrows leave a single source site, a concurrent read takes place. When many arrows collide at a single destination site, the multiple incoming values are reduced into a single value by repeatedly applying a user-specified, two-argument combining function. (Combining will not be needed for the examples presented in this paper, but it is an important part of the paralation model.) Finally, when a destination site receives no incoming values, a value is taken from a user-specified default field in the destination paralation. match creates mappings which describe communication patterns; `<-` makes use of mappings, and includes the functionality of both concurrent read and combining.

We now outline some of the operations that can be defined on top of the match and `<-` operations.

Vref: The vref operation "sums" the elements of a field according to any binary operator. So, for example:

```
(vref #F(7 4 1 11 2 6) :with 'max)
```

$\Rightarrow$ `11`

Collapse and Collect: The collapse operator takes a set of *keys* and generates a mapping in which all elements with equal valued keys are mapped to the same position. In the case of a collapse of a field containing numbers the positions are ordered by the relative values of the keys. In the case of a collapse of a field containing booleans, there are only two destination positions and the position containing `nil` appears before the position containing `t`.

The collect operator takes a mapping and a field and appends all the elements which are mapped into the same position into a subfield. This can be implemented using a `<-` with a combiner of concatenate. The collect operator returns a field of fields.

As an example of collect and collapse consider the following operation:

(let ((A #F(a_0 a_1 a_2 a_3 a_4 a_5))
      (B #F(k_0 k_1 k_0 k_2 k_1 k_1)))
  (collect A :by (collapse B)))

$\Rightarrow$ #F(#F(a_0 a_2) #F(a_1 a_4 a_5) #F(a_3))

Expand: The expand operator takes a field of fields and appends all the subfields into a single field. So, for example, expand applied to the *result* field given above returns:

```
(expand (collect A :by (collapse B)))
```

$\Rightarrow$ `#F(`a_0 a_2 a_1 a_4 a_5 a_3`)`

3.3 Restrictions

The compiler implements a small enough subset of PARALATION LISP that the subset is more concisely described by what it does include rather than what it does not include.

The subset only supports homogeneous fields and the data type of each elements of a field must be either an integer, boolean, field or structure. Since the elements can be fields, the subset supports nested fields. Many other data types, such as floating-point numbers or characters, would be easy to add but were left out for the sake of simplicity.

The subset supports the following sequence operations on fields: elt, length, sort, reduce, and concatenate. Other sequence operators would be easy to add but these were the only ones we needed for our test code.

The subset supports the following operations of PARALATION LISP: elwise, collect and collapse. (It would be relatively straighforward to add match and <-.) The subset supports most of the operations on integer and boolean values inside an elwise. It also supports nested operations on fields. For example, any of the above mentioned sequence functions can appear in an elwise. The only conditional that the subset supports is the *if* special form, and it places the restriction that the results returned from both the then-expression and the else-expression must be of the same type. Section 7 discusses compiler extensions that remove some of these restrictions.

4 Target Code: Scan-Vector Lisp

We now describe the target code of the compiler, SV-LISP (Scan-Vector Lisp) [3]. SV-LISP is a small subset of COMMON LISP with the addition of a new data type, the *pvector*, and a set of instructions for manipulating this data type, the *pvector instructions*. The pvector data type—a vector of atomic values—is more primitive than the *field* data type of PARALATION LISP since it neither allows nested collections nor collections of structures. Likewise, the pvector instructions—which include pairwise adding the elements of two pvectors or permuting the atomic elements of a pvector—are more primitive than the collect, collapse and extract instructions of PARALATION LISP.

The pvector data type is mapped onto a parallel machine by placing the elements of each vector evenly balanced across the processors. When executing a pvector instructions, each processor is responsible for its own elements (see [3] for more details). Many of the techniques considered for compiling APL onto both serial and parallel machines [4,6] would also be useful here.

SV-Lisp includes a set of segmented instructions which are key to the implementation of nested parallel routines. These instructions, in effect, can operate over many sets of data by partitioning a vector into independent, virtual segments. The innermost parallelism of a nested parallel routine is available from within each segment and the outer levels of parallelism are available since a vector can contain any number of segments.

4.1 Data Structures

SV-Lisp has five data types. Three of these come from Common Lisp: *integers, booleans* and *structures*. Two of them are new: *boolean pvectors* and *integer pvectors*. Pvectors are arbitrarily long linear-ordered collections of either completely boolean or completely integer values, respectively.[1] Every pvector can have a different length and the only operations that can create or manipulate the pvector data types are the pvector instructions discussed in Section 4.2. To implement a complete Paralation Lisp rather than the subset discussed in this paper, SV-Lisp would need to be augmented with other types such as floating-point numbers and floating-point pvectors.

4.2 Operations

Figure 4 lists the operations of SV-Lisp. The operations are separated into three major classes, operations from Common Lisp, pvector instructions and segmented pvector instructions. The Common Lisp operations are defined in the Common Lisp reference manual [14]. The pvector instructions are further separated into four subclasses. The *elementwise* instructions execute an elementary arithmetic or logical operation, such as $+, -, *$, `or`, and `not`, over the indices of the input vectors (see Figure 5). The *permutation* instructions permute the elements of a vector based on another vector of indices. The *scan* instructions execute a prefix operation on a vector: each element receives the "sum" of all previous elements.[2] The *vector-scalar*

[1]The term pvector is used instead of vector so as not to confuse it with the Common Lisp vector data type—a linear-ordered collection whose elements can be of any type.

[2]The name scan comes from APL, but in APL the scan includes the value itself, while in SV-Lisp it does not.

	COMMON LISP Operations
Special Forms:	if, defstruct, defun, let, let*, progn, setq
Scalar Operations:	$+, -$, and, or, $=, <$, ...
	Pvector Instructions
Elementwise:	p+, p−, p-and, p-or, p= , p<, p-select, ...
Permutation:	permute, select-permute
Scan:	+-scan, max-scan, min-scan, or-scan, and-scan
Vector-Scalar:	insert, extract, distribute, length
	Segmented Pvector Instructions
Permutation:	p-permute, p-select-permute
Scan:	p-+-scan, p-max-scan, p-min-scan, p-or-scan, p-and-scan
Vector-Scalar:	p-insert, p-extract, p-distribute, p-length

Figure 4: The operations of SV-LISP.

instructions are used to create vectors and to extract scalar elements from a vector.

Each segmented pvector instruction operates independently on a set of nonoverlapping, contiguous segments of its vector arguments. A vector is partitioned into segments by keeping a second vector which specifies the length of each segment (see Figure 6). The segmented versions of the pvector instructions take pairs of vectors for each argument: the value vector and the segment descriptor. The segmented version of the permutation primitive bases its indices relative to the beginning of each segment so values permute within a segment. The segmented version of the scan instructions restart at the beginning of each segment. Elsewhere we show that all the segmented instructions can be implemented with a small constant number of calls to the unsegmented versions [3].

It is important to realize that when mapping segmented vectors onto a parallel machine both the value vector and the segment descriptor vector are evenly balanced across the processors. The mapping need not know that the two parts belong to a segmented vector. For example, in the segmented vector of Figure 6, if we had four processors, the elements a_0 and a_1 would be placed in the first processor a_2 and a_3 in the second, a_4 and a_5 in the third, and a_6 and a_7 in the fourth. The three lengths 2, 4 and 2 would be placed in the first three processors.

The Elementwise Instructions

A	=	[5	1	3	4	3	9	2	6]
B	=	[2	5	3	8	1	3	6	2]
A p+ B	=	[7	6	6	12	4	12	8	8]
A p$\times$ B	=	[10	5	9	32	3	27	12	12]

The Permute Instruction

A (data vector)	=	[a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7]
I (index vector)	=	[2	5	4	3	1	6	0	7]
permute(A, I)	=	[a_6	a_4	a_0	a_3	a_2	a_1	a_5	a_7]

The Scan Instructions

A	=	[5	1	3	4	3	9	2	6]
+-scan(A)	=	[0	5	6	9	13	16	25	27]
max-scan(A)	=	[0	5	5	5	5	5	9	9]

The Vector-Scalar Instructions

A	=	[a_0	a_1	a_2	a_3	a_4	a_5]

V = v I = 3 L = 5

extract(A, I)	=	a_3				
distribute(V, L)	=	[v	v	v	v	v]

Figure 5: Examples of the pvector instructions. For the permute instruction it is an error for more than one element to contain the same index. Not shown is the select-permute instruction which permutes elements between vectors of different lengths by masking out certain elements and placing defaults in certain positions.

Vector	=	[a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7]
Segment Descriptor	=	[2	4	2]					
Segmented Vector	=	[a_0	a_1]	[a_2	a_3	a_4	a_5]	[a_6	a_7]

A	=	[5	1]	[3	4	3	9]	[2	6]
B	=	[1	0]	[2	0	3	1]	[0	1]
I	=	[0	3	1]					
S (segment descriptor)	=	[2	4	2]					
p-+-scan(A)	=	[0	5]	[0	3	7	10]	[0	2]
p-permute(A, B)	=	[1	5]	[4	9	3	3]	[2	6]
p-extract(A, I)	=	[5	9	6]					

Figure 6: Examples of a segmented vector and the segmented versions of the pvector instructions. In the calls to the segmented instructions, the variables A and B are structures of two vectors, the value vector and the segment descriptor.

5 Translation

This section discusses how PARALATION LISP is translated into SV-LISP. In keeping with the rest of the paper, it first describes data structures and then describes operations.

5.1 Data Structures

This section discusses how the compiler maps the collections of PARALATION LISP, *fields*, onto the primitive data structures of SV-LISP, *pvectors*. The mapping discussed allows a particularly efficient manipulation of nested fields by the vector instructions of SV-LISP. The representation of nested fields is based on segments as introduced at the end of Section 4 and allows the compiled code to operate over all subfields in parallel.

All fields are constructed from the *pfield* structure—a COMMON LISP structure type with two slots. The first slot stores a *segment-descriptor*, which describes the segmentation of the field (see Section 4).[3] The second slot stores the actual values. This slot contains a pvector if the field contains only atomic values, contains another

[3]In the actual compiler the segment-descriptor contains several descriptions of the segmentation, each useful in different contexts. For the purposes of this paper, we assume the segment-descriptor only contains the lengths of each segment.

pfield if the field is nested, and contains a user-defined structure if the field is a field of user-defined structures. Each of these cases is discussed below. Since the subset of Paralation Lisp considered only supports homogeneous fields, heterogeneous fields are not considered.

Simple Field: To represent a simple field—a field whose elements are all atomic—we use a single *pfield* structure. The first slot contains a definition of a single segment—its length. The second slot contains a pvector with the values of the field. For example:

#F(a_0 a_1 a_2 a_3 a_4) $\longrightarrow$

> *pfield*
> segdes: [5]
> values: [a_0 a_1 a_2 a_3 a_4]

We use a *pfield* structure instead of using a pvector directly since it allows us to check if two equal length fields belong to the same paralation.[4] Using the *pfield* structure also permits a more homogeneous implementation of the stepping-up and stepping-down manipulations to be discussed in Section 5.2.

Nested Field: We represent a nested field—a field whose elements are themselves fields—by nesting the *pfield* structures and using segments of a single pvector to represent each subfield. For example:

#F(#F(a_{00} a_{01}) #F(a_{10} a_{11} a_{12}) #F(a_{20}))

$\Downarrow$

> *pfield*
> segdes: [3]
> values:
> > *pfield*
> > segdes: [2 3 1]
> > values: [a_{00} a_{01} a_{10} a_{11} a_{12} a_{20}]

In this example, the *segdes* slot of the inner *pfield* describes the segmentation of the *values* slot. This technique can be applied recursively to represent a nesting of any depth. A field nested n deep can be represented with n segment-descriptors and n *pfield* structures.

[4]Since the segment-descriptor is actually a structure, we can check if two fields are from the same paralation by seeing if the two segment-descriptors are `eql`.

Structure Field: We represent a structure field—a field whose elements are each a user defined structure—by pulling the structure out from inside the field. For example:

$$\mathtt{\#F}\left(\boxed{\begin{array}{l} u:\mathtt{u}_{00}\\ v:\mathtt{v}_{00}\end{array}}\;\boxed{\begin{array}{l} u:\mathtt{u}_{01}\\ v:\mathtt{v}_{01}\end{array}}\;\boxed{\begin{array}{l} u:\mathtt{u}_{02}\\ v:\mathtt{v}_{02}\end{array}}\right) \rightarrow \boxed{\begin{array}{ll} \textit{pfield} & \\ \quad \text{segdes:} & \mathtt{[3]} \\ \quad \text{values:} & \boxed{\begin{array}{l} \textit{uv-structure} \\ \quad \text{u:}\ \mathtt{[u_0\ u_1\ u_2]} \\ \quad \text{v:}\ \mathtt{[v_0\ v_1\ v_2]} \end{array}} \end{array}}$$

In this example, the field of three uv-structures is mapped onto a single uv-structure whose slots contain a pvector with the values of all three of the original uv-structures.

Mappings in Paralation Lisp can be represented in canonical form as a pair of integer vectors: the first vector are indices into the destination from the source, and the second are indices into the source from the destination (see [12] for more details).

5.2 Operations

This section discusses the manipulations necessary to translate code from the subset of Paralation Lisp into SV-Lisp. The discussion is broken into four parts: (1) compiling two versions of all code, one parallel and one serial; (2) translating the elwise form; (3) transforming conditionals; (4) and implementing the Paralation Lisp collection operations.

5.2.1 Compiling Two Versions

When a function is defined in Paralation Lisp, it must work both if called at top level (not within an elwise), and if called within an elwise. Consider the example of Figure 7. In the first case, the compiler uses a serial version of `plus-times` while in the second case the compiler uses a parallel version of the routine. The serial version uses the standard Common Lisp `+` and `*` operations while the parallel version uses the pvector primitives `p-+` and `p-*`. Figure 8 illustrates an example of the translation of a Paralation Lisp routine into the two SV-Lisp routines.

The compiler keeps two versions of every user-defined function and every function supplied by Paralation Lisp: the top level version and the replicated version (the version called inside a nested parallel form). When compiling the replicated version of a new function, the function calls inside the routine are simply replaced with their replicated version (see Figure 8). The *special forms*, however, cannot in general be

```
(defun plus-times (a b)
  (* a (+ a b)))
```

Called at top level

```
(plus-times 5 2)

 ⇒ 35
```

Called within an elwise

```
(elwise ((a #F(2 1 6))
         (b #F(3 5 2)))
  (plus-times a b))

 ⇒ #F(10 6 48)
```

Figure 7: Any routine in Paralation Lisp can be called either at top level or within an elwise.

```
(defun plus-times (a b c)
   (* a (+ b c)))
```

$\Downarrow$

```
(defun s-plus-times (a b c)
  (* a (+ b c)))

(defun p-plus-times (a b c)
  (p-* a (p-+ b c)))
```

Figure 8: Compiling both a parallel and a serial version of a routine. The parallel version replaced all function calls with their parallel versions.

replaced with a parallel function. This is because the arguments of special forms in COMMON LISP are not necessarily all evaluated. For example, the `if` form only evaluates the second argument if the first evaluates to `T`. Special forms can therefore require some extra manipulations. (Fortunately, COMMON LISP does not permit user-defined special forms.) The translation of the if special form is discussed in Section 5.2.3. The let, let* and progn special forms require no manipulations.

5.2.2 Compiling Elwise Forms

The compiler applies several manipulations to translate an elwise form. First, it executes the same manipulations required when creating a parallel form of a function as discussed in the previous section. Second, it inserts code that copies all the free variables—variables that appear in the body but not in the binding list—across the elements of the elwise. Third, it inserts code that steps-down all the values bound in the binding list, and steps-up the result of the body.

We first discuss copying free variables. In PARALATION LISP, if a variable appears in the body of an elwise but not in the binding list, the variable in implicitly copied across the elements of the elwise.[5] For example, in the form:

```
(let ((b 3))
  (elwise ((a #F(4 1 2)))
    (+ a b)))

 ⇒ #F(7 4 5)
```

the value of the variable `b` is implicitly copied across the three elements and added to each. When translating from PARALATION LISP to SV-LISP, the translator inserts code to execute this copy. The particular code inserted depends on the type of value that needs to be copied. If the value is a scalar, the distribute pvector primitive (see Section 4.2) is inserted. Figure 9 illustrates an example of this manipulation. If the value is a structure of scalars, a distribute primitive is inserted for each slot of the structure. If the value is a field, a distribute-segment operation is inserted that creates a nested field with the original field in each element. The type of the variable to be copied can often be inferred at compile time so that the correct code can be inserted at compile time (in the above example `b` must be a scalar since it is being added). If the type cannot be inferred at compile time, the compiler inserts code that executes a type dispatch at run time.

[5]This is similar to *scalar extension* in APL [8], but in PARALATION LISP any value will be extended, not just scalars.

```
(elwise ((a A))
  (+ a b))
```

$\Downarrow$

```
(simp-elwise ((a A)
              (b (distribute b (pfield-segdes A))))
  (+ a b))
```

Figure 9: An example of the code inserted for copying free variables. All free variables are removed by this manipulation. The `simp-elwise` form is a version of `elwise` that does not accept free variables.

```
(simp-elwise ((a A)
              (b B))
  (+ a b))
```

$\Downarrow$

```
(let ((a (pfield-values A))
      (b (pfield-values B))
      (current-segdes (pfield-segdes A)))
   (make-pfield
      :segdes current-segdes
      :values (p-+ a b)))
```

Figure 10: An example of the stepping-down and stepping-up manipulations.

We now discuss *stepping-down* and *stepping-up*, which are crucial to the implementation of operations on nested fields. Stepping-down consists of stripping off the top *pfield* from each value being bound in the elwise bindings, and setting a variable called the *current-segdes* to this value. So for a nested field, each time the field is passed inside another elwise another of its *pfield* structures is stripped off. Stepping-up is the inverse of stepping-down. When leaving an elwise, stepping-up consists of tagging on a *pfield* structure to the result returned from the body of the elwise, and restoring the value of the *current-segdes*. Figure 10 illustrates the code inserted by these manipulations.

To see how stepping-down and stepping-up are used, consider the following code:

```
(let ((field-of-fields #F(#F(7 4) #F(11) #F(8 1 17))))
  (elwise ((field field-of-fields))
    (elwise ((value field))
      (+ value value))))
```

$\Rightarrow$ `#F(#F(14 8) #F(22) #F(16 2 34))`

Based on the representation discussed in Section 5.1, the original field is represented as:

field-of-fields =

> *pfield*
> segdes: `[3]`
> values:
> > *pfield*
> > segdes: `[2 1 3]`
> > values: `[7 4 11 8 1 17]`

When entering the outer elwise, the stepping-down code strips off the top *pfield* leaving:

field =

> *pfield*
> segdes: `[2 1 3]`
> values: `[7 4 11 8 1 17]`

And when entering the inner elwise, the next *pfield* is stripped off leaving:

field = `[7 4 11 8 1 17]`

Now when `p-+` is applied to `field`, the result is:

`[14 8 22 16 2 34]`

When exiting the inner elwise the stepping-up code appends a *pfield* back on, returning:

```
pfield
   segdes: [2 1 3]
   values: [14 8 22 16 2 34]
```

And when exiting the outer elwise another *pfield* is appended, returning:

```
pfield
   segdes: [3]
                pfield
   values:         segdes: [2 1 3]
                   values: [14 8 22 16 2 34]
```

Which is the representation of the desired result:

```
#F(#F(14 8) #F(22) #F(16 2 34))
```

In this example, the code that executes the addition runs in parallel over all elements therefore taking advantage of the parallelism within each subfield and also the parallelism among the subfields. This technique works regardless of the depth of the nesting and regardless of the complexity of the operations executed within the elwise.

One way of thinking about what is going on is that the compiler converts an elwise, which is a mapping of a function over many sets of data, into a new "composite" function over one larger set of data—the data sets all appended together. The effect of stripping off a *pfield* by the translated elwise is to remove a level of dividing boundaries and therefore effectively appending the data sets. The "composite" function (the replicated function) can then be applied to this appended data set. So, in the above example, inside the inner elwise there are no longer any dividing boundaries—all the original values are appended into one long vector—and the parallel vector version of + is then applied over all the elements.

5.2.3 Compiling Conditionals

This section describes the translation used for the `if` special form. Many of the other Common Lisp control forms, such as `cond`, `when`, and `do`, can be implemented with the `if` special form. The general `throw`, `catch` and `go` special forms, however, cannot and are not supported by the subset of Paralation Lisp accepted by the compiler. The translation described in this section is based on a technique called replicating [3].

The problem with the parallel (replicated) version of a routine with an `if` form is that, some of the segments might take one branch while others might take the other. The parallel version therefore might need to execute both branches. It, however, cannot simply evaluate both branches and select the appropriate result based on the conditional flag. This would render the program incorrect for the following two reasons:

- if there are any side effects (such as a `setq`) in the then-expression or else-expression, these side effects might be evaluated in segments in which they should not be;
- if there is a recursive call in one of the branches, the recursion would never terminate—the branch would be executed in the recursive call and again in the next recursive call with no termination condition.

It would also make the program inefficient for the following reason:

- If both branches are executed for all segments, computation is performed on the segments which were not supposed to take a branch. This can be particularly bad when conditionals are nested.

The first problem can be solved by guaranteeing that side effects are only executed in segments which should be taking the current branch. The second problem can be solved by only executing a branch if there is at least one segment that needs to execute that branch. The third problem can be solved by packing the active segments (segments that take the branch) into a shorter vector by deleting the inactive segments. The compiler inserts code for all these manipulations.

Figure 11 illustrates an example of the transformation executed by the compiler for the `if` special form. Each `or-reduce` is inserted to check if any segment needs to execute that branch. This code guarantees that at run-time a branch will only be executed if there are segments taking the branch. The `recursive-pack` function packs the segments of a variable so that inactive segments are dropped out. This function is applied to every variable that appears in a branch and can pack nested fields: it recursively packs the levels of a nested field and returns when it reaches the leaves. The `recursive-flag-merge` function merges two segmented values based on the conditional flags. As with the `recursive-pack`, it can be applied to nested fields. To merge in variables that are side effected in one of the branches (with a `setq`), a `recursive-unpack` routine is inserted at the exit of each branch. At run-time, this

```
(if flag
    (func1 a)
    (setq a (func2 b)))
```

⇓

```
(recursive-flag-merge flag
  (when (or-reduce flag)
    (func1 (recursive-pack a flag)))
  (when (or-reduce (p-not flag))
    (let ((result (func2 (recursive-pack b (not flag)))))
      (setq a (recursive-unpack result a (not flag)))
      result)))
```

Figure 11: Translating the parallel version of the `if` special form. If the flag is `NIL` in all of the segments, only `func2` is executed. If the flag is `T` in all of the segments, only `func1` is executed. If some flags are `T` and other `NIL` then the respective segments are packed before execution and merged after execution.

routine replaces the altered segments with the new values but leaves the unaltered segments unchanged.

Since our subset of Paralation Lisp only manipulates homogeneous fields, the results from the two branches of an `if` special-form must be of the same type so that the `recursive-flag-merge` will return a homogeneous field.

5.2.4 Operations

This section has covered everything except how the Paralation Lisp operations defined in Section 3.2 are implemented. The operations considered here are the elt, collapse, collect, and expand. A full description of the implementation of these functions is not within the scope of this paper, but the code is provided and the basic ideas are outlined (see Figure 12). Only one version of each function needs to be written because the compiler itself can be applied to this group of library functions to generate the nested parallel versions automatically.

The `collapse` routine generates a mapping which can then be used to move data using the `collect` routine. The mapping is represented with two vectors and a scalar. The *pointer* vector contains pointers from the original field into a new vector so that

```
(defun elt (sequence index)
  (extract sequence index))

(defun collapse (field)
  (make-map
    :pointers (rank field)
    :length (value-count field)
    :segdes (elements-counts field)))

(defun collect (field &key by)
  (make-pfield
    :values (make-pfield
              :values (permute field (map-pointers by))
              :segdes (map-segdes by))
    :segdes (map-length by)))

(defun expand (field)
  (let ((child-field (pfield-values field)))
    (make-pfield
      :values (pfield-values child-field)
      :segdes (+-reduce (pfield-segdes child-field)))))
```

Figure 12: The PARALATION LISP operations defined in SV-LISP.

equal values in the original field are adjacent. This pointer vector is generated by executing a `rank` on the original field. For example:

```
(rank #F(2 5 3 8 2 2 8))
  ⇒  #F(0 4 3 5 1 2 6)
```

If the field is an number field, the implementation of rank requires a sort. If the field is a boolean field, however, the rank can be implemented with two scans and a permute [3].

The *segdes* vector of the mapping contains the number of occurrences of each distinct value in the source field. It is generated with the `elements-counts` routine; for example:

```
(elements-counts #F(2 5 3 8 2 2 8))
  ⇒  #F(3 1 1 2)
```

If the field is a boolean field, this requires two calls to the `+-scan` primitive. The *length* slot of the map contains the number of distinct values and is generated with the `value-count` routine. For example:

```
(value-count #F(2 5 3 8 2 2 8))
  ⇒ 4
```

This is just the length of the `segdes` vector.

The `collect` routine extracts the pointers from the map structure and permutes its input to those pointers. Collect creates a field of fields from a field, using the *length* slot as the top level segment-descriptor and the *segdes* slot as the next level segment-descriptor.

The `expand` removes a level of nesting. It strips off two levels of segment descriptors, sums the lengths of the subfields, and creates a new segment descriptor with the length specified by this sum.

6 Compiler Output for Quicksort Example

This section expands upon the quicksort example of Section 2 by stepping through the actual output produced by our compiler. Earlier, we mentioned that the compiler must produce two versions of every function. One of the two functions works when called at top level (not within an elwise); the other works when called from within an elwise. Functions whose name begins with `s-` are top-level (serial) functions that operate on single elements of fields; functions whose name begins with `p-` are nested (parallel) functions. Both versions of the quicksort function are presented below. We begin with the top level function, `s-qsort`.

```
 0 (defun s-qsort (keys)
 1   (if (s-not-sorted-p keys)
 2       (let* ((pivot (s-elt keys (s-random (s-length keys))))
 3              (side
 4 x              (let ((key (pfield-values keys))
 5 x                    (pivot (s-distribute pivot
 6 x                                         (pfield-segdes keys)))
 7 x                    (current-segdes (pfield-segdes keys)))
 8 x                (make-pfield
 9 x                  :segdes current-segdes
10 x                  :values (p-< key pivot))))
11              (sub-data
12                (s-collect keys :by (s-collapse side)))
13              (sorted-sub-data
14 x              (let ((sub-data (pfield-values sub-data))
15 x                    (current-segdes (pfield-segdes sub-data)))
16 x                (make-pfield
17 x                  :segdes current-segdes
18 x                  :values (p-qsort sub-data)))))
19         (s-expand sorted-sub-data))
20       keys))
```

The parts of the source code that are not within an elwise require no translations. However, to clarify the distinction between the serial and parallel version, in this section we append a prefix of `s-` to each of the serial function names.[6] For example, the `>` in the original becomes `s->` in the output.

The parts of the source code that are within `elwise` require the three transformation steps discussed in Section 5.2.2. In the above example, the two elwise forms that appear in the source code are translated into the code that appears in the lines marked with an initial `x` (lines 4–10 and 14–18).

The first translation step translates the body of an elwise into its *parallel form* as described in Section 5.2.1. This includes replacing all the functions with their parallel versions (with a prefix of `p-`) as seen in lines 10 and 18 (note that recursive call of `qsort` is to the parallel version). Producing the parallel form also includes translating conditionals into their parallel form as described in Section 5.2.3, but in

[6]As mentioned in Section 5.2.1, special forms such as `if` and `let` have no parallel form and therefore receive no `s-` prefix.

the quicksort neither of the elwise statements include a conditional. The parallel form of a conditional will, however, be required below in the parallel version of `qsort`.

The second transformation step inserts code to distribute the value of variables that are free relative to the elwise binding as seen in lines 5–6. The third transformation step inserts code for *stepping-up* and *stepping-down*. The stepping-down translation can be seen in lines 4 and 14. The stepping-up translation can be seen in lines 8–10 and lines 16–18.

We now consider the nested (parallel) version of `qsort`, `p-qsort`. To generate this version from the source code, the entire function body is translated into its *parallel form* as in the first transformation step above. This transformation is necessary because the parallel version is always called from within an elwise. Notice that all of the `s-`'s in `s-qsort` have been turned into `p-`'s (this includes the `s-distribute` generated by translating the `elwise`).

```
 0 (defun p-qsort (keys)
 1   (let ((flag (p-not-sorted-p keys)))
 2 x   (recursive-flag-merge flag
 3 x     (when (or-reduce flag)
 4 x       (let ((keys (recursive-pack keys flag)))
 5           (let* ((pivot (p-elt keys (p-random (p-length keys))))
 6                  (side
 7                    (let ((key (pfield-values keys))
 8                          (pivot (p-distribute pivot
 9                                               (pfield-segdes keys)))
10                          (current-segdes (pfield-segdes keys)))
11                      (make-pfield
12                        :segdes current-segdes
13                        :values (p-< key pivot))))
14                  (sub-data
15                    (p-collect keys :by (p-match side)))
16                  (sorted-sub-data
17                    (let ((sub-data (pfield-values sub-data))
18                          (current-segdes (pfield-segdes sub-data)))
19                      (make-pfield
20                        :segdes current-segdes
21                        :values (p-qsort sub-data)))))
22             (p-expand sorted-sub-data))))
23 x     (when (or-reduce (p-not flag))
24 x       (recursive-pack keys (p-not flag))))))
```

The conditional in line 2 of the original has been translated into its parallel form (see the lines marked with an `x`). The translation, described earlier in Section 5.2.3, includes several parts. One is the two calls to `or-reduce` on lines 3 and 23, which guard the execution of the translated `else` and `then` clauses of the conditional. If there are no active segments that need to execute a clause, the clause is not executed. Another part is the inclusion of the `recursive-pack`'s (lines 4 and 24), which packs the active segments together, eliminating gaps caused by inactive segments (this is an optimization to reduce the length of vectors). The final part is the inclusion of the `recursive-flag-merge` (line 2), which splices together the packed results from the two conditional clauses.

7 Extensions and Optimizations

This section discusses some optimizations that could be made to the compiler, and discusses how it could be extended to include a broader subset of PARALATION LISP, and how it might be extended to another collection-oriented language, SETL,

One source of optimization is in the implementation of conditionals. In the implementation discussed in Section 5.2.3, a guard test is executed before running each clause to ascertain that at least one segment is active and needs to execute the branch. In some cases a clause is small enough that it is cheaper to always execute it than to check if it can be avoided. For example, in the code

```
(elwise ((flag flags))
  (if flag 0 1))
```

it costs little to generate the 0 and 1 compared to the cost of the two `or-reduce`s needed to test if all the `flag`s are `t` or `nil`. This optimization cannot be applied if the clause includes a recursive call since this might lead to an infinite loop—if the test is removed, there is no longer a termination condition. The decision of whether or not to apply this optimization might depend on several factors, some of which might not be available until run time.

A similar decision concerns the use of `recursive-pack`. Its use was described earlier as an optimization that reduces the length of vectors by removing idle segments. In some cases, such as the elwise above, it might be more efficient to operate on sparse segments and avoid the cost of the `recursive-pack`s (one per conditional clause) and the joining `recursive-flag-merge`. In other cases, it might be most efficient to pack the variables of one clause and not of the other and to use a special `recursive-flag-merge` to merge the packed and unpacked data. For example, in the quicksort discussed in Section 6, the second clause in `p-qsort` need not be packed (line 24) since it is immediately unpacked (line 2).

```
proc quicksort(S);
if S =  then return [];
else
  pivot-value := arb S;
  side := [pivot-value < key :  key in keys];
  sub-data := [[y in S | not(side)] [y in S | side]];
  sorted-sub-data := [qsort(data) :  data in sub-data];
  return sorted-suba-data(1) + sorted-sub-data(2);
end if;
end proc;
```

Figure 13: Quicksort in the language SETL.

One extension to the subset of PARALATION LISP accepted by the compiler would be to include more COMMON LISP data types and operations. These might include lists and list operations. Another extension would be to include heterogeneous fields. Heterogeneous fields might be added by using the representation discussed in [3]. This representation uses a separate vector for each type in a field along with a *tag* vector which specifies where the elements of each type belong in the field.

Figure 13 shows the quicksort routine in the language SETL [13]. All the techniques we have discussed for PARALATION LISP are also relevant to SETL, and the two quicksort routines could compile into the same SV-LISP code.

8 Summary

This paper has described a compiler for a subset of PARALATION LISP. The most interesting aspect of this compiler is the group of techniques used for flattening nested parallelism for execution on flat parallel hardware. The main ideas involve the use of segments to represent boundaries between nested subcollections. This allows the same code that operates on a flat collection to be applied to a nested collection, by replacing each primitive that occurs in the original code into its segmented parallel equivalent. Stepping-up and stepping-down is used to control the level at which parallelism is applied. Finally, the compilation of conditionals involves some non-obvious code manipulations and optimizations.

Nested parallel languages express parallelism in a natural way that at first does not appear to be supportable by massively parallel machines. This paper helps form a bridge from the convenience and power of nested collections and operations to the speed and simplicity of machines like the CM-2. We hope that this bridge will encourage the development and implementation of more easy to use high-level parallel languages like PARALATION LISP. Only by implementing languages that allow easy access to the power of parallel machines can we hope to fully take advantage of the natural parallelism available in most applications and algorithms.

References

[1] *APL2 Programming: Language Reference.* IBM, first edition, August 1984. Order Number SH20-9227-0.

[2] Guy Blelloch. Scans as primitive parallel operations. In *International Conference on Parallel Processing*, pages 355–362, IEEE Computer Society, 1987.

[3] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models.* PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, October 1988.

[4] Timothy A. Budd. *An APL Compiler for a Vector Processor.* Technical Report 82-6, The University of Arizona, Department of Computer Science, Tuscon, Arizona 85721, July 1982.

[5] C. J. Date. *Database Systems, Volume I.* Addison-Wesley, Reading, Massachusetts, fourth edition, 1986.

[6] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8, 1978.

[7] C. A. R. Hoare. Quicksort. *Computer J.*, 5(1):10–15, 1962.

[8] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc, New York, 1962.

[9] Kenneth E. Iverson. A dictionary of APL. *APL Quote Quad*, 18(1):5–40, September 1987.

[10] Alan H. Karp. Programming for parallelism. *IEEE Computer*, 43–57, May 1987.

[11] Trenchard More. The nested rectangular array as a model of data. In *APL79 Conference Proceedings*, pages 55–73, ACM, 1979.

[12] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* The MIT Press, Cambridge, Massachusetts, 1988.

[13] J. T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

[14] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[15] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine LISP: fine-grained parallel symbolic processing. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 279–297, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, August 1986. Also available as Thinking Machines Technical Report 86.16.

[16] Thinking Machines Corporation. *The Essential *Lisp Manual.* Cambridge, Massachusetts, 1986.