The Subspace Model: A Theory of Shapes for Parallel Systems*

Kathleen Knobe‡† William J. Dally‡

April 28, 1995

Artificial Intelligence Lab., Massachusetts Institute of Technology,
545 Technology Square, Cambridge, 02139, U.S.A.

Abstract

This paper presents a shape based abstraction for compiling to parallel systems. Data layout is often the subject of direct analysis while shape is addressed in ad hoc ways at best. However, a suboptimal shape can be more costly than a suboptimal location. Unnecessary serialization can result if the shape used is too small and unnecessary communication and computation can result if the shape is too large. For each dimension in the space of an object, the object attains that dimension, serially, in parallel or via a parallel prefix operation. These expansion categories are O(N), O(1) and O(log N) respectively, where N is the extent of the dimension. Using an expansion category slower than the natural one is unnecessarily slow. The subspace model addresses these problems.

There are three major aspects of this work. First, the subspace model is useful by itself. The subspace abstraction subsumes existing shape related optimizations (such as privatization and invariant code motion) and shape related strategies (such as owner computes and scalar replication). In this capacity it removes excess communication, excess computation and unnecessary serialization.

Second, the subspace model has an impact on the requirements for earlier phases. The processing to determine natural shapes and natural expansion categories uncovers parallelism. This implies that the source code need not be parallel. In this regard, the subspace model plays a role similar to the SPMD (Single Program Multiple Data) model.

Third, the subspace model has an impact on the effectiveness of subsequent phases. By converting the code to natural shapes and natural expansion categories, it improves the input to the data partitioner and the scheduler by providing them with additional flexibility.

1 Introduction

Although there are many sophisticated algorithms for data partitioning and scheduling, [1, 2, 3, 4, 10, 11, 13, 16] the inputs to these algorithms, that is the objects to be partitioned and the code fragments to be scheduled, are usually determined naively. Generally the objects

*An earlier version of this paper appeared in the Workshop on Automatic Data Layout and Performance Prediction, Rice University, Houston Texas, April 1995.

†Supported in part by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

‡Supported in part by the Air Force Systems-ESD under contract F19628-92-C-0045.

to be partitioned are the named arrays as declared by the user and the code to be scheduled is based on loop iterations as specified by the user.[1] The subspace model will improve the results of data layout and scheduling, not by improving the data layout and scheduling algorithms themselves, but by altering the input to the algorithms. This view has lead to an abstraction based on subspaces and a class of optimizations within that abstraction.

A shape is a subspace of the iteration space. We will use the notation $\{i, j\}$ to refer to the subspace spanned by the indices (vectors) `i` and `j`. Notice that subspace is distinct from dimensionality. For example, $\{i, j\}$ and $\{i, k\}$ are distinct subspaces but of the same dimensionality. Subspace is also distinct from location. Two objects can be in the same subspace, say $\{i, j\}$, but in distinct locations and two objects can be in distinct subspaces, say $\{i, j\}$ and $\{i, k\}$, but in the same location. The subspace abstraction is also distinct from the virtual processor abstraction used during alignment, for example, `a(i)` and `a(i+1)` are in distinct virtual processors but are in the same subspace, $\{i\}$.

The examples below show that the user can create objects whose dimensionality is too small or too large. Consider the two statements below with the same right hand sides (RHS) but distinct left hand sides (LHS). Assume that they are each within loops on `i`, `j` and `k`.

```
s1: a(i, j, k) = b(k) * c(i)
s2:       a(i) = b(k) * c(i)
```

If the reference `b(k)` refers to a 1-dimensional object with values at points along the `k` axis and the reference `c(i)` refers to a 1-dimensional object with values at points along the `i` axis, then their product, `b(k) * c(i)`, is a 2-dimensional object with values at points in the `ik` plane. In `s1` the LHS has more dimensions than the RHS expression. In `s2` the LHS has fewer. Inefficiencies may result if the subspace is not just right.

In statement `s1` since we assign the 2-dimensional object, `b(k) * c(i)`, to the 3-dimensional array `a(i,j,k)`, we may incur the cost of unnecessary communication to deliver those values to each point along the `j` axis. Worse yet, if we employ the *owner computes* rule (described below), `b` will be communicated across `i` and `j` axes and `c` will be communicated across `j` and `k` axes. Since only the 2-dimensional result needs to be expanded across `j` this communication is excessive. In addition, the number of multiplies is $imax * jmax * kmax$, a factor of $jmax$ more than required. In general, if a subspace has too many dimensions, we may incur the cost of unnecessary communication and unnecessary computation.

Although statements like `s1` are not very common, the owner computes rule applied to a similar statement `S3`

```
s3:  a(i, j, k) = x(i, j, k) + b(k) * c(i)
```

would cause the identical problem.

On the other hand, in statement `s2`, since we assign this same 2-dimensional object to the 1-dimensional array `a(i)`, we must serialize along the missing axis, `k` in this case. Also, all uses of `a(i)` must be completed at one point along `k` before `a(i)` can be defined at the next point along `k` (i.e., anti-dependences must be observed). In addition, the execution order of all assignments to `a(i)` for the same value of `i` must be maintained (i.e., output-dependences must be observed). In general, if the subspace of the LHS object has too few dimensions, we may incur the cost of unnecessary serialization.

The examples above demonstrate that the appearance of an index as a subscript in the source does not imply that the index is part of the subspace and the absence of an index as a subscript in the source does not imply the absence of the index from the subspace.

Although there are many compilation strategies and optimizations whose goal is to affect the shape of objects or operations, they are each aimed at distinct effects and operate in

[1] Although the grouping of iterations into chunks in order to optimize for granularity and aspect ratio, is often determined automatically, the individual iteration is usually that given by the user.

specific and restricted ways. Some examples of shape affecting compiler strategies are listed below. These are taken from the SPMD paradigm used by High Performance Fortran (HPF) [9].

- the owner computes rule - specifies the shape of all the *operations* in an expression to be the shape of the left hand side array section. It does this indirectly by specifying that the location of the computations is the location of the LHS.[2] This rule makes no distinction between shape and location.
- scalars are replicated - specifies that all *named objects* that are *scalars* will have a location on each processor. In other words, the shape of a scalar is the shape of the processor configuration. This rule applies only to named scalar objects. It does not apply to small arrays, to dimensions of arrays whose extent is small or to expressions whose value is a scalar.
- control flow expressions are replicated - specifies that the value of any *expression* that is *used as a predicate* will be distributed to all processors. In other words, the shape of such an expression is the shape of the expression itself expanded by the shape of the configuration.

In addition to the compiler strategies that affect shape mentioned above, we list below some optimizations designed to affect shape.

- privatization - is a transformation that *adds* a loop index to the shape of a *named object* in the specific case when all references to that object within an iteration of the loop refer to the value defined within that same iteration.
- invariant code motion - is a transformation that *removes* a loop index from the shape of an *operation* when the results of all iterations are identical. If the index is not the immediately surrounding one then this transformation can only be performed if loop interchange is legal.

As is evident by the examples above, the goals and domain of each of these strategies and optimizations are restricted in somewhat arbitrary ways. Some apply only to named objects, others only to expressions. Some distinguish between scalars and arrays. Privatization has the specific goal of increasing the dimensionality whereas invariant code motion has the specific goal of decreasing the dimensionality. On the other hand, sometimes no distinction is made where one might prove useful. For example, the owner computes rule does not distinguish between shape and location even though the cost of reshaping might be greater than the cost of relocating.

The goal of this work is to develop a general theory of shapes that subsumes or improves on the compilation strategies and optimizations above, that automatically determines the objects to be distributed and the code to be scheduled and that supports a variety of parallel targets.

The philosophy is analyze first, optimize, then synthesize, rather than successive transformations, each of which results in legal source code. The analysis phase first converts the code into single assignment form, then determines the natural subspace of the object associated with each assignment (Sections 2 and 3). The result of the combination of static single assignment and natural subspaces is a form that is dynamic single assignment at the element level, that is each element is assigned to at most once during execution. Finally the analysis phase determines the natural expansion category of each index in the subspace of each object (Section 5). Natural expansion category determination distinguishes between objects attaining an axis via a serial loop, a parallel-prefix loop and a parallel loop. The synthesis phase generates the appropriate set of loops, serial, parallel or scans as indicated

[2]There have been efforts to improve on the owner computes rule [4, 10] but they are within the domain of data partitioning.

by the expansion category (Section 6). This process may break up code that was within a single iteration into several distinct loops with possibly distinct expansion categories. The advantages of the resulting code as input to the data partitioner and scheduler are addressed (Section 7). The optional optimization phase, if included, occurs between the analysis and synthesis phases. It optimizes expressions given the natural subspaces and natural expansion categories of the leaves of the expression trees (Section 8).

2 Natural Subspaces

This section introduces some relevant concepts and terminology. We will refer to the statement below which, like most of the examples in this paper, is scalar code. We will be uncovering additional dimensionalities of objects and determining the natural type of parallelism of each dimension of each object defined. If some of the dimensions are already specified via declarations and some of the parallelism is specified, for example, via vector operation notation in Fortran90, these are incorporated directly into the algorithms; however, since the interesting aspects of our approach occur when a dimension or the parallelism is not explicit, most of the examples are simply scalar.

Assume `S4` is within loops on `i` and `j`.

```
s4:  c(i) = d(i+1, j) * e(i) + i
```

The term *named object* denotes any textual occurrence (definition or reference) of a scalar or array that is given a name in the source (e.g., `c` in `S4`). *Unnamed objects* are the results of computations (e.g., `d(i+1, j) * e(i)` in `S4`) . An *object* is either a named or an unnamed object. An *element* of an n-dimensional object is specified by `n` subscripts, one associated with each dimension. Each such element has a single value. An *index* is a loop index. The *natural subspace* of an object is a *subspace* of the iteration space, such that an index is part of the natural subspace of an object if the value of an element referenced may vary as the index varies. In general, the subspace of an object (named or unnamed) is the span of the subspaces of the objects used to compute it. If the subspace of `d(i+1, j)` and `e(i)` are $\{i, j\}$ and $\{i\}$ respectively, then the subspace of their product is $\{i, j\}$.

From the perspective of the source program, the value of a reference may change for two reasons, space or time. The element referenced may change as an index that appears in a subscript expression varies. For example, as `i` varies, the value of `e(i)` varies because a distinct element is referenced. This is a change in space. However, even if the index does not appear as a subscript, it is possible that the value changes. For example, as `j` varies the reference to `e(i)` may vary if there is an assignment to `e` within the `j` loop. This change does not correspond to a change in space, however, since the same element is referenced. The change is based on a change of time as the `j` loop is executed. In general, changes over space provide more flexibility than changes over time. In the above example, as `i` changes over space, and `e(i)` is defined over the range of `i`, all the values are available. However, for a given `i`, as `j` changes over time, and `e(i)` is defined for a given `i` over the range of `j`, only the most recent value is available. We will be converting the source with its changes over both space and time to an internal form called subspace internal form that represents all changes as changes over space. See [15] for a general discussion of data structures relating time, data space and processor space.

Consider the space $\mathbf{Z}^n$ with basis vector $\{e_1, e_2, ...e_n\}$ where each e_i is an index in the iteration space. This set has 2^n subsets. Each subset spans a subspace of the iteration space. For example, objects defined within loops on `i`, `j` and `k` may be in one of the eight distinct subspaces of the iteration space corresponding geometrically to the origin of the cube ($\{\}$), the axes of the cube ($\{i\}$, $\{j\}$, $\{k\}$), the faces of the cube ($\{i, j\}$, $\{i, k\}$, $\{j, k\}$) and the whole cube ($\{i, j, k\}$). Notice that this abstraction distinguishes among, for example, the 3 different 2-dimensional planes that have different orientations but does not distinguish among different positions on a given parallel plane, that is, it ignore details of expressions in subscripts. For example, regardless of whether the reference is `c(i)`, `c(i+1)`, `c(i*2)`, `c(sin(i))`, `c(v(i))`, or even `c(k)`, `i` is in the subspace of the reference if and only if the value of the reference may change as `i` varies.

The *dimensionality* of an object is the cardinality of the set of indices that specify its natural subspace. (If the natural subspace of the unnamed object `d(i+1, j) * e(i)` is $\{i, j\}$, its dimensionality is 2.) A loop index is always a 1-dimensional object (The natural subspace of `i`, used in `S4` both as a term and as a subscript, is $\{i\}$). Notice that we distinguish between constants (known to be in the empty subspace), loop indices (known to be in a 1-dimensional subspace) and all other references. We do not distinguish between array and scalar references (other than loop indices) because the algorithm will uncover the natural subspace of an object whether it starts out with 0, 1, or more dimensions.

A *definition* is an occurrence of a name that modifies values. A *reference* is an occurrence of a name that does not modify values. Consider an assignment to `x(i,k)` that creates a 2-dimensional object in $\{i, k\}$ space. A subsequent reference to `x` may refer to `x(i,k)`, also 2-dimensional and also in $\{i, k\}$ space. It may refer to `x(i,i)`, a 1-dimensional reference in $\{i\}$ space. It might refer to `x(i,v(k,j))`, a 3-dimensional reference in $\{i, j, k\}$ space.

Since each assignment may define an object of distinct subspace, several assignments to the same name in the source may define distinct objects in the subspace internal form specifically so the partitioner and scheduler can treat them independently.

A dimension in a reference is a *contributing dimension* if it contributes to the subspace. Consider the reference `x(i, 3)` where `i` is a loop index. The second dimension can not be a contributing dimension. If `i` belongs to the subspace of the reference, that is, if the value of `x(i, 3)` varies as `i` varies, then the first dimension is a contributing dimension. How we determine if `i` is part of the subspace is the subject of Section 3.

We specify a *reference mapping* to indicate the relationship between the source object and an internal subspace object. Dimensions of the subspace object are either source dimensions or expansion dimensions. For source dimensions, the reference mapping indicates the contributing dimensions. For example, the reference mapping $<(i, ., k)>$ indicates that the first dimension contributes `i`, the second is not a contributing dimension and the third contributes `k`.

For expansion dimensions the reference mapping indicates the expansion. Reexamine the assignment in the previous section. In assignment `s2`, since the source reference `a(i)` is determined to be in subspace $\{i, k\}$, the internal object generated is 2-dimensional with reference mapping $<(i)[k]>$[3] indicating that the first of the two dimensions in the internal object is a source dimension associated with index `i`, whereas the second is an expansion dimension associated with index `k`. The reference mapping notation may be nested when array subscripts are themselves arrays.

Recall that the subspace of the result of an arithmetic operation is the span of the subspaces of its arguments. An array reference is an operation whose arguments are the objects created by the subscript expressions of the contributing and expansion dimensions. The subspace of the result of an array reference is the span of the subspaces of these arguments. The determination of natural subspaces then involves determining the contributing and expansion dimensions for each array reference.

3 Algorithm for Determining Natural Subspaces

Now we are ready to present an initial subspace determination algorithm. For ease of understanding, we first consider only the language consisting of assignments and `do` loops.

[3]Multiple expansion indices are ordered canonically from outer to inner loops in this notation.

This restriction is relaxed in Section 9. This algorithm will determine a subspace for each textual occurrence of a named object.

Briefly, after an initialization phase (See Section 3.1), the algorithm propagates indices from the RHS to determine the subspace of the LHS. The indices of the subspace of the LHS determine the contributing dimensions of the LHS object, that is, the reference mapping (See Section 3.2). The contributing dimensions of the LHS object determine the subspace of the RHS references to that object (See Section 3.3). This propagation continues until there are no further changes. The algorithm is executed in worklist style. A reference is inserted onto the worklist when its current state changes. A reference is pulled off the worklist to propagate its change. No particular order of processing, such as inner loops first, is necessary.

The reference mapping described earlier showed the end product of this analysis. During the propagation, however, two temporary situations may hold. Recall that there are two requirements for a dimension of an array to contribute an index: That dimension must be a contributing or an expansion dimension. The index must be in the subspace of the subscript. These two may be determined in either order. The symbol `?` indicates that, for some dimension, we do not yet know if it contributes or not. The symbol `*` indicates that we know the dimension contributes but we do not yet know what indices it contributes.

Details of the algorithm follow.

3.1 Initialization

- Constants are in the null subspace since they do not vary with any index.
- Indices are in their own subspace.

  Consider the expression `a(i, j) + j`, in loops on `i` and `j`. The subscripts `i` and `j` and the term `j` are all known initially to be 1-dimensional in subspaces $\{i\}$, $\{j\}$ and $\{j\}$ respectively.
- For each reference that is live on entry (arguments and globals), each dimension is known initially to be a contributing dimension. There are no expansion dimensions.

  A reference `a(i, j)` that is live on entry is given the reference mapping $< (*,*) >$. If `i` and `j` are user scalars then this is all we know at initialization and the algorithm below will determine the subspace of `i` and `j` which will determine the subspace of `a(i, j)`. If `i`, for example, is a loop index then the reference mapping is converted from $< (*,*) >$ to $< (i,*) >$ based on the previous initialization rule.

  Note that if a source array is referenced several times, some of the references may be live on entry while others are not. Only the specific references that are live on entry are set by initialization. The subspaces of references not live on entry are determined by the algorithm.

3.2 Propagate from the RHS to the LHS

Subspace: If an index, `i`, is part of the subspace of one of the references on the RHS of an assignment, then it is part of the subspace of the LHS object.

In the expression `d(i+1, j) * e(i)`, when we determine that `j` is in the subspace of `d`, `j` will be added to the subspace of the LHS object and when we determine that `i` is in the subspace of `d` or `e`, `i` will be added to the subspace of the LHS object. The LHS object will be in natural subspace $\{i, j\}$ upon termination regardless of the number of dimensions declared for the LHS in the source.

Reference mapping: The mapping of an LHS to internal subspace form is determined by the indices in its subspace.

Consider an LHS, `a(i, j)`. When index `i` is added to the subspace, the first dimension becomes a contributing dimension with reference mapping $< (i, ?) >$. When index `k` is added to the subspace, an expansion dimension is added and the reference mapping becomes $< (i, ?)[k] >$. If the algorithm terminates without adding `j`, the final reference mapping for this reference is $< (i, .)[k] >$

3.3 Propagate from the LHS to each RHS reference

- For each contributing dimension in a source dimension of an LHS

  Reference mapping: If dimension, d, in an LHS reference is contributing, then dimension d contributes in any RHS reference reached via a true dependence.

  Consider the following code within loops on `i`, `j` and `k`:

```
s5:   a(i, j, k) = . . .
s6:        . . . = a(i, j, k)
s7:        . . . = a(i, k, j)
```

  If LHS of `s5` is in space $\{i, k\}$, the first and third dimensions are contributing and the reference mapping is s $< (i, ., k >$. When transferring from the LHS to an RHS, the relevant information is the contributing and expansion dimensions, not the specific indices contributed. Therefore the LHS reference mapping, $< (i, ., k) >$, is converted to $< (*, ., *) >$ to indicate the contributing dimensions before being transferred to the RHS references. $< (*, ., *) >$ is then applied to the references in both `s6` and `s7` indicating that the first and third dimensions are contributing.

  Subspace: If a dimension is contributing, the subspace of the subscript in that dimension determines the indices contributed.

  $< (*, ., *) >$ applied to `s6` becomes $< (i, ., k) >$. $< (*, ., *) >$ applied to `s7` becomes $< (i, ., j) >$. The first reference is in subspace $\{i, k\}$ whereas the second is in subspace $\{i, j\}$ even though the definition for both was in $\{i, k\}$.

- For each expansion dimension of an LHS

  Reference mapping: A expansion dimension in an LHS is transferred to an RHS it reaches as an expansion dimension exactly when the RHS is within the loop associated with the expansion index.

  Consider the following example.

```
s8:  a = 0
     do i = 1, imax
s9:      a = a + b(i)
s10:       ... = a ...
     enddo
s11:  ... = a ...
```

  The LHS reference to `a` in `s9` has reference mapping $< ()[i] >$, with expansion index `i`. This reference reaches three RHS references, in `s9`, `s10` and `s11`. The reference mapping for both `s9` and `s10` is $< ()[i] >$.[4] However, since the reference in `s11` is not within the loop on `i`, `i` is not included. A new object is created by *reduction* along that axis. Reduction typically involves taking the last value defined along an axis to create a new lower dimensional object.

  Subspace: An expansion dimension contributes the expansion index to the subspace.

[4] Since the reference in `s9` is reached via a loop carried dependence, the actual expansion will refer to the previous element along that axis, that is it will become `a(i) = a(i-1) + b(i)`.

The reference mapping, $< ()[i] >$, for the references in `s9` and `s10` indicate that these references are in subspace $\{i\}$ whereas the reference mapping for the reference in `s11` is $< () >$ and therefore in subspace $\{\}$.

To assess the complexity of the algorithm, consider the processing of the true dependence on `x` in the following code within loops on `i`, `j` and `k`

```
s12: x = b + c * d
 ...
s13:   = ...   x
```

If, at some point during the propagation, $\{i\}$ is added to the subspace of one of the RHS references in `s12` then `i` is propagated to the subspace of `x` on the LHS. If the subspace of `x` did not already include `i` then `i` is propagated through the dependence to the reference in `s13`. At some later time, if $\{j\}$ is added to the subspace of the same or another RHS reference in `s12` then `j` is propagated to the subspace of `x` on the LHS. The same dependence to the reference in `s13` is then processed a second time. It may be processed a third time for index `k`. It can be processed at most once for each enclosing loop. Consider the processing of a dependence to include both the propagation required from an LHS to an RHS and from that RHS to its own LHS. In the worst case each true dependence is processed at most once for each loop index. This algorithm is $O(D * L)$ where D is the number of true dependences (Anti- and output-dependences are ignored.) and L is the maximum loop depth.

Natural subspace analysis determines a set of objects and their subspaces. The objects include the named objects as determined above and the unnamed objects whose subspace is the span of the subspaces of their operands.

4 Operational Subspace

Given the natural subspace of the leaves of the expression trees, we now take a closer look at the intermediates. On distributed memory systems, operations occur between two operands of the same subspace. Corresponding elements are combined to produce a result. Assuming the obvious natural subspaces, the assignment `a(i,j) = b(i) + c(i,j)` in loops on `i` and `j` requires that we expand `b` from its natural 1-dimensional subspace, $\{i\}$, to a 2-dimensional subspace $\{i, j\}$ before performing the operation. The *operational* subspace of an operand is the subspace of the operation in which it is involved. In order to participate in the operation, the operand must be in its operational subspace. Since the subspace of an operation is the span of the subspaces of its operands, natural subspace is always a subspace of the operational subspace. The expansion from natural subspace to operational subspace can be performed via replication in logarithmic time.

Notice that some expressions can be computed locally without cost in any space that includes their natural space. Although a constant is in the empty subspace, it is available in any subspace without communication cost. In the expression `a(i, j) + i`, the term `i` is available in any subspace containing `i`, in particular in $\{i, j\}$, without communication. Such objects are referred to as *implicitly distributed* and incur no communication cost in expanding from their natural to their operational subspace.

The explicit expansion to operational subspace applies to all machines that exploit spatial (not just hierarchical) memory locality including distributed memory systems with both separate address spaces and shared address space (NUMA machines). The expansion to operational subspace does not benefit uniform memory access machines in which all processors can access all memory locations with equal cost (UMA machines). (This restriction only applies to expansions to operational subspace. Expansions to natural subspace apply to all these machines.)

One advantage to explicitly representing the operational subspace on a a system with physically distributed memory is that the expansion to an operational subspace may be redundant even when the specific communication is not. An example of this optimization is seen in the following examples. Assume that the code is within loops `i`, `j`, `k` and `m`.

```
s14:
 x(m,i,j,k)*v(i,j,k) + y(m,i,j,k)*v(i,j,k) + z(m,i,j,k)*v(i,j,k)
```

Assume `x`, `y` and `z` are all in subspace $\{m, i, j, k\}$ and that `v` is in subspace $\{i, j, k\}$. The communication to expand `v` to its operational subspace for each of the three multiplications will be exposed by the data layout phase. Assuming the obvious layout, with `x`, `y` and `z` aligned, two of the three expansions can be eliminated as redundant by optimizations after specific communication operations are exposed. However, consider the minor modification below.

```
s15:
 x(m,i,j,k)*v(i+1,j,k) + y(m,i,j,k)*v(i,j+1,k) + z(m,i,j,k)*v(i,j,k+1)
```

With the same layout of these objects, where the object `x`, `y` and `z` align, the actual communication to expand `v` across each of these objects will differ. So the expansion is not redundant at the level of explicit communication based on a specific layout and there are no redundant operations in the source. However, if we look at the intermediate level of natural and operational subspaces, the expansion of `v` across the axis `m` is redundant for two of the three multiplications. At the subspace level, we expand `v` across `m` once and at the location level, we simply adjust the alignment of this expanded object by one for each of the subsequent operations. Notice that the same argument would apply if the indices of `v` were identical but either the subscripts or the layout of `z`, `y` and `x` were distinct. Although this example is within a single expression and might be handled by statement level communication optimization, this type of redundancy is more common between than within statements. (See Section 8.3.)

5 Natural Expansion Categories

Now that we know the shape of each object, we will determine how the object attains each dimension of its shape. When an array, `x`, is privatized [14, 17] along a loop index, `i`, two things happen. The array is given an additional dimension, in our terminology, an expansion dimension, and accesses to `x` along `i` are asserted to be parallelizable.

We extend this notion in two ways. First, we find not only parallel expansions but all expansions, determining the expansion category of each. Second, we determine this category, not only for expansion dimensions but for all dimensions of the internal object, both expansion dimensions and source dimensions. These expansion categories are described below. The examples show both source and expansion dimensions.

- *Local* - the value of each element along an axis is not based (directly or indirectly) on elements earlier along that axis. The anticipated communication costs are O(1) for local axes. `s` in the following code is locally expanded across `i`.

```
do i = 1, imax               do i = 1, imax
 ...  = s                     ...  = s(i-1)
 s = x * y(i)                 s(i) = x * y(i)
 ...  = s                     ...  = s(i)
 enddo                       enddo
```

Our definition of local is similar to that of privatization but differs in two important ways. First, as mentioned above, it applies to existing source dimensions as well as to expansion dimensions. In addition, it does not require that all references to an element be in the same loop iteration. In the first statement, the use of `s` defined on the previous iteration does not prevent the definition of `s` in the second statement from being local, though it would prevent the object from being privatized. The critical importance of privatization has been detailed. [7]

- *Linear* - the value of an element is potentially based on the value of elements earlier along that axis. The communication costs are O(N) where N is the extent of the axis. `s` in the following code is linearly expanded across `i`.

```
do i = 1, imax                do i = 1, imax
  s = userfunc(s)              s(i) = userfunc(s(i-1))
enddo                         enddo
```

  Here we must order the calls to `userfunc` as in the original code.

- *Scan* - the value of an element is based on the value of elements earlier along that axis but the operations can be converted to parallel-prefix form with communication cost of O(log N). `s` in the following code is scanned across `i`.

```
do i = 1, imax                do i = 1, imax
   s = s + b(i)                s(i) = s(i-1) + b(i)
enddo                         enddo
```

The examples above are simplified in that in each, the axis in question corresponds to the innermost loop. In fact, an expansion category is determined for each dimension of an object, whatever level loop it corresponds to. In the following code, s is expanded via scan along `j` but is local along `i`.

```
do i = 1, imax
   s = 0
   do j = 1, jmax
      s = s + b(i, j)
   enddo
enddo
```

For linear and scan expansions, there is a *cycle* of nodes involving a loop carried dependence from an LHS to an RHS and from that RHS to the LHS within the same statement. The natural expansion category determination finds these cycles and distinguishes among distinct cycles created by loop carried dependences on the same loop. A given cycle may involve more than one object as in

```
do i = 1, imax
  x(i) = y(i) + ...
  y(i+1) = ...+ x(i)
enddo
```

Natural expansion category determination finds these distinct cycles and finds hierarchical relationships among cycles.

Recall that the subspaces of intermediates are determined by the subspaces of their children. The expansion category for an axis of an intermediate is determined by the expansion category of that axis of their children and of the LHS. Consider `a(i) = a(i-1)`

`+ b(i)`.[5] `a` is expanded via a scan across `i`. The object corresponding to the result of the `+`, is part of the scan because both the LHS and one of the children are part of the scan. In the assignment `a(i) = a(i-1) + b(i) * c(i)`, the `+` is still part of the scan but the `*` is not since neither child is part the scan.

To see why we must consider the LHS, consider an assignment `h(i) = a(i) + g(i)` following the above definition of `a`. The `+` in this assignment is not part of the cycle involving `a`, even though `a` is one if its children, because the LHS, `h(i)`, is not part of that cycle. This `+` can be a parallel use of the object, `a`, even though `a` was created linearly.

6 Restructuring

When the analysis component is complete, the natural subspace for each object, the natural expansion category for each axis of each object, and the cyclic relationships among expansions have all been determined. At this stage, a collection of optimizations available within the subspace abstraction constitute an optional optimization component (See Section 8). But before we discuss these optimizations, we should first understand how code is generated based on the analysis. This is the work of the synthesis component and is described in this section.

The subspace internal form will be transformed into hierarchically organized fragments based on subspaces, expansion categories and specific cycles, as determined by the analysis component.

An innermost fragment may be one of three types

- a *natural expansion fragment* of type local, linear, or scan (see Section 5)
- a *reduction fragment* (see Section 3.3)
- an *operational expansion fragment* (see Section 4)

Innermost natural expansions are represented as individual fragments. For example, the three expansions above in Section 5 are represented as the three fragments below.

```
local (i = 1,imax)      linear (i = 1, imax)       scan (i = 1, imax)
 s(i) = x * y(i)         s(i) = userfunc(s(i-1))    s(i) = s(i-1) + b(i)
endfragment             endfragment                endfragment
```

Although these fragments include entire statements, in fact, the fragments are determined at the level of specific operations. Operations are in distinct innermost fragments if

- they are in distinct subspaces,
- they are in distinct expansion categories over a given index,
- they are in distinct cycles over a given index,

Since a given cycle may contain parts of several statements and a given statement may not all be within the same cycles, a fragment may contain parts of several statements.

Consider the following code

```
y(i) = (a(i, j) + b(i)) + (x(i-1) + c(i))
x(i) = y(i) * d(i, j)
```

where the reference mappings for leaf nodes are

```
y<(i)[j]> = (a<(i,j)> + b<(i)>) + (x<(i)[j]> + c<(i)>)
x<(i)[j]> = y<(i)[j]> * d<(i,j)>
```

[5] This might have appeared this way in the source or it might have been expanded to this form via a natural subspace determination on `a = a + b(i)`.

and the cycles for leaf nodes are shown below. $\mathcal{C}$ indicates an occurrence that is part of the cycle. $\mathcal{N}$ indicates an occurrence that is not part of the cycle.

$\mathcal{C} = (\mathcal{N} + \mathcal{N}) + (\mathcal{C} + \mathcal{N})$
$\mathcal{C} = \mathcal{C} * \mathcal{N}$

Breaking this code into fragments involves determining the subspace and the cycles for each intermediate node. All four operations are in $\{i, j\}$ space. Only the left most plus is not within the cycle. The following four fragments are generated. **F2** and **F4** are natural expansion fragments. **F1** and **F3** are operational expansion fragments. (There are no reduction fragments in this example.):

```
F1: replicate (j = 1,jmax)
     bj(j) = b
    endfragment

F2: local (i = 1,imax; j= 1,jmax)
     t1(i, j) = a(i, j) + bj(i, j)
    endfragment

F3: replicate (j = 1,jmax)
     cj(j) = c
    endfragment

F4: local (j = 1,jmax)
     linear (i = 1,imax)
      y(i)[j] = t1(i, j) + (x(i-1)[j] + cj(i, j))
      x(i)[j] = y(i)[j] * d(i, j)
    endfragment
```

Notice the partial ordering among the fragments above. F4 depends on F3 and F2. F2 depends on F1.

Multi-dimensional objects may be computed within a single fragment as in **F4** above, but they may also generate a hierarchy of fragments if they have complex bodies. For a complex fragment, F, the body of F may contain multiple fragments, each is F_i for some i. Each fragment, F_i, is either an innermost fragment or another complex fragment. The fragments F_i are related by a partial ordering which indicates ordering constraints imposed on their execution, just as the fragments **F1-F4** above are related by a partial ordering.

The reorganization performed by natural expansion category determination exposes various forms of parallelism.

- Fragments (complex and innermost) with expansion category local for some axis can run in parallel across that axis
- Fragments (complex and innermost) with the expansion category scan for some axis can be performed via parallel-prefix operations.
- Fragments consisting of expansions to operational space can be performed via a parallel fan-out.
- For complex fragments, the partial ordering of the fragments composing its body expose task parallelism. Two fragments with no ordering between them can run concurrently even if they are based on the same index. Finding each distinct cycle based on each axis allows independent cycles based on the same index to run concurrently. If each of the two cycles has N operations and the axis has extent E the first loop completes in $E * N$ operations. The second, executing concurrently one processor behind the first, needs only an additional N time for its last iteration (total $E * N + N$), whereas the original combined loop requires $E * 2N$ operations.

7 Output of the Subspace Phase

This section addresses how this internal subspace form is used as the input for the scheduler and the data partitioner. The data partitioner determines the layout of the data across the processors. If the target system is data parallel, the layout of the data determines which processors perform the operations. Otherwise, a separate scheduler determines the distribution of the operations across the processors.

7.1 Scheduler

If the target system includes a compile-time scheduler, it is the fragments, in the natural subspaces and natural expansion categories, that become the basic units of scheduling. The scheduler will determine the appropriate distribution strategy for each loop of the fragment, presumably taking the natural expansion categories of the loops into account.

There are several advantages to scheduling based on the results of subspace analysis. The conversion to natural subspaces gives the scheduler more flexibility by eliminating unnecessary serialization due to subspaces that were too small. User iterations may be broken into distinct fragments. This has two advantages. First each fragment can run in its natural expansion category as opposed to the most conservative category. In addition, the concurrency among distinct fragments can be exploited within the constrains imposed by the required partial ordering.

The scheduler also has the option of combining fragments to increase the grain size to match target machine characteristics.

7.2 Data Partitioner

The data partitioner is given the objects (named and unnamed) in their natural spaces as the objects to be partitioned. The partitioner will determine the appropriate layout for each object, presumably taking the expansion categories of the object into account.

There are several advantages to partitioning the objects that result from subspace analysis as opposed to user specified objects.

- The objects are in their natural subspaces.

  This provides the partitioner with more options. Compare a user declared 1-dimensional object with its counterpart in its 2-dimensional natural subspace. If the partitioner chooses to serialize the added dimension of the 2-dimensional object, the options for partitioning the two objects are identical. But the partitioner has the added option of partially or totally distributing the added dimension. Since the options available for partitioning an expanded object are a superset of those for partitioning its unexpanded counterpart, it now has more flexibility.

- Each assignment to a named object is partitioned.

  This not only provides more options, it also means that for any given object, there are fewer other objects that interact with it. This means fewer constraints. Consider source object `x` that interacts with `a`, `b`, `c`, `d`, `e`, `f`, and `g`. When three distinct assignments to `x` are isolated, `x1` interacts with `a`, `b`, `c` and `x2`. `x2` interacts with `d`, `e` and `x1`. `x3` interacts with `f` and `g`. The partitioner still has the option of identically partitioning`x1`, `x2`, and `x3`, but now, in addition, it has other options that might be more effective. This is similar to scalar renaming[5] and static single assignment[6] but applied to arrays.

- Objects and their operational expansions are both partitioned.

  Since the data partitioner will be given an object in its natural subspace and operational expansions across distinct axes as distinct objects to be partitioned, if redistribution is required, the partitioner has the explicit choice of redistributing the original object or an expansion.

- Scalars and conditionals are partitioned.

  Scalars and conditionals in the source are converted to their natural subspace and their natural expansion category and are then treated as first class objects by the partitioner. Scalars are not automatically replicated to every processor at every modification. Nor are conditionals automatically replicated to every processor. (Conditionals are addressed in Section 9.)

- Expansion categories are available to the partitioner.

  Partitioning involves trade-offs between increased parallelism and decreased communication. The expansion categories for each axis of each object is the information needed to assess this trade-off.

After data partitioning and scheduling, the operands of the operations within a fragment may not all be aligned. Communication may be required to align them but, of course, this communication will include no reshaping, only alignment of objects within the same subspace.

7.3 Serialization

Since the data partitioner takes the actual machine size into account, many of the dimensions will be totally serialized or blocked (partially serialized).

Consider an object with an axis that is expanded by subspace analysis and then serialized by the partitioner. Although the goal of the expansion was to enable parallelism, no parallelism was enabled. Since the cost of the expansion in storage is worthwhile only when the parallelism is achieved, we reverse the expansion for serialized axes.

If the axis is blocked (partially serialized), limiting the extent of the parallelism, we limit the expansion to correspond exactly to the blocking.

8 Subspace Optimizations

We have introduced the subspace abstraction and have shown the determination of natural subspaces and natural expansion categories for each named object in the program and have discussed the form of the output to subsequent phases. In addition to these analyses, there is a class of optimizations within the subspace abstraction. Several such optimizations are presented below.

For an expression tree, given the subspace and expansion categories of the leaves, we can directly determine the subspace and expansion categories of the intermediates, and we can determine the necessary expansions to operational space. For some expressions, the same result can be computed in several different ways. Expressions can be reordered, for example, via commutative and associative laws, via factoring and the distributive law and via DeMorgan's law. These reorderings can alter the subspaces and expansion categories of the intermediates and the required expansions to operational space. We present below three different optimizations. Associated with each optimization is a cost metric on which to base an optimal expression ordering.

8.1 Subspace Minimization

Some of the legal reordering will reduce the subspace of the intermediates. For example, consider the following expression with the obvious subspaces.

```
s16:  a(i) + b(i,j,k) + c(i)
```

If the expression is evaluated as `(a(i) + b(i,j,k)) + c(i)` both `+` operations will be in subspace $\{i, j, k\}$ but by reordering the expression as `(a(i) + c(i)) + b(i,j,k)`, the first `+` is now in subspace $\{i\}$. This reordering results in fewer operations for that `+` by a factor of $jmax * kmax$ and also less communication since it requires the expansion of a single value instead of two values across $\{j, k\}$ to the operational subspace of $\{i, j, k\}$. This optimization is similar in flavor to transformations performed in APL compilers to limit the computations required in the presence of subspace changes [8]. This optimization has the same functionality as loop invariant code motion combined with some loop transformations. But it is more direct for two reasons. First, since the operands are in their natural subspaces, no analysis is necessary to determine if the expression is invariant within a particular loop. Second, since we do not maintain the loops while we do a series of loop transformations, we need no analysis to determine if a loop transformation is legal. We generate the appropriate loops based on the subspace and expansion category determination (See Section 7). This optimization roughly corresponds to minimizing total work.

8.2 Cycle Minimization

An operation is within a cycle if both the LHS and one of the children of the operation are within that same cycle. As with subspace minimization, reordering of the expression can improve performance, in this case by reducing the number of operations within the cycle. Consider the statement

```
s17:    a(j) = a(j-1) + b(j) + c(j)
```

where the two references to `a` are part of a cycle but neither `b` nor `c` are part of that particular cycle. If this is computed as `a(j) = (a(j-1) + b(j)) + c(j)` then both `+` operations are within the cycle. However, if we reorder it as `a(j) = a(j-1) + (b(j) + c(j))` then the sum of `b` and `c` is computable outside the cycle. Notice that the subspaces of both the intermediates are identical in both cases. We have simply made the expansion category of one of the intermediates more efficient. This optimization roughly corresponds to minimizing critical path.

8.3 Minimizing Expansions to Operational Subspace

We have shown above (See Section 4) that redundant expansions to operational subspace can be eliminated. However, we can go one step further and order the expressions to minimize the total cost of expansions to operational subspace. Consider the following two statements.

```
s18: ... = a(i) + b(j) + c(k)

s19: ... = a(i) * y(i, j)
```

Looking at `S18` alone, we have no reason to choose any one of the three possible expression orderings over any other, however, `S19` requires the expansion of `a` across `j`. This means that `a` expanded across `j` is available for free in statement `S18`. Therefore, in `S18`, adding `a(i)` and `b(j)` in $\{i, j\}$ space will be less expensive than the other two possibilities. We not only eliminate expansions that turn out to be redundant, we determine an ordering for the expressions that minimizes the cost of expansions given that we will be performing each expansion only once.

Since expansions to operational subspace are not performed on uniform memory access systems, minimizing operational expansions is not relevant on those systems.

9 Relaxing Language Restrictions

In our discussion so far, we have only addressed the tiny language that includes `do` loops and assignment statements because the ideas are best illustrated within this framework. This section addresses the incorporation of additional language features, specifically I/O and predicates, into the model.

I/O statements are easily incorporated into the natural subspace computations. Input statements act as definitions of objects and output statements act as references. The subspace model uncovers potential for parallel I/O. If an input statement reaches a use that is parallel along some axis, then that input statement may be parallelized along that axis. If an output statement outputs an object that is defined in parallel along some axis, then that output statement may be parallelized along that axis.

Now we consider incorporating `if` statements with predicates.

A predicate is simply an expression and its subspace is determined as we determine the subspace for any expression. For example, in the code

```
if (a(i) .gt. b(j)) then
```

the predicate is in subspace $\{i, j\}$ if `a(i)` is in subspace $\{i\}$ and `b(j)` is in subspace $\{j\}$. It might seem that since a predicate has no explicit LHS, the propagation of subspaces would stop at the conditional expression itself. However, we treat predicates simply as additional operands of the statements they control. We will allow them to be combined with other operands producing either an arithmetic result or the reserved value, NIX. For example, in the code

```
if (a(i) .gt. b(j)) then
 x(i, j, k) = c(i, j, k) + d(i, j)
```

we have the option of incorporating the predicate, whose natural subspace is $\{i, j\}$, into the expression either within subspace $\{i, j\}$ or within subspace $\{i, j, k\}$. In the first case, `d(i, j)` and the expression `(a(i) .gt. b(j))` are combined within $\{i, j\}$. The result is a single object in $\{i, j\}$ where each element has either the corresponding value of `d` or the special value NIX. Only that single result is expanded across $\{k\}$. In the second case, both `d(i, j)` and the expression `(a(i) .gt. b(j))` are expanded across `k`. The addition and the assignment are performed locally within subspace $\{i, j, k\}$ exactly where the predicate is true. When incorporating the predicate into an expression, if any of the operands of the operation are NIX, the result is NIX. This view of predicates facilitates their use as masks in Fortran90 library operations and the use of masks in `where` statements.

Predicates participate in determining the subspace of the result of the operation in the usual way. Predicates also participate in expansion category determination. For example, `s20` below creates a cycle not unlike that created by `s21`.

```
s20:                         s21:
   if a(i)... then              b(i) = a(i) + ...
      ...
      a(i+1) = ...              a(i+1) = b(i) + ...
```

Since predicates can be combined with any operation, they can participate in reordering for subspace minimization and cycle minimization. When predicates are involved, some expression orders can generate speculative execution, where subexpressions are computed before it is known that the results will be used. This speculative aspect can be controlled by controlling where predicates are incorporated in the expression.

Predicates are also optimized by elimination of redundant expansions and, in fact, their expansions are more likely than operands of arithmetic operations to be found to be redundant since a predicate may control many statements.

Notice that with this treatment, predicates are expanded only where they are needed and no further.

We must also consider the impact of blocking and serialization by the partitioner on conditionals. As with other code, expansion dimensions are limited to one instance per processor. But with (partially or fully) serialized conditionals, we also convert from the masking style of conditional execution back to use of control flow within a given processor.

10 Conclusions

We have presented a model based on subspaces. The model subsumes specific shape related optimizations (such as invariant code motion and privatization) and shape related compilation strategies (such as owner computes and scalar replication). The model determines the natural subspace and natural expansion category of each named object and optimizes the subspace and expansion category of intermediates. These optimizations reduce communication, reduce the amount of computation and remove serialization restrictions. The result of these transformations becomes the input to the data partitioning and code scheduling phases, that is, the objects to be partitioned and the code to be scheduled. This improved input provides the data partitioning and code scheduling phases with additional flexibility.

11 Acknowledgments

We would like to thank Bill Weihl, Joan Lukas, Fred Chong, Don Yeung, John Kubiatowicz and Carl Offner for fruitful discussions and helpful comments on earlier versions of the paper.

References

[1] S. G. Abraham, and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. In *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318-328, July 1991.

[2] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993. IEEE.

[3] Jennifer Anderson and Monica Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[4] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shuang-Hua Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC, January 1993. Association for Computing Machinery.

[5] Ron Cytron and Jeanne Ferrante. What's in a name? -or- The value of renaming for parallelism detection and storage allocation. In *Proceedings of the International Conference on Parallel processing*, Penn State Univ. University Park, PA, August, 1987. IEEE.

[6] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and F. Kenneth Zadeck. An efficiently method of computing static single assignment form. In

ACM Sixteenth Annual Symposium on Principles of Programming Languages, Austin, Texas, January 1989. Association for Computing Machinery.

[7] R. Eigenmann, J. Hoefinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In *Proceedings of the 4th workshop on Programming Languages and Compilers for Parallel Computing.* Pitman/MIT Press, AUG 1991.

[8] L. Guibas and D. Wyatt. Compilation and delayed evaluation in APL. In *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages.* Association for Computing Machinery, January 1978.

[9] HPF language specification, version 1.0. Technical Report CRPC-TR 92225, Rice University, Houston, Texas, January 1993.

[10] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[11] Kathleen Knobe and Venkataraman Natarajan. Automatic data allocation to minimize data motion on SIMD machines. *Journal of Supercomputing*, 1993.

[12] Kathleen Knobe. The subspace model: a target independent theory of shapes for parallel systems. PhD dissertation. Massachusetts Institute of Technology. (In preparation.)

[13] Jingke Li and Marina Chen. Index domain alignment: Minimizing costs of cross-referencing between distributed arrays. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, Maryland, Oct 1990. University of Maryland.

[14] Dror Maydan, Saman Amarasinghe and Monica Lam. Array Data-Flow Analysis and its Use in Array Privatization. In *ACM Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, Jan 1993.

[15] Carl Offner. A data structure for managing parallel operations. In *Proceedings of the 27th Hawaii International Conference on System Sciences. Volume II: Software Technology*, January 1994.

[16] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[17] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. *ACM SIGPLAN Notices*, 28(1), January 1993. Proceedings of the Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors.