Regions: An Abstraction for Expressing Array Computation

Bradford L. Chamberlain E Christopher Lewis Calvin Lin[†] Lawrence Snyder

University of Washington, Seattle, WA 98195-2350
[†]University of Texas, Austin, TX 78712
{*brad,echris,snyder*}*@cs.washington.edu*, *lin@cs.utexas.edu*

Abstract

Most array languages, such as Fortran 90, Matlab, and APL, provide support for referencing arrays by extending the traditional array subscripting construct found in scalar languages. We present an alternative approach that exploits the concept of *regions*—a representation of index sets that can be named, manipulated with high-level operators, and syntactically separated from array references. This paper develops the concept of region-based programming and describes its benefits in the context of an idealized array language called *RL*. We show that regions simplify programming, reduce the likelihood of errors, and enable code reuse. Furthermore, we describe how regions accentuate the locality of array expressions and how this locality is important when targeting parallel computers. We also show how the concepts of region-based programming have been used in ZPL, a fully-implemented practical parallel programming language in use by scientists and engineers. In addition, we contrast region-based programming with the array reference constructs of other array languages.

1 Introduction

Since the earliest programming languages, array references have had subscripts associated with them. This notation, which was inherited from linear algebra, is natural and convenient for scalar languages since they operate on single values at a time. In contrast, array languages support the atomic manipulation of multiple array elements, so they typically extend traditional subscripting to a more complex form. APL [7], the first array language, supports the use of integer vectors in each subscript position, computing the outer product of the indices in each dimension to determine the elements referenced. Fortran 90 [1] uses a simplified variation on this syntax to support common reference patterns using triple or "slice" notation to describe a regular subset of elements. Both languages allow the subscript to be elided when referring to all elements of an array.

Though array language subscripting is a natural extension of scalar subscripting, array languages exhibit an important property that constrains subscripts. Operands in an array expression must be *conformable*, meaning that the subarrays referenced must have the same shape and size.[1] Conformability ensures that there are corresponding elements in each operand of an array expression so that its evaluation is well-defined. Conformability results in a strong correspondence between the subscripting expressions of array operands. Specifically, they will often be identical and almost always be similar. This property, which we refer to as *index locality*, follows naturally from the fact that programmers tend to organize and reference data in logical, constrained, and meaningful ways. Index locality motivates an alternative means of array reference using *regions*.

```
[lo..hi,lo..hi] A = B + C
```
(a)

$A[S;S] \leftarrow B[S;S] + C[S; S \leftarrow {}^{-}1 + LO + \iota HI\text{-}LO\text{-}1]$
(b)

```
A(lo:hi, lo:hi)=B(lo:hi,lo:hi)+C(lo:hi,lo:hi)
```
(c)

Figure 1: Different representations of the same array language computation in (a) RL and ZPL, (b) APL, and (c) Fortran 90 and Matlab.

This paper describes a region-based approach for expressing array computations. A region is a source-level index set that prefixes a statement to specify default indices for its array references. Array operators that modify the default indices can be applied to array expressions, resulting in different access patterns. In this way, *region-based programming syntactically separates array indexing from array references.*

In this paper, we present region-based programming and its benefits using a simple, idealized array language called *RL*. RL provides a vehicle for discussing region-based programming in general terms. As an example of region use in RL, Figure 1(a) shows a 2-dimensional region (in brackets) that covers an array statement. The statement specifies that elements of an $n \times n$ sub-array of `B` and `C` (where $n = \texttt{hi} - \texttt{lo} + 1$) are summed and assigned to the corresponding elements of array `A`. Semantically equivalent statements are given for APL in Figure 1(b) and for Fortran 90 and Matlab in Figure 1(c). From this trivial example, regions may appear to be a minor

[1]Though this is a common definition of conformability, it is not universal. Virtually all variations, however, at least require that the number of elements be the same.

To appear in the ACM SIGAPL/SIGPLAN Conference on Array Programming Languages, June 1999, Scranton, Pennsylvania.

syntactic variation on the other forms of indexing. However, we explain how region-based programming provides a powerful abstraction that has advantages related to notation, code reuse, and performance analysis.

This paper also describes how the ZPL parallel programming language is based on the region concept. ZPL is a real-world programming language that is publicly available on the world-wide web[2] and is in use by scientists and engineers [14]. We describe decisions made in the design of ZPL that not only support efficient parallel implementations, but also provide a performance model that allows programmers to easily reason about parallel overheads—an uncommon feature in parallel programming languages.

This paper makes two primary contributions. The first is an evaluation of region-based programming as an approach to array language indexing. The second is a discussion of how an efficient parallel programming language can be designed around the region construct. We detail the properties of region-based programming and enumerate its benefits. We give formal definitions of region and array operators and describe how their use enables the source-level identification of index locality, thereby improving programmers' understanding of their codes' performance on parallel computers. Furthermore, we compare the region-based representation to the conventional subscripted form. Although the region-based ZPL language has been described before [9], this is the first discussion of regions as an abstract programming language concept.

Section 2 describes the advantages of region-based programming. In Section 3, we give a formal definition for regions and describe RL's support for region declarations and operators. In Section 4, we introduce RL's array operators and describe how they are used to express general array computations. In Section 5, we describe how regions highlight index locality and provide benefits for parallel computing. We also discuss the relationship between RL and ZPL, and we compare the expressive power of the region-based approach with subscripting. In the final sections we describe related work and give conclusions.

2 Benefits of Region-based Programming

This section provides an overview of the benefits of region-based programming.

Notational Benefits

Regions eliminate redundancy. Factoring the common portions of a (potentially compound) statement's array references into a single region eliminates the redundancy of subscript specification, as illustrated by Figure 1. Though subscripted languages typically allow a subscript to be elided when referencing all the elements of an array, interior and boundary elements of an array are often treated separately, necessitating the use of subscripts. As a result, a region-based representation is more concise, easier to read, and less error prone. As an informal measure of conciseness, consider the example given in Table 1. Stripped of all indexing constructs, the ZPL and Fortran 90 versions of the SPEC CFP92 swm256 benchmark are similar in size. However, comparing the complete programs, Fortran 90 is considerably larger, devoting more than half of its characters to indexing, as compared to ZPL's 27%.

Regions accentuate the commonalities and differences among array references. Because the common portions of references are factored into a region, all that is left at the array references is an indication of how they differ. This applies the common language design principle that similar things should look similar, and different things should look different [10]. For example, the following RL statement contains four references to array `A`, each shifted in one of the cardinal directions. It is clear exactly how array `A` is being referenced in each operand.

```
[1..m,1..n] Temp = A@(-1,0) + A@(1,0) + A@(0,-1) + A@
```

The subscripted equivalent of this code requires closer scrutiny to discover the same relationship in its operands, let alone to verify its legality.

```
Temp(1:m,1:n) = A(0:m-1,1:n) + A(2:m+1,1:n) + A(1:m,2
```

Regions can be named. By naming regions, programmers can give meaning to index sets. It is difficult to associate meaning with unnamed indices, just as it is difficult to associate meaning with a memory address without using a variable name. For example, the name `TopFace` is far more illustrative than `[0,1:n,1:n]`. Providing the ability to name index ranges (as in APL) or even entire slices does not yield the same benefit, because a programmer must potentially name a great number of similar things. For example, the five distinct slices in the code fragment above would require five different names.

Regions encode high-level information that can be manipulated by operators. While subscript-based languages allow arithmetic operators to be applied to individual dimensions of a subscript, RL provides operators that apply to the index set as a whole. Regions can be defined in terms of other regions, which is conceptually simpler than repeatedly constructing related but different index sets. For example, RL's `of` operator assists in the clear definition and interpretation of boundary regions: given a region `Cube = [1..n,1..n,1..n]` and a direction vector `top = (-1,0,0)`, the region `TopFace` above can be defined using the expression `top of Cube`. Using region operators, a change to one region is reflected in all regions that are defined in terms of it, thus localizing modifications to the code.

Code Reuse Benefits

By separating the specification of array indices from the specification of computation, regions result in code that is more general and more reusable than subscripted code. For example, regions make it trivial to write statements or procedures that can operate on arrays of arbitrary size, while subscripted languages require the programmer to pass around and explicitly manipulate array bound information in order to achieve the same generality. Furthermore, regions can be applied to blocks of statements, including entire procedures, so the indices for entire blocks of code can be easily changed. In particular, scalar procedures can be trivially *promoted* to operate on arrays of any size and shape by simply specifying a region at the call site and passing in actual parameters of the appropriate rank. Moreover, changing a region-based program to operate on higher dimensional arrays can be a simple matter of changing the region declarations. The array computations themselves may not need to change, or they may need to change in minor and obvious ways, depending on the characteristics of the algorithm. In contrast, an array language such as Fortran 90 would require modifications to every array reference.

[2] http://www.cs.washington.edu/research/zpl/

language	*total characters*	*non-indexing characters*	*indexing overhead*
Fortran 90	3154	1513	52%
ZPL	1957	1421	27%

Table 1: Character counts for the SPEC CFP92 swm256 benchmark written in ZPL and Fortran 90. *Total characters* indicates the total number of non-whitespace characters in the codes once they are stripped of variable declarations and I/O. *Non-indexing characters* indicates the number of characters remaining once subscripting (in Fortran 90) and region/direction specification (in ZPL) are removed. *Indexing overhead* indicates the percentage of characters that are devoted to array indexing.

Performance Analysis Benefits

Perhaps the biggest advantage of region-based programming is its potential for aiding in performance analysis. The use of special operators to highlight correlations between each array operand's reference pattern emphasizes index locality. This has great benefit in the parallel realm where data locality plays a crucial role in determining performance. By supporting such operators and by clearly defining its data allocation policy, a parallel region-based language such as ZPL can enable programmers to reason about the parallel execution of their codes using straightforward syntactic cues. As a result, programmers and compilers can locate optimization opportunities by looking at the array operators used within a program, thereby avoiding complex analysis of subscripting expressions. These benefits are discussed in further detail in Section 5.

3 Regions

In RL, a region is a rectangular index set of arbitrary rank and stride, useful for defining arrays and array computations. This section gives a formal definition of regions, explains how they are declared in RL, and describes RL's operators for manipulating them.

3.1 Formal Region Definition

Each dimension of a region is defined by a 4-tuple *sequence descriptor*, $r = (l,h,s,a)$, where l and h represent the low and high bounds of the sequence, s is the sequence's stride, and a encodes the alignment of the sequence. A sequence descriptor, r, is interpreted as defining a set of integers, $S(r)$, as follows:

$$S(r) \quad = \quad \{x \mid l \leq x \leq h \text{ and } x \equiv a \pmod{s}\} \tag{1}$$

For example, the descriptor $(1,6,2,0)$ describes the set of even integers between one and six, inclusive: $\{2,4,6\}$.

A d-dimensional region $\mathbf{r}$ is defined as a d-ary sequence of sequence descriptors $r_1 \ldots r_d$, where r_i represents the indices of the region's i^{th} dimension:

$$\mathbf{r} \quad = \quad \langle r_1, r_2, \ldots, r_d \rangle$$

The index set, $I(\mathbf{r})$, defined by the region is simply the cross-product of the integers specified by each of its dimensions:

$$I(\mathbf{r}) \quad = \quad S(r_1) \times S(r_2) \times \ldots \times S(r_d)$$

For example, the index set of the 2-dimensional region $\langle (1,6,2,0),(1,6,2,1) \rangle$ would be described as follows:

$$\begin{aligned} I(\langle (1,6,2,0),(1,6,2,1) \rangle) &= S(1,6,2,0) \times S(1,6,2,1) \\ &= \{2,4,6\} \times \{1,3,5\} \\ &= \{(2,1),(2,3),(2,5),(4,1), \\ &\quad\ (4,3),(4,5),(6,1),(6,3),(6,5)\} \end{aligned}$$

3.2 Basic Region Declarations

Since dense regions constitute the common case in array-based languages, RL adopts the following as its most basic region specification:

$$\texttt{R = [}l_1\texttt{..}h_1\texttt{, }l_2\texttt{..}h_2\texttt{, ..., }l_d\texttt{..}h_d\texttt{]}$$

This style of declaration is used to define regions with trivial stride and alignment. A *degenerate dimension*—one with just a single index—can be declared by simply specifying the index (*e.g.*, `[3,1..n]`). Note that although RL could simply allow programmers to express regions in a sequence descriptor format, the more abstract syntax is clearer, improving readability. In RL, the specification above defines the formal region:

$$\mathbf{r} \quad = \quad \langle (l_1,h_1,1,l_1),(l_2,h_2,1,l_2),\ldots,(l_d,h_d,1,l_d) \rangle$$

Since the stride is always set to 1, the complete integer range $l_i \ldots h_i$ will be used for dimension i. RL's region operators (described in the next section) are used to modify the stride and alignment values of a region. Although a stride of value 1 makes the alignment term in a basic region inconsequential, setting it to the range's low bound results in a consistent and meaningful interpretation when region operators are used to modify its stride and bounds.

3.3 Region Operators

A set of *prepositional operators*—*of*, *in*, *at*, and *by*—are defined for the sequence descriptors. Each of these operators combines an integer value, δ, and a sequence descriptor to produce a new sequence. The operators are defined to transform the sequences as follows:

$$\delta \texttt{ of } (l,h,s,a) \quad = \quad \begin{cases} (l+\delta, l-1, s, a) & \text{if } \delta < 0 \\ (l,h,s,a) & \text{if } \delta = 0 \\ (h+1, h+\delta, s, a) & \text{if } \delta > 0 \end{cases} \tag{2}$$

$$\delta \texttt{ in } (l,h,s,a) = \begin{cases} (l, l+(-\delta-1), s, a) & \text{if } \delta < 0 \\ (l,h,s,a) & \text{if } \delta = 0 \\ (h-(\delta-1), h, s, a) & \text{if } \delta > 0 \end{cases} \quad (3)$$

$$(l,h,s,a) \texttt{ at } \delta = (l+\delta, h+\delta, s, a+\delta) \quad (4)$$

$$(l,h,s,a) \texttt{ by } \delta = (l,h,|\delta|\cdot s, a) \quad (5)$$

In short, the `of` and `in` operators modify the sequence bounds relative to the existing bounds, leaving the stride and alignment unchanged (`of` describes a range adjacent to the original range, whereas `in` describes a range interior to the previous range). The `at` operator translates the sequence bounds and alignment of the sequence. The `by` operator is used to scale the stride of the sequence, leaving the bounds and alignment unchanged.

Although there are certainly other region operators that could be useful to a programmer, those listed here were selected as a basis set since they support common array reference paradigms and are closed over our region notation. For example, RL does not support the set-theoretic union and difference operators due to the increased overhead of storing and iterating over non-rectangular index sets.

RL applies the prepositional operators to regions by factoring the δ offsets for each dimension into a vector called a *direction*. The following code specifies example directions and a region in RL:

```
south = (1,0)
east  = (0,1)
se    = (1,1)
se2   = (2,2)

R = [1..m,1..n]
```

Using RL's prepositional region operators, new regions can be specified using region R and directions:

```
EasternBoundary = east of R
SouthernInterior= south in R
ShiftedSE       = R at se
OddElements     = R by se2
```

The prepositional operators are evaluated for regions by distributing each component of the direction to its corresponding sequence descriptor and applying the prepositional operator. For example, the `at` operator would be distributed as follows:

$$\begin{aligned} \mathbf{r} \texttt{ at } (\delta_1,\delta_2) &= \langle (l_1,h_1,s_1,a_1),(l_2,h_2,s_2,a_2)\rangle \text{ at } (\delta_1,\delta_2) \\ &= \langle (l_1,h_1,s_1,a_1) \text{ at } \delta_1, (l_2,h_2,s_2,a_2) \text{ at } \delta_2\rangle \\ &= \langle (l_1+\delta_1, h_1+\delta_1, s_1, a_1+\delta_1), \\ &\quad\ (l_2+\delta_2, h_2+\delta_2, s_2, a_2+\delta_2)\rangle \end{aligned}$$

Having defined the prepositional region operators, we can now evaluate the RL regions defined above (see Figure 2 for illustrated interpretations):

$$\begin{aligned} I(\texttt{east of R}) &= (0,1) \text{ of } \langle (1,m,1,1),(1,n,1,1)\rangle \\ &= \langle 0 \text{ of } (1,m,1,1), 1 \text{ of } (1,n,1,1)\rangle \\ &= \langle (1,m,1,1),(n+1,n+1,1,1)\rangle \end{aligned}$$

$$\begin{aligned} I(\texttt{south in R}) &= (1,0) \text{ in } \langle (1,m,1,1),(1,n,1,1)\rangle \\ &= \langle 1 \text{ in } (1,m,1,1), 0 \text{ in } (1,n,1,1)\rangle \\ &= \langle (m,m,1,1),(1,n,1,1)\rangle \end{aligned}$$

$$\begin{aligned} I(\texttt{R at se}) &= \langle (1,m,1,1),(1,n,1,1)\rangle \text{ at } (1,1) \\ &= \langle (1,m,1,1) \text{ at } 1, (1,n,1,1) \text{ at } 1\rangle \\ &= \langle (2,m+1,1,2),(2,n+1,1,2)\rangle \end{aligned}$$

$$\begin{aligned} I(\texttt{R by se2}) &= \langle (1,m,1,1),(1,n,1,1)\rangle \text{ by } (2,2) \\ &= \langle (1,m,1,1) \text{ by } 2, (1,n,1,1) \text{ by } 2\rangle \\ &= \langle (1,m,2,1),(1,n,2,1)\rangle \end{aligned}$$

3.4 Flood Dimensions

RL also supports the concept of *flood dimensions* to represent lower-dimensional arrays as if they were higher-dimensional. Flood dimensions are represented by the sequence descriptor $(-\infty,\infty,0,0)$. While this specialized descriptor doesn't make strict mathematical sense by equation (1) above, it is used to represent a dimension with a single set of defining values that are replicated across an infinite index range. Flood dimensions are expressed in RL region specifications using an asterisk. For example, the following two regions would be used to represent 1-dimensional vectors perpendicular to one another in a 2-dimensional space:

```
Row = [*   ,1..n]
Col = [1..n,*   ]
```

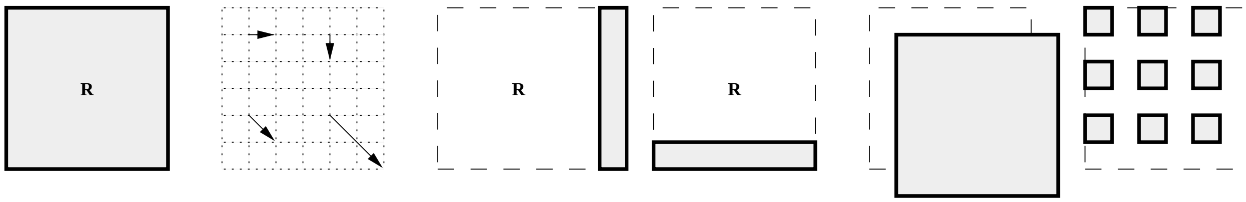
Flood dimensions are included in RL because they provide a means of expressing interactions between arrays of (conceptually) different rank without relying on explicit indexing. Their utility becomes even more pronounced in parallel region-based languages like ZPL, due to the performance implications of aligning arrays in a distributed context.

4 Computing with Regions

This section explains how regions are used to represent array computations in RL. We describe how regions specify the extent of array computations and then define RL's operators that modify these indices for individual array expressions.

4.1 Extent Specification with Regions

In RL, every array operand must be enclosed within a *region scope* of matching rank, known as its *covering region*. These region scopes prefix RL statements and specify the base set of indices named by their array references. Region scopes are dynamically scoped, allowing for the creation of region-independent functions and libraries. The following RL code fragment illustrates several properties of region scoping. Assume that `Interior=[1..m,1..n]`, `south`=(1,0), and arrays A, B, and U are 2-, 2-, and 1-dimensional, respectively.



(a) region R *(b) directions east, south, se, and se2* *(c) region east of R* *(d) region south in R* *(e) region R at se* *(f) region R by se2*

Figure 2: Illustrations of the region and direction declarations from Section 3.3. Note that the prepositional operators give intuitive meaning to the regions they define.

```
[Interior] begin
             A = 0;
             [south in "] A = 1;
             [1,] A = 2;
             [1..q] U = I1;
             A = A + B;
           end;
```

This fragment applies the 2-dimensional region scope `Interior` to a compound statement, providing a default set of indices for the 2-dimensional array references contained within. For example, the first assignment will zero out the $m \times n$ subarray of array `A` as specified by `Interior`. Similarly, the fifth will increment the same elements of `A` by their corresponding elements in `B`. The second and third assignments are locally covered by region scopes of rank 2, thereby eclipsing `Interior` as the cover for their 2-dimensional array references. The fourth assignment zeroes a 1-dimensional array, and therefore requires a 1-dimensional region cover.

The region scopes prefixing the second and third assignments demonstrate a region's ability to inherit information from an enclosing region. The second assignment's region scope is 2-dimensional (due to its reference to `south`) and uses the `"` symbol as a means of referring to the enclosing region of matching rank, namely `Interior`. The result is that elements of `A` in the southernmost row of `Interior` will be assigned the value 1. The next region scope omits a dimension specification, indicating that the dimension should be inherited from the covering region of matching rank, namely `1..n` from `Interior`. Thus, elements of `A` in row 1 of `Interior` are assigned the value 2. Providing `"` and blank dimensions in RL is more than a syntactic convenience, since they support the construction of semantically meaningful operations within region-independent functions (*e.g.*, operate on the south border or k^{th} row of the call site's covering region).

The fourth assignment uses `I1`, one of RL's predefined index arrays, to assign each element of `U` its unique index value. These arrays (`I1`, `I2`, *etc.*) give the programmer access to the indices of the covering region. In particular, the value of array $\mathtt{I}_i$ at a particular index is defined to be the value of the index in the i^{th} dimension. Array $\mathtt{I}_i$ may be used wherever an array of rank $\geq i$ is expected. As another example, the following statement assigns each element of array `A` its position within `Interior` in row-major order.

```
[Interior] A = (n * (I1-1)) + I2
```

4.2 Array Operators

In the examples of the previous section, every statement resulted in an elementwise operation over its operand arrays, due to the statement-level granularity of the region scopes. RL uses explicit array operators to express more complex array references whose indices vary from those of the covering region. This section defines the RL array operators, which are used to transform the covering region's indices when accessing their array operands. The result of any operator can be used as the operand to any other, and except where noted, array operators have an l-value.

The *shift* operator (infix `@`) translates the portion of its operand array that is referenced. Its left operand is the array to shift, and the right operand is a direction vector of the same rank that specifies the magnitude and direction of the translation in each dimension. For example, the following RL statement assigns the nearest neighbor average of the elements of array `B` as specified by the covering region into array `A`. Assume that the following directions are defined: `north` = (-1,0), `south` = (1,0), `east` = (0,1), and `west` = (0,-1).

```
[1..m,1..n] A = (B@north + B@south + B@east + B@west)
```

The *scale* operator (infix `$`) adjusts the stride in each dimension of a single array reference relative to the covering region. Its left operand is an array to scale, and the right operand is a direction of the same rank. The new stride in each dimension is the product of the corresponding direction element and the stride in the covering region. The low element referenced is the same as the low element in the covering region. For example, the following RL statement assigns the odd elements of array `B` between 1 and $2n$ to the consecutive elements of array `A` between 1 and n, inclusive.

```
[1..n] A = B\$(2)
```

The *promotion* operator (prefix $>$) transforms a d'-dimensional array into a d-dimensional array by replicating along d_f of its dimensions (where $d' = d - d_f$). A d-dimensional region—called an *operator region*—is encoded in the operator. The flood dimensions in this region (there must be d_f of them) specify which dimensions of the resulting array are to contain replicated data. For example, the following RL statement replicates elements 1 through m of 1-dimensional array `U` across the columns of 2-dimensional array `A`.

```
[1..m,1..n] A = $>$[,*] U
```

As this example shows, operator regions may contain blank dimensions to inherit from the covering region. Operator regions serve as the covering region for the operand array, which may itself be a complex array expression. Because the operand array expression for promotion has lesser rank than the operator region, the region formed by eliminating its flood dimensions covers the array operand expression. For example in the following statement, elements 1 through m of U and V are added together before performing the promotion.

```
[1..m,i] A = $>$[,*] (U+V)
```

The promotion operator can also be used to promote a subarray. This is expressed by specifying degenerate dimensions in the operator rather than flood dimensions. For example, the following RL statement copies the i^{th} column of 2-dimensional array B into columns 1 through n of 2-dimensional array A.

```
[1..m,1..n] A = $>$[,i] B
```

It is important to note that the implementation of promotion does not actually need to create a new array of increased rank (and increased storage requirements). Promotion simply provides a different way to reference data without changing memory requirements. Promotion expressions do not have l-values because they represent more elements than are actually represented in memory.

The *demotion* operator (prefix $<$) collapses d_d dimensions of an d'-dimensional array to produce an d-dimensional array ($d = d' - d_d$). A d'-dimensional operator region is encoded in the operator. The degenerate dimensions of the region (there must be d_d of them) specify which dimensions of the operand array are to be collapsed. For example, the following RL statement assigns column i of 2-dimensional array A into 1-dimensional array U.

```
[1..n] U = $<$[,i] A
```

As this example shows, the demotion operator's operator region may use blank dimensions. Though the covering region and the operator region have different rank, the operator region's blank dimensions will inherit from the corresponding dimension in the covering region (determined by ignoring degenerate dimensions in the operator region).

The *remap* operator (infix #) allows for arbitrary references by permitting the programmer to specify a map from indices of the covering region to indices of the operator's operand array. The operator's left operand is an array to remap, while the right is a vector of integer indices whose corresponding elements form an index into the operand array. The value of each element of the resulting array is the data appearing at this index in the operand array. The ranks of the argument array, integer arrays, and resulting array are all the same. For example, the following RL statement assigns each element (i, j) of A the value of element $(\mathtt{I}(i, j), \mathtt{J}(i, j))$ of B.

```
[1..m,1..n] A = B\#(I,J)
```

As a more specific example, the following statement assigns the transpose of array B to A. Note the use of predefined arrays I1 and I2.

```
[1..n,1..n] A = B\#(I2,I1)
```

Though all of RL's operators can be expressed using the remap operator, the specialized operators are not without value. They provide a more concise and readable representation of certain common operations compared to the general #-operator. Moreover, the specialized operators serve as a more accurate indicator of index locality and parallel cost, as discussed in Section 5.

4.3 Operator Summary

Figure 3 summarizes the semantics of each array operator. A function, $f_{\text{op}}(\ldots)$, is given for each operator that maps indices $\mathbf{j} = \langle j_1, \ldots, j_d \rangle$, of the rank d covering region to indices $\mathbf{j}' = \langle j'_1, \ldots, j'_{d'} \rangle$ of the operator's rank d' operand array.

5 Discussion

5.1 Index Locality in RL

At first glance, RL's array operators may appear to be gratuitous. For example, why should a language support the special-purpose @ and $ operators, when they can be expressed with the general-purpose # using simple functions of the index arrays $\mathtt{I}_i$? The answer is that RL's operators were selected to emphasize different types of index locality.

Index locality describes relationships between array indices. These relationships are important in the context of parallel computing because they translate directly to interprocessor communication. We have identified five types of index locality. Identical indices (*e.g.*, $(1,1)$ and $(1,1)$) exhibit *perfect locality*. Indices close to one other in the traditional Cartesian sense (*e.g.*, $(1,1)$ and $(2,1)$) exhibit *spatial locality*. Indices that are distant but which share common indices in one dimension (*e.g.*, $(1,1)$ and $(1,100)$) are considered to have *dimensional locality*. *Inter-rank locality* is exhibited by indices of different rank that share common coordinates (*e.g.*, $(1,2)$ and $(1,100,2)$). Finally, two indices whose coordinates are separated by a multiplicative factor are considered to have *locality of scale* (*e.g.*, $(2,2)$ and $(6,6)$). These definitions can be trivially extended to describe the locality of a pair of index sets rather than individual indices. Furthermore, note that indices may be related by a combination of locality types.

Since index sets are used both to define and access arrays, index locality directly correlates to locality of reference (dependent also on the data allocation scheme). This relationship between index locality and locality of reference is especially important in the realm of parallel computing, where locality affects the amount of communication (explicit or implicit) required between processors. RL thus emphasizes index locality through its region-based syntax and choice of array operators. Statements with complete locality (*i.e.*, all operations performed element-wise on identical indices) simply require the region defining the index set with no other special array operations. Other statements use the RL array operators to describe different types of index locality and to syntactically differentiate the different types of interprocessor communication:

- Statements with spatial locality use the shift operator to modify indexing by a constant offset.
- Dimensional locality is expressed using the dimension-preserving instance of promotion.
- Inter-rank locality is expressed using the promotion and demotion operators.
- Locality of scale is achieved using the scale operator.

code fragment	*signature*	*rank relationship*	$\mathbf{j}'$ *value* $(1 \le i \le d')$
$\ldots$	$f_{\text{no-op}}(\mathbf{j}) = \mathbf{j}'$	$d = d'$	$j'_i = j_i$
$\ldots @\mathbf{v} \ldots$	$f_{@}(\mathbf{j},\mathbf{v}) = \mathbf{j}'$	$d = d'$	$j'_i = j_i + v_i$
$[\mathbf{r_c}] \ldots \$\mathbf{v} \ldots$	$f_{\$}(\mathbf{j},\mathbf{v},\mathbf{r_c}) = \mathbf{j}'$	$d = d'$	$j'_i = \begin{cases} (j_i - g_{\text{low}}(\mathbf{r}_i))v_i + g_{\text{low}}(r_i) & \text{if } s_i > 0 \\ (j_i - g_{\text{high}}(\mathbf{r}_i))v_i + g_{\text{low}}(r_i) & \text{otherwise} \end{cases}$
$\ldots > [\mathbf{r_o}] \ldots$	$f_{>}(\mathbf{j},\mathbf{r_o}) = \mathbf{j}'$	$d = d' + d_{\text{nflood}}(\mathbf{r_o})$	$j'_i = \begin{cases} l_i & \text{if dimension } r_i \text{ is degenerate} \\ j_{i'} & \text{otherwise } (i' = d_{\text{flood}}(\mathbf{r},i)) \end{cases}$
$\ldots < [\mathbf{r_{o'}}] \ldots$	$f_{<}(\mathbf{j},\mathbf{r_{o'}}) = \mathbf{j}'$	$d = d' - d_{\text{ndegen}}(\mathbf{r_{o'}})$	$j'_i = \begin{cases} l_i & \text{if dimension } r_i \text{ is degenerate} \\ j_{i'} & \text{otherwise } (i = d_{\text{degen}}(\mathbf{r},i')) \end{cases}$
$\ldots \#(\mathbf{x}) \ldots$	$f_{\#}(\mathbf{j},\mathbf{x}) = \mathbf{j}'$	$d = d'$	$j'_i = x_i(j_1, j_2, \ldots, j_d)$

Notation

- $\mathbf{v} = \langle v_1, \ldots, v_{d'} \rangle$: rank d' direction
- $\mathbf{r_c} = \langle r_1, \ldots, r_d \rangle$: rank d covering region
- $\mathbf{r_o} = \langle r_1, \ldots, r_d \rangle$: rank d operator region
- $\mathbf{r_{o'}} = \langle r_1, \ldots, r_{d'} \rangle$: rank d' operator region
- $\mathbf{x} = \langle x_1, \ldots, x_d \rangle$: d-ary list of rank d integer arrays

Functions

- $d_{\text{flood}}(\mathbf{r}, i)$ = i^{th} non-flood dim. of $\mathbf{r}$
- $d_{\text{degen}}(\mathbf{r}, i)$ = i^{th} non-degenerate dim. of $\mathbf{r}$
- $d_{\text{nflood}}(\mathbf{r})$ = no. of flood dims in $\mathbf{r}$
- $d_{\text{ndegen}}(\mathbf{r})$ = no. of degenerate dims in $\mathbf{r}$
- $g_{\text{low}}(r = (l,h,s,a)) = l + (a-l) \bmod s$
- $g_{\text{high}}(r = (l,h,s,a)) = h - (h-a) \bmod s$

Figure 3: Array operator summary. The first column gives a code fragment indicating the operator's use. The second column summarizes the map function's argument signature for each operator. The third column describes the relationship between d and d'. The final column gives the value of an element of the resulting d'-ary $\mathbf{j}'$ index.

- The absence of index locality is indicated by using the catch-all remap operator, which can be used to arbitrarily scramble index sets and which may lead to unstructured communication.

The result is that the RL operators serve as clear visual annotations of a statement's index locality. This is a useful language property because given a particular data allocation scheme, both the programmer and the compiler have a clear means of reasoning about the implementation and expense of a particular piece of code. This simplifies analysis and optimization for both parties.

We can now see why RL enforces a stricter definition of conformability than slice-based languages. In the terms given above, `a(i,1..n)`, `b(1..n,j)`, and `c(1..n)` do not exhibit perfect locality and must therefore use array operations to describe their relationship. This is particularly important in a parallel implementation when an absense of locality implies interprocessor communication. In addition, many algorithms naturally tend to exhibit index locality, due to the ways in which data is typically stored and accessed. Though conformability merely requires that array operands need to be the same shape and size, there often exist additional logical correlations between the operand indices due to the ways in which programmers organize and reference data—the indices may be offset by a constant factor, scaled by different amounts, or projected from one dimension to another. Cases in which arrays are accessed in completely arbitrary patterns are relatively infrequent. To this end, the introduction of specific operators to emphasize the common case simplifies the expression of the operation (*e.g.*, A@(1,1) rather than `A#(I1+1,I2+1)`) and makes code easier to write and to understand (both for humans and compilers).

5.2 ZPL: A practical parallel region-based language

ZPL is a real-world instance of a region-based programming language that was designed for portable data parallel computation. Like APL, ZPL was designed to support array computations. Unlike APL, one of ZPL's chief design goals was to give programmers an intuitive model for reasoning about the concurrency and parallel costs of their programs. As a result, ZPL de-emphasizes general purpose operators that obscure costs. Instead, ZPL explicitly defines how arrays are allocated and provides operators that accurately reflect the cost of manipulating arrays with respect to the allocation. This is known as ZPL's WYSIWYG performance model [3].

Arbitrary array indexing is difficult to parallelize efficiently [5]. For example, the following Fortran 90 statement

```
A(i) = A(j);
```

is a simple assignment if the i^{th} and j^{th} elements of A reside on the same processor, but requires communication at a considerably higher cost if they do not reside on the same processor. Thus, in the general case this statement requires runtime checks to determine whether to perform communication. Worse, it is difficult for the programmer to reason about the performance of the statement since different compilers for the same language may compile the code differently. The ZPL solution is to use regions and region operators, which provide syntactic cues to indicate when the compiler will generate communication.

In order to emphasize data locality in the parallel context, ZPL maps all *interacting regions* (defined in [3]) to a conceptual processor grid of the same rank in a *grid-aligned* fashion, mapping region indices to processor indices in the corresponding dimension (*e.g.*, rows of a 2-dimensional region would be mapped to rows of a virtual 2-dimensional processor grid). Arrays are mapped to processors according to the region mappings. This has the result of preserving perfect, spatial and dimensional index locality across the virtual processor grid's topology. When there are no parallel operators, perfect locality exists, and the statement may be executed entirely in parallel. The use of the @ operator exploits spatial locality, potentially requiring relatively inexpensive nearest neighbor communication in the processor grid. The use of the promotion (>) operator exploits dimensional index locality, potentially requiring data to be broadcast along one or more dimensions of the processor grid. The remap (#) operator allows for completely general data movement and thus does not exploit locality.

For practical reasons, ZPL includes a number of features and operators not included in this discussion, such as masked computation, wrap and reflect operations for initializing and maintaining boundary conditions, scan operations, reduction operations, and multi- regions, arrays and directions for efficient parallel support of multi-grid applications.

5.3 Relationship to Subscripting

Two array references in a subscript-based language are typically considered conformable if the same number of array elements are referenced in corresponding, non-degenerate dimensions of the references. Region-based programming enforces a stricter meaning of conformability, because a single region selects the indices of all array references in a statement. Thus, it is the role of the array operators to map indices of the covering region to indices of the array operands, allowing for the expression of more general referencing. Despite the stricter definition of conformability, region-based programming is no less expressive. Additional operators are sometimes required to make references conformable, emphasizing the type of locality. For concreteness, Table 2 summarizes a number of ways by which array references may conform without being identical (column 1). For each, a Fortran 90 and APL example statements are given (columns 2 and 3) and their corresponding RL statement (column 4).

6 Related Work

The most prevalent alternative to region-based programming is array subscripting, as found in APL, Fortran 90, and Matlab [7, 1, 6]. As we have argued, array subscripting is a more cumbersome means of expressing simple array operations and is no more powerful than a region-based approach. Most importantly, these languages were not designed with parallelism in mind, thus it is very difficult for programmers to consistently achieve good performance and reason about their codes' parallel overheads.

Matlab is the current most popular array language for scientific computing, principally due to its interactive workbench approach to application prototyping and development and its extensive auxiliary library support. Besides being subscript-based, Matlab differs from ZPL in that it is designed to be a serial, interpreted language, so it is highly dynamic. Attempts to compile it are hindered by dynamic array allocation and data types [8], and attempts to parallelize it are limited by the fact that the language provides no support for managing locality or communication costs [11] in parallel implementation.

SAC (Single Assignment C) is a strict, purely functional subset of the C programming language, extended with richer support for arrays [12]. Like regions, the WITH-loop construct is used to limit the indices involved in a computation for a statement or group of statements. Despite the fact that SAC provides simple array operations such as `cat` and `rotate`, indexing is still required on array references in WITH-loops. Conversely, regions are dynamically scoped, so they may be separated from the statements to which they apply. As a result, a single region may apply to many nonadjacent statements, resulting in more concise code. SAC requires repeated use of WITH-loops and repeated index range specification. Finally, SAC is not a parallel language; it encourages the use of features, such as `reshape`, that are at odds with efficient parallelization.

Several parallel languages have supported mechanisms for storing and manipulating index sets. Parallaxis-III and C$*$ are two such examples, both designed to express a SIMD style of computation [2, 15]. Both languages support dense multidimensional index spaces that are used to declare parallel arrays. Parallaxis-III array statements are performed over the entire array, and therefore do not use index sets to describe computation. C$*$ does use its index sets (*shapes*) to designate parallel computation over entire arrays. However, it enforces a tight correspondence between the shapes of the computation and the arrays being operated on. Due to this restriction, its shapes are more of a type modifier than a general index set for expressing array computation. Both languages allow for individual elements to be masked on and off. Neither provides support for strided index sets.

FIDIL is another parallel array language designed for scientific computation [13] with support for more general index sets called *domains*. Domains need neither be rectangular nor dense, and FIDIL supports computation over them using set-theoretic union, intersection, and difference operations. The role of domains is limited to describing the structure of arrays (*maps*) and not for specifying computational references. Statements therefore operate either over the entirety of an array, or by indexing into the array as in scalar languages. Conformability in FIDIL is somewhat more dynamic than in other languages—operations are only performed on indices that are present in both operators.

KeLP [4] is a C++ runtime library that is a descendent of FIDIL. It supports shift, intersect, and grow operators on rectangular index sets called *regions*. KeLP uses regions to express iteration spaces using a “for all indices in the region” control construct. It departs from the region-based programming model described in this paper in that regions are used to enumerate indices which are then used to subscript arrays in the standard way. As a result, it does not support array operators to emphasize index locality. Furthermore, since regions are not an inherent part of C++, region manipulation is less elegant, with no implicit support for dynamically scoped regions and dimension inheritance.

7 Conclusions

This paper has developed the concept of regions in the context of parallel programming languages. Regions have advantageous properties that assist both programmers and compiler writers.

Regions provide notational advantages, removing the redundancy of applying identical or similar subscript specifications to each array operand. Rather, a set of indices is specified as a context in which conformant array computation is to be performed. Not only are the programs more succinct, the software engineering precept that similar things should look similar is respected. Programs are easier to read and write, will have fewer errors, and be easier to debug.

Regions are an effective abstraction. They can be named, allowing programmers to give them problem specific meaning. Further, the array operations such as `@` and `$` that transform regions to reference related elements define these elements more abstractly, in whole array terms. These higher-level concepts permit the programmer to think globally, saying what the index set should be, rather than how to realize that state operationally.

reference difference	*Fortran 90*	*APL*	*RL*
shift	`U(2:n+1) = W(1:n)+W(3:n+2)`	$U[1+\iota N] \leftarrow W[\iota N] + W[2+\iota N]$	`[1..n] U = W@(-1)+W@(1)`
stride	`U(1:n) = W(1:2*n:2)`	$U[\iota N] \leftarrow W[{}^{-}1 + 2 \times \iota N]$	`[1..n] U = W$(2)`
rank change (promotion)	`A(1:n,i) = U(1:n)`	$A[\iota N;I] \leftarrow U[\iota N]$	`[1..n,i] A = >[,*] U`
rank change (demotion)	`U(1:n) = A(1:n,i)`	$U[\iota N] \leftarrow A[\iota N;I]$	`[1..n] U = <[,i] A`
dim. alignment change	`A(1:n,i) = B(j,1:n)`	$A[\iota N;I] \leftarrow B[J;\iota N]$	`[1..n,i] A = B#(j,I1)`
vector subscripts (1-dim.)	`U(1:n) = W(V(1:n))`	$U[\iota N] \leftarrow W[V[\iota N]]$	`[1..n] U = W#(V)`
vector subscripts (2-dim.)	`A(1:m,1:n) = B(U(1:m),W(1:n))`	$A[\iota N] \leftarrow B[U[\iota N];W[\iota N]]$	`[1..m,1..n] A = B#(>[,*]U,>[*,]W)`

Table 2: Equivalent Fortran 90, APL, and RL statements that contain conformable, yet not identical references. All arrays in this table contain more than *n* elements in each dimension. In other words, these array references refer to subarrays. Arrays `A` and `B` are 2-dimensional and arrays `U`, `V`, and `W` are 1-dimensional.

From a program analysis and performance point of view, regions are ideal for parallel programming. Regions are the basis for identifying parallelism, for allocating memory and for planning out interprocessor communication. Using regions, the programmer and compiler can communicate at a high level regarding how these performance sensitive features will behave. These benefits have been demonstrated in ZPL.

We have neutralized the one potential liability of regions, that the higher level of expression with its somewhat stricter form of conformability is somehow overly constraining relative to slice notation. Regions were shown to be as expressive as array slice notation.

Our choice to restrict regions to a regular, rectangular index set was made in order to ensure an efficient parallel implementation and clear performance model. Masks (not described in this paper) are a mechanism for selecting arbitrary subsets of indices from these rectangular sets. In future work, we intend to generalize regions to describe less regular index sets for efficient sparse computation. The challenge will be to do so without sacrificing the efficiency and clear performance model of the current scheme.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jean T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, 1992.

[2] Thomas Bräunl. Parallaxis-III: A language for structured data-parallel programming. In *Proceedings of the IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, pages 43–52. IEEE, April 1995.

[3] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE, March 1998.

[4] S. J. Fink, S. R. Kohn, and S. B. Baden. Efficient runtime support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 1998. To appear.

[5] Raymond Greenlaw and Lawrence Snyder. Achieving speedups for APL on an SIMD distributed memory machine. *International Journal of Parallel Programming*, 19(2):111–127, April 1990.

[6] Duane Hanselman and Bruce Littlefield. *Mastering MATLAB*. Prentice-Hall, 1996.

[7] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[8] S. C. Johnson and C. Mohler. Compiling MATLAB. In *Proceedings of the USENIX Symposium on Very High Level Languages (VHLL)*, pages 119–27, Santa Fe, New Mexico, October 1994. USENIX Association.

[9] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, pages 96–114. Springer-Verlag, 1993.

[10] Bruce J. MacLennan. *Principles of Programming Languages*. Saunders College Publishing, 2nd edition, 1987.

[11] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: A MATLAB interactive restructuring compiler. In C. H. Huang, P. Sadayappan, U. Banerjee, S. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 8th Iternational Workshop on Languages and Compiler for Parallel Computing*, pages 269–88, Columbus, Ohio, August 1995. Springer-Verlag.

[12] Sven-Bodo Scholz. On programming scientific application in SAC — a functional language extended by a subsystem for high-level array operations. In Werner Kluge, editor, *Proceedings of the 8th International Workshop on the Implementation of Functional Languages (IFL '96)*, pages 85–104, Bad Godesberg, Germany, September 1996. Springer-Verlag.

[13] Luigi Semenzato and Paul Hilfinger. Arrays in FIDIL. In Robert Grossman, editor, *Symbolic Computation: Applications to Scientific Computing*, pages 155–169. SIAM, 1989.

[14] Lawrence Snyder. *Programming Guide to ZPL*. MIT Press (in press—available at publication date at ftp://ftp.cs.washington.edu/pub/orca/docs/zpl_guide.ps), 1999.

[15] *C* Programming Guide, Version 6.0.2*. Thinking Machines Corporation, Cambridge, Massachusetts, June 1991.