An Object-Oriented Approach to Nested Data Parallelism

Thomas J. Sheffler
Siddhartha Chatterjee

The Research Institute for Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 212, Columbia, MD 21044, (410) 730-2656

Work reported herein was supported by NASA via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035-1000.

An Object-Oriented Approach to Nested Data Parallelism

Thomas J. Sheffler * Siddhartha Chatterjee *

Abstract

This paper describes an implementation technique for integrating nested data parallelism into an object-oriented language. Data-parallel programming employs sets of data called "collections" and expresses parallelism as operations performed over the elements of a collection. When the elements of a collection are also collections, then there is the possibility for "nested data parallelism." Few current programming languages support nested data parallelism however.

In an object-oriented framework, a collection is a single object. Its type defines the parallel operations that may be applied to it. Our goal is to design and build an object-oriented data-parallel programming environment supporting nested data parallelism. Our initial approach is built upon three fundamental additions to C++. We add new parallel base types by implementing them as classes, and add a new parallel collection type called a "vector" that is implemented as a template. Only one new language feature is introduced: the `foreach` construct, which is the basis for exploiting elementwise parallelism over collections.

The strength of the method lies in the compilation strategy, which translates nested data-parallel C++ into ordinary C++. Extracting the potential parallelism in nested `foreach` constructs is called "flattening" nested parallelism. We show how to flatten `foreach` constructs using a simple program transformation. Our prototype system produces vector code which has been successfully run on workstations, a CM-2 and a CM-5.

1 Introduction

The data-parallel programming model has proven to be popular because of its power and simplicity. Data-parallel languages are founded on the concept of collections and allow programmers to express parallelism through operations over the elements of a collection. Quite often the elements of a collection are simple types such as numbers. However, it is natural to think of collections containing other collections, and to write parallel algorithms in terms of nested collections. This style of programming leads to "nested data parallelism."

We distinguish between two forms of data parallelism: *elementwise parallelism* and *aggregate parallelism*. Elementwise parallelism allows a function f on a single element to be applied in parallel over the multiple elements of a collection. An example is adding 2 to every component of a vector. Aggregate parallelism applies a function g to the entire collection, presumably using a parallel implementation. Finding the mean of the components of a vector is an example. Moving to the object-oriented domain, a homogeneous collection $C(T)$ whose elements have type T has a natural representation using two classes: a class t for the elements, and a class c for the collection itself. We expect classes c and t to be "orthogonal": the operations on class vector are hopefully independent of whether the vector contains integers or floating-point numbers. The functions f used in elementwise parallelism are naturally defined as methods on class t, while the functions g used in aggregate parallelism are naturally defined as methods on class c. (In type-theoretic terms, this implies that the functions g are polymorphic over possible element types.) Finally, we need a mechanism to express the parallel application of f to each element of a collection.

Our key contribution is an object-oriented representation of elements and collections that allows not only the expression of elementwise and aggregate parallelism, but also the expression of nested data parallelism for nested collections. Our initial approach is built upon three fundamental additions to C++.

*Research Institute for Advanced Computer Science, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000 (sheffler@riacs.edu, sc@riacs.edu) The work of these authors was supported by the NAS Systems Division via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA).

1. We added new parallel base types `Int`, `Float`, `Bool`, and `Char` by implementing them as classes. These correspond to the base classes of element types t.

2. We chose the *vector* as an initial parallel collection type c, and implemented it through the use of templates, reinforcing the orthogonality of c and t.

3. We added one new language construct, the `foreach` construct, for supporting elementwise parallelism in collections.

The strength of our method lies in the compilation strategy, which translates collection-oriented code into regular C++. Most of the programming system is C++ code that carefully orchestrates the interaction between the base and collection classes. The foreach construct is the only new language feature and it is necessary to provide the fundamental source of parallelism for data-parallel programming. Exposing the parallelism expressed in a `foreach` construct is called "flattening nested parallelism." The task of translating the `foreach` construct is the most complicated responsibility of the compiler, and we propose a localized code transformation that flattens the nested parallelism in the `foreach` construct. This new method differentiates our system from others supporting nested parallelism [6, 14].

No other project currently addresses the integration of nested data parallelism into an object-oriented language. Support for nested parallelism requires that it be integrated into the language and run-time system. Rather than designing a new language with data-parallel semantics, we chose to merge nested data parallelism into C++ with a minimum of new language features. The result is that we are able to demonstrate a prototype programming system that meets our goal and that runs on a number of machines, including workstations, a CM-5, and a CM-2.

1.1 Benefits

Because our programming language is nearly standard C++ and our compiler translates sources to C++ as an intermediate step, our programming system reaps the benefits of the popularity of C++. Our system allows the use of many programming tools already available for C++. The programming model is familiar: parallel classes and collections are simply new data types with parallel semantics. The learning curve for C++ users is small because there is only one new language feature to learn: the `foreach` construct.

The system is also extremely portable. Our initial version isolates the parallel implementation to the lowest level of the class hierarchy. By implementing these low-level primitives on a new machine, the entire system may be ported to a new parallel computer with little effort. (This technique allowed us to quickly port the system to a number of parallel computers in this initial experiment.) This paper describes the initial implementation scheme based on a portable vector library called CVL [3]. However, we believe that higher performance may be obtained in the future by more sophisticated compiler techniques.

1.2 The importance of nested parallelism

Most data-parallel languages, such as the array sublanguage of Fortran 90 [1], limit the type of the elements of a collection to base types such as integers. Nested parallelism is the ability to build a collection whose elements are themselves collections, and to apply parallel operations defined on the elements of a collection in an element-wise fashion. For example, it should be possible to create a "vector of vectors" or an "array of vectors." If it is possible to sum a single vector with a parallel reduction function, then it should be possible to apply that same reduction in parallel to a collection of vectors.

Nested parallelism occurs naturally in the decomposition of many scientific problems. For example, a natural implementation of matrix-vector multiplication has the following structure.

```
foreach row R in matrix M in parallel {
    rowsum = sum(R);
}
```

The function `sum` is a parallel algorithm to compute a reduction in parallel. In the absence of a language that supports nested parallelism, the algorithm cannot be expressed in such a natural manner.

Nested parallelism is important for modular programming and the construction of reusable code libraries. Without it, users must develop new algorithms for each context in which a function occurs. In the previous example, the function `sum` is used in a parallel context. Without support for nested parallelism, a programmer designing a library might have to write two versions of `sum`: one to be used in a parallel context, and one that is not. Since a library designer cannot foresee the context in which a function can be called, it is impossible to write a truly general parallel function such as `sum` if nested parallelism is not integral to the programming system.

1.3 Comparison to other work

Many other projects are building parallel programming systems that support nested parallelism, object-oriented programming, or collection-oriented programming. This section touches on some of the efforts that are most directly related to ours. None have focused on object-oriented data-parallel programming that supports nested parallelism.

NESL NESL [5] is a data-parallel language designed to fully support nested data parallelism. It provides one collection type: the sequence. NESL is a strongly typed, polymorphic language with a type inference system. This makes it possible to write generic functions that work on any type of sequence, for example.

Our project is a direct outgrowth of NESL, and we owe many of our ideas to it [9, 17]. However, we believe that our programming system offers many practical features over NESL. First, NESL is purely functional. Because of this, NESL suffers the usual problems when updating subsets of aggregate variables, and it makes it impossible for users to manage memory use. An imperative language circumvents these problems — or at least places their solution under control of the programmer. NESL's runtime system is incompatible with other languages. In contrast, we translate our code into C++, and many compilers support mixing C, C++ and Fortran modules. We also provide a means to share data between such modules. It is currently not possible to link the NESL runtime system with other compiled modules.

pC++ The pC++ [7] language defines a new programming model called the "distributed collection model." This model is not quite data-parallel and it does not support nested parallelism. Its collections provide "object parallelism": a member function of a collection describes a parallel algorithm over the elements of the collection. Elementwise parallelism in collections is specified through special member functions of the collection that have a `MemberOfElement` designation. In our terminology, the classes c and t are merged into one, but the methods f are identified with the `MemberOfElement` designation. pC++ currently runs on workstations, the CM-5, the Paragon, the Sequent Symmetry, the BBN TC2000, and the KSR-1.

pC++ provides classes for managing thread parallelism over the distributed nodes of a SPMD parallel program, and provides means for specifying the distribution of objects over the processors of a parallel computer. The implementation of object-parallel functions on collections is quite complicated. It exposes three levels of parallelism to the writer of such a function: collection parallelism, thread parallelism, and element parallelism. The intent is that libraries will provide collection types, and most users will not have to deal with the intricacies of collection definition. However, since collections may not be members of other collections, the data types available to a programmer are somewhat limited. We believe that pC++ may be a suitable back-end for the definition of the low-level distributed data structures that our run-time system requires.

C** C** [12] is an object-oriented language based on a new computation model called "large grain data-parallelism." It makes good use of the data abstraction features of C++, but currently limits its collection types to arrays. Elementwise parallelism is provided for with special `parallel` functions. We believe that using functions to delimit element-wise code regions leads to an explosion in the number of functions it takes to write an algorithm, thus impeding its clarity. Our `foreach` may be placed at any point in a function, and can itself be nested. C** syntax allows for nested parallelism, but the runtime system does not currently support it.

Concurrent Aggregates Concurrent Aggregates [10] is a programming model that attempts to manage the complexity of concurrent access to a distributed data structure from multiple competing tasks. The elements of its collections are autonomous agents that each have their own thread of execution. Concurrent Aggregates allows collections to be acted

on as a single entity, as does our model, but allows collections to have elements of heterogenous types and supports control parallelism. This model is more general than ours, but may be more complicated to use and compile.

CC++ Compositional C++ (CC++) [8] supports data-parallel programming, cooperative processing and task parallelism. Its focus is synchronization and communication between processes. While it is object-oriented, it would be difficult to build data-parallel collections with it. There is no support for nested parallelism.

Paralation Lisp Paralation Lisp [16] supports collections in what are called "paralations" (parallel relations). A paralation has a shape ("vector" or "array," for example) and a number of data elements called "fields." Paralation Lisp adds parallel collections to Common Lisp along with an `elwise` clause that is nearly identical in function to the `foreach` construct we propose.

Because Paralation Lisp is built on Common Lisp, it is dynamically typed, making it difficult to compile effectively. Although Paralation Lisp defines the semantics of nested parallelism, only a limited implementation of the language was ever completed.

Proteus Proteus [13] is an architecture-independent language for prototyping parallel and distributed programs. Its primary data structures are sets and sequences, and it allows both parallel composition of processes and data-parallel computations. The data-parallel subset of Proteus is similar to NESL and it supports nested data parallelism. Proteus is an expressive language, but is difficult to compile. Currently, Proteus runs on a parallel shared-memory machine, and to a limited extent, on a MasPar MP-1.

1.4 Organization of this paper

The rest of this paper is organized as follows. Section 2 describes an object-oriented data-parallel language based on C++ that supports nested data parallelism. Numerous examples are used to show the power of the language and the notational simplicity of algorithms. Section 3 discusses the basis of nested data parallelism and the data structures we develop to support it. It then goes on to explain the translation procedure that flattens nested data parallelism. Section 4 describes a small application and presents performance results collected from a number of machines. The purpose of this section is merely to prove the existence of the implementation. The performance figures reported reflect a very immature implementation. The last section summarizes the points made in this paper and presents some directions for future research.

2 Language features and overview

This section gives a brief overview of the classes and templates that contribute to the language and programming system. As stated earlier, our desire was to experiment with the constructs and their implementation at an early stage. For this reason, the current language is limited to just a few base types and one collection type. In this discussion, we assume a basic familiarity with the syntax of C and C++. Note that the code presented appears somewhat unremarkable in that so little has been added to C++ to implement nested parallelism.

Four new classes define the primitive base types `Int`, `Float`, `Bool` and `Char`, and a large set of standard operations on these types. (Note that types `Int` and `Float` are different from the normal `int` and `float` of C.) The template `v<T>` defines a vector class given a type `T`, where `T` can be a primitive base type, a type derived from a primitive base type, or another collection type. The functions defined for vector types include permutation, truncation, concatenation, etc. In addition, the two special vector types, `v<Int>` and `v<Float>`, have parallel prefix and reduction operations defined.

Because one collection type may be the base of another, the definition of nested parallel types is simple. A vector-of-vectors is defined by nesting template invocations. (C++ parsing peculiarities require a space between the closing brackets of nested templates.) Vectors may be defined (space allocated) with an initial `size`, simply declared, or initialized from strings. Standard C++ stream I/O is defined for vectors so that vectors may be read and written just like any other type.

```
Int         x = 3;                  // given an initial value
v<Float>    vx(size(5));            // defined, all values are 0
v<Int>      vy;                     // declared
v<Int>      vz = "[1 2 3 4 5]";     // initialized

                                    // vector-of-vector-of-Int
v<v<Int> > vv = "[[1 2] [3 4 5] [6 7 8]]";

vy << cin;                          // read from standard input
```

The `foreach` construct is the basis for parallelism. In its body, the variables of the `foreach` argument list represent an individual element of the collection, and the statements are executed on the corresponding elements of each of the vectors. Vectors of the `foreach` argument list with a defined length must be of the same length, or the runtime system signals an error.

```
v<Int>      ind = index(5);         // [0 1 2 3 4]
v<Int>      v1;
v<v<Int> > v2;

foreach (ind, v1, v2) {
    v1 = ind * 2;
    v2 = index(ind);
}
// v1 ==> [0 2 4 6 8]
// v2 ==> [ [] [0] [0 1] [0 1 2] [0 1 2 3] ]
```

In this example, the function `index` creates an index vector whose values begin at zero. The vectors `v1` and `v2` demonstrate two types of parallelism over the vector `ind`. `v1` is result of simple elementwise parallelism using arithmetic operations. `v2` is the result of a nested parallel call to the `index` function, which creates the nested vectors of various size.

2.1 Sparse data structures

Nested vectors are useful for defining sparse data structures. A sparse matrix is represented as a collection of rows, where each row is a collection of (column, value) pairs. For example, the sparse matrix

$$\begin{pmatrix} 1.0 & 2.0 & \\ 3.0 & 4.0 & 5.0 \\ & 6.0 & 7.0 \end{pmatrix}$$

may be stored in two vectors as the following.

```
v<v<Int> >   col = "[[0 1] [0 1 2] [1 2]]";
v<v<Float> > val = "[[1.0 2.0] [3.0 4.0 5.0] [6.0 7.0]]";
```

Each sub-vector (called a *segment*) of the two vectors stores the information pertaining to one row. A new class can be defined to encapsulate the two vectors that define a sparse matrix. Using this representation, it is easy to write an algorithm that multiplies a sparse matrix by a dense vector. The following algorithm demonstrates several levels of nested parallelism. Temporary variables are used to help explain its operation.

```
class Sparsemat {
    v<v<Int> >   col;
    v<v<Float> > val;
}
```

```
v<Float>  mvmult(Sparsemat m, v<Float> x)
{
    v<Float>  y;
    v<v<Int> >   col = m.col;            // extract members
    v<v<Float> > val = m.val;

    foreach (col, val, y) {               // for each row
        v<Float> product;                 // temporary vector for each row
        foreach (col, val, product) {     // for each element
            Float get = x[col];
            product = get * val;
        }
        y = sum(product);                 // nested parallelism
    }
    return y;
}
```

The algorithm works as follows. The outer `foreach` construct operates over the rows of the matrix, and the inner `foreach` over the elements of each row, which is defined by a (column, value) pair. At the innermost level, each element retrieves the value from the vector `x` indexed by its column and stores it in a temporary variable called `get`. Next, each element multiplies its value by the value retrieved from the vector and stores it in `product`. Each row computes the `sum` of its `product` values to produce the `y` value for that row. Upon exiting the outermost `foreach`, `y` becomes a vector with one result value for each row.

2.2 Classes and user-defined types

New parallel types may be defined as C++ classes whose members are the primitive base types, or other parallel types. Member functions may also be defined for new types. A point on the plane can be defined as a new parallel type that has an `x` and `y` value. The member function `dist` computes the distance of a point from the origin.

```
class Point {
  public:
    Float    x, y;

    Float dist()
    {
        return sqrt(x*x + y*y);
    }
};

Point  p1, p2;           // declare two points
...
cout << p1.dist();       // print the distance of p1 from the origin
```

New parallel types may serve as the base type for new vector types, and the member and friend functions of the base type may be called inside a `foreach` clause. The following function computes the average distance of a collection of points from the origin. (Note: `length` is a standard member function defined for all vectors.)

```
Float avgDist(v<Point> points)
{
    v<Float>    distance;
```

```
    foreach (points, distance) {
        distance = points.dist();
    }
    return sum(distance) / Float(points.length());
}
```

The `Point` class is a type derived from the primitive base types and may be used as the base type for a collection. Thus, a vector of points is a type that can be generated by the vector collection template. A more complicated example illustrates how to fit a line to a vector of points using a least-squares fit. This function takes as arguments a vector of points, and references to two variables in which to place its results.

```
void leastSquares(v<Point> points, Float &yintercept, Float &slope)
{
    Float     num = Float(points.length());   // of the vector
    Float     xavg, yavg, Stt;
    v<Float>  x, y, xx, xy;

    foreach (points, x, y) {
        x = points.x;                    // memberwise field extraction
        y = points.y;
    }

    xavg = sum(x) / num;
    yavg = sum(y) / num;

    foreach (x, y, xx, xy) {
        Float tmp = x - xavg;            // element-wise temporary
        xx = tmp * tmp;
        xy = tmp * y;
    }

    Stt = sum(xx);
    slope = sum(xy) / Stt;
    yintercept = yavg - xavg * slope;
}
```

The first `foreach` construct demonstrates the idiom for extracting members of a vector of structures into separate vectors. The next two lines compute averages of the x and y values, and the rest of the function is straightforward. The last two lines store the two results in the reference variables. Of course, since function `leastSquares` is defined on parallel data types, it can be called inside a `foreach` over a vector of type `v<v<Point> >`.

2.3 Template functions

Users may write template functions to implement generic operations on any type. For example, the following function reverses the elements of any vector. Its implementation uses the function `permute`, which is defined for all vector types. In fact, all of the predefined operations for vectors in the system (`permute`, `send`, `get`, `distribute`, `append`, etc.) are implemented using template functions.

```
template <class T>
v<T> reverse(v<T> vector)
{
    Int        len = vector.length();  // vector length
    v<Int>     ind = index(len);       // [0 1 2 ... n-1]
```

```
    foreach (ind) {
        ind = len - ind - 1;          // reverse index computation
    }
    return permute(vector, ind);
}
```

2.4 Summary

This section touched on some of the features of the language. We introduced the primitive base types and the vector collection template. Note how the various features of C++ are used to express data parallelism. Classes are used in a fundamental way to encapsulate both collections and elements of collections. Template classes are used to implement the collection types. Both true polymorphism and the more *ad hoc* operator overloading are used in building these classes. Stream I/O is cleanly integrated into the language. Finally, static type checking is used extensively to assist in compile-time optimizations. This final feature differentiates our effort from Paralation Lisp, which relies on dynamic type checking.

Two major features of C++ that we have not used so far are inheritance and exception handling. We feel that these features appear when writing large codes, independent of whether the code is sequential or parallel. Thus, we expect these constructs to show up in large user codes.

3 Flattening nested parallelism

Our approach to flattening the nested parallelism of `foreach` constructs is based on a simple program translation method. We add a small prolog and epilog to each `foreach` body and eliminate the `foreach` keyword entirely, leaving standard C++ code. The `foreach` body remains textually unchanged, and the final compilation interprets the statements of the body in the new context of the prolog and epilog. Our approach makes use of the strong typing of C++ to manage the nesting of parallel constructs. The key idea we develop is a translation scheme that enables elementwise parallelism to be implemented through an effective change in the type of a collection object. This section sketches only the basic idea of the translation.

3.1 Parallel contexts

Nested data-parallel programs have different contexts of execution. A region inside a `foreach` construct defines a new context that changes the *type* of its variables. A variable name that refers to a vector outside the `foreach` construct is changed to refer to one of its elements inside the construct. Consider the following very simple code fragment.

```
v<Int>    v1(size(10)), v2(size(10)), v3(size(10));
Int       s1, s2, s3;
...
s1 = s2 + s3;                    // of type 'Int'
foreach (v1, v2, v3) {
    v1 = v2 + v3;                // refers to type 'Int'
}
```

Outside of the `foreach`, the variables `v1`, `v2` and `v3` refer to vectors of type `v<Int>`; inside they refer to simple `Int` types. In fact, the `foreach` construct changes the effective type of the variables of its argument list from a collection type, to the type of one of its elements. The abstraction presented by the `foreach` clause is that inside its boundaries, statements are executed on each element of a collection in parallel.

The example shows two instances of the addition operator (+): one outside and one inside the `foreach` construct. In effect, both instances of this operator mean to add two variables of type `Int`. However, it is apparent that inside the `foreach` the operator signifies the addition of the elements of two collections (in lock-step synchrony because

this is a data-parallel model), and that outside it signifies the addition of two scalar variables. In fact, by introducing the attribute of *plurality*, we can unify the meaning of the two apparently different operations implied by the contexts in which `operator+` occurs.

3.2 Plurality

In our programming system, *all* types are plural — there is no simple scalar type. The plurality of a variable is an attribute that describes the number of its *components*. (In C* [15], for example, there is a differentiation between mono and poly types. Our programming system makes no such distinction.) What might be considered a "scalar" is actually a variable with a plurality attribute of one, but this is only a special case.

For this discussion, we will introduce a notation for writing the components of a variable of a given plurality. Each component is written separated by a vertical bar. The printed values of two variables are shown below. `i1` is an `Int` of plurality one, and `i2` is an `Int` of plurality five.

```
i1 ==>  5                                  // one component
i2 ==>  10 | 20 | 30 | 40 | 50             // five components
```

All variables in our system have the plurality attribute. The plurality of a variable, however, is not a user-accessible feature of an object: it is an attribute of use only to the implementation. For instance, a variable of class `Int` may have plurality of 10. The class `Int` defines addition for values of the same plurality.

With the attribute of plurality, the meanings of the two instances of `operator+` in the example above may be unified. We say that a `foreach` construct deconstructs a collection yielding a variable whose *plurality* is the *size* of the collection. In the example, the type of the variables `s1`, `s2` and `s3` is `Int`, and their plurality is one. Inside the `foreach` construct, the type of the variables `v1`, `v2` and `v3` is also `Int`, but their plurality is 10. Because the type `Int` is defined over values of arbitrary plurality, the same operator applies to both contexts. This is why we needed to define the class `Int`; C++'s `int` type does not provide the arbitrary plurality that we need.

The class `Int` is defined such that its constructor only creates instances with plurality one. However, by deconstructing collections, instances of `Int` may be obtained with greater plurality. Similarly, the constructor for `v<Int>` creates a vector collection whose plurality is one, but a deconstructed `v<v<Int> >` could result in a vector collection of greater plurality.

3.3 Plurality and collections

The obvious question now arises:

> How is a variable of a given plurality different than a collection of the same size?

In short, the plurality of a variable merely indicates the number of its components, but this set of components is not necessarily arranged in any particular order. A collection imposes a *shape* (like "vector" or "array") on the components of a variable to form a collection.

Operations defined for plural variables can include only their construction and destruction and those operations that can be applied to their components in parallel. Examples are parallel addition on the components of variables of type `Int`, and parallel permutation on the components of variables of type `v<Int>`. (Recall that *all* types have multiple components.) However, no aggregate-parallel operations are allowed, because their interpretation depends on the shape of the collection. Such operations must be defined in terms of a collection. A collection imposes a shape on a plural object, and thus determines the type of operations that make sense for the collection. For example, a vector collection type may define `reduction` and `parallel-prefix` operations, and a 2-D array collection type may define `transpose`. The operations of `parallel-prefix` and `transpose` do not make sense on the components of a plural data type: they only have meaning in the context of a defined shape.

These notions suggest a representation of a collection. A collection must include two things:

- a plural variable,
- and a *shape descriptor*.

The shape descriptor is a data structure that describes how to arrange the components of the plural variable into the shape of the collection.

3.4 An example: vector collections

The simplest type of collection is the vector. A vector collection has one plural variable, and a shape descriptor called a "segment descriptor" which groups the components of a plural variable into logically contiguous "segments." The C++ class representing a vector of type `T` is the following.

```
class v<T> {
    T             val;
    segment_desc  segd;

    // ... member functions follow ...
};
```

A segment descriptor is a sequence of numbers specifying the division of the components of the underlying plural variable into segments. The length of the sequence is the plurality of the collection, and the sum of the numbers in the sequence must match the plurality of the underlying variable.

For example, a vector of type `v<T>` of plurality one has a segment descriptor of length one whose value matches the plurality of the underlying variable of type `T`. Given a vector whose value is `[t1 t2 t3 t4 t5 t6]`, its members would have the following values.

```
val  ==>   t1 | t2 | t3 | t4 | t5 | t6
segd ==>   { 6 }
```

A vector whose plurality is greater than one is written as a sequence of vectors separated by vertical bars. The following vector has type `v<T>` and plurality three.

```
[ t1 t2 t3 t4 ] | [ t5 t6 ] | [ t7 t8 t9 ]
```

Its members would have the following values.

```
val  ==>   t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9
segd ==>   { 4 2 3 }
```

The length of the segment descriptor is 3 and gives the plurality of the collection. Its values (`{ 4 2 3 }`) describe the *lengths* of each of the components of the (plural) vector.

Of course, vectors may be nested. Let type `T` itself be a collection type, then the the `val` member of type `v<T>` will be a collection with another segment descriptor. Consider a vector of type `v<v<T> >` of plurality two.

```
[[t1]] [t2 t3 t4]]  |  [[t5 t6 t7] [t8 t9]]
```

Its members have the following values.

```
val  ==>  [t1] | [t2 t3 t4] | [t5 t6 t7] | [t8 t9]
segd ==>  { 2 2 }
```

The `val` member of the collection has plurality four, and the segment descriptor of the collection arranges these four components into two segments. Because each element of the collection is also a collection, the `val` member has its own internal structure.

```
val.val  ==>  t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9
val.segd ==>  { 1 3 3 2 }
segd     ==>  { 2 2 }
```

Member `val.val` has type `T` and plurality nine, and the first level segment descriptor divides these nine components among four segments to produce a vector of type `v<T>` and plurality four. The second level descriptor divides these four components into two segments to produce a vector of type `v<v<T> >` and plurality two. In general, a nested vector has as many segment descriptors as there are levels of nesting.

3.5 The translation scheme

A `foreach` construct strips away the shape descriptor of a collection, causing a variable to refer to the underlying plural variable, instead of the collection. Through a simple translation scheme, we are able translate a `foreach` construct so that inside the construct the variable names refer to the plural variable of the collection *without* the context of a defined shape. The advantage of this approach is that the translation is very simple, and the type system of C++ is used to resolve the references to operators and functions inside the `foreach`. The resulting code is *type safe*.

The translation of a `foreach` construct involves merely changing the variables of its argument list to refer to the `val` field of the collection instead of the entire collection. Reference types are used to implement the change in type, and the C++ type system takes care of the rest. The following code

```
v<Int>  a(size(10)), b(size(10)), c(size(10));
foreach (a, b, c) {
    a = b + c;              // foreach body
}
```

becomes

```
v<Int>  a(size(10)), b(size(10)), c(size(10));
{
    Int &ar = a.val, &br = b.val, &cr = c.val;
    {
        Int &a = ar, &b = br, &c = cr;
        {
            a = b + c;              // foreach body
        }
    }
}
```

after translation. A reference is like a synonym for a variable in C++. Initially, variable `a` refers to a vector. Variable `ar` is a reference to the `val` field of `a`, and has type `Int` and plurality ten. The next block creates a reference to `ar` named `a`. Now, when the `foreach` body is compiled the name `a` refers to a variable of type `Int`. Note that the `foreach` body is not changed in any way.

By using references, the translation arranges for each variable of the `foreach` list to refer to its `val` member. The reference then actually modifies the `val` member of the collection. (Note: uninitialized collections are given a shape descriptor in the epilog created in the translation. In the example above, all collections were defined so no epilog was generated.)

Note the various features of C++ used here: scoping, variable references, static type checking, and function/operator name resolution. This translation uses these features to change the effective types of the variables inside the `foreach` construct. Its body remains textually unchanged. While this transformation may seem to increase the code by a large factor, the new statements merely initialize references (pointers) and are trivial. Even a simple compiler should be able to optimize the few redundant initializations performed.

3.6 Conditionals and loops

Our present compilation strategy for conditionals is similar to that used by vectorizing compilers [18] and also by NESL [5]. We give only a brief overview of the technique here.

Note that the plurality of all variables inside a given `foreach` construct must be the same. Thus, we speak of a foreach construct having plurality p. When an `if` statement is encountered inside a `foreach`, the conditional expression is evaluated and stored in a special variable called the *mask*. The mask is a boolean variable of plurality p that has p_t true components, and p_f false components.

At this point, the run time system splits each live variable in the `foreach` body, v, into two parts. One part, v_t, contains the components corresponding to the true components of the mask and has plurality p_t. The other part,

Table 1: Times measured for the small version of the NAS Conjugate Gradient benchmark on various machines.

CG Benchmark	
Sun 4/370	143 s
8K CM-2	40 s
32 Proc. CM-5	21 s

v_f contains the components corresponding to the false components of the mask and has plurality p_f. The translator arranges for the `then` clause of the conditional to be executed by replacing each variable v with v_t, and the `else` clause by replacing each variable v with v_f. Result vectors of the two branches (of plurality p_t and p_f) are then merged using the mask to produce final values of plurality p.

The drawback of this approach is that a conditional may result in a significant amount of communication. We are currently examining other translation techniques that may be more efficient. Loops are handled in a similar fashion, except that the live variables are successively split at each trip through the loop.

3.7 Summary

This section sketched the procedure for using program translation to flatten nested parallelism. While we omitted many details (the runtime system, the parallel extension of variables across `foreach` constructs, etc.), the procedure described captures the essence of the compilation system.

Other descriptions of nested parallelism have described requiring two functions [4, 14]. In the terms presented here, these systems allowed the user to write a function in terms of a variable of plurality one, and the compiler extended it to operate on values of greater plurality. We believe that our method simplifies the notion of flattening nested parallelism. Because *all* functions are written in terms of plural variables, only one version of any particular function is required.

4 Experiments

For an initial experiment, we wanted to demonstrate the feasibility of our approach by implementing a small subset of the language and runtime system. We also desired to show that the system could be fairly portable. Our goal was to demonstrate code with nested parallel constructs that ran on a variety of machines. For this simple test, we selected the small version of the NAS Conjugate Gradient (CG) benchmark [2]. Our translator is currently somewhat incomplete, so we manually translated the `foreach` constructs using the method described earlier.

To date, we have ported the base classes to two different runtime systems. The first is a straightforward implementation for serial computers. The second runtime target is the CVL vector library [3] which runs on workstations, Cray Y-MP and C90 computers, the CM-2, and the CM-5. Another port of CVL from UNC also runs on the MasPar MP-1 [11]. By using this library we hoped to run our code on each of these machines.

However, we ran into some problems because of the current state of C++ compilers on some of these systems. The CRAY C++ compiler is ATT `cfront` based and does not handle templates correctly. Some of the other machines did not have C++ compilers installed. For example, the CM-2 we used did not have a C++ compiler, but we managed to produce object files on another Sun Sparcstation and to link these with the necessary runtime libraries on the CM-2 front end. We tried a similar technique on the MasPar MP-1, because it did not have a C++ compiler installed either, but discovered an inconsistency in object file formats. We expect these problems to be worked out in the future as C++ becomes more widely used and demand for support of the language increases.

In the end, we successfully ran the benchmark on a Sun-4/370 workstation, a CM-2 and a CM-5. The CG benchmark is a good test of nested parallel constructs. It uses the sparse matrix-vector multiplication function shown earlier, and also requires the computation of a dot-product. Table 1 presents the execution times measured on each of these machines.

The times reported are actually quite poor. In performing these tests we discovered a large number of inefficiencies that we plan to correct shortly. (For instance, default constructors for segment descriptors were called in many places we did not intend.) However, we were encouraged by the fact that we managed to demonstrate a functioning system in a short time with a minimum of effort. In the next few months we expect to improve these execution times by an order of magnitude, and to complete implementations on the other machines as C++ support improves.

5 Conclusions

We have explained the implementation of a programming language and runtime system that supports nested data-parallelism in an object-oriented framework. Our system makes use of many of the features already in C++ and only adds one new language feature: the `foreach` construct. Our compilation process translates nested parallel code into regular C++ which can then be processed by regular compilers. The advantages of this approach are that users retain the features of C++ (including data hiding, security and inheritance), and gain access to parallel types that can be composed to create nested parallel types.

Our goal was to produce an extensible prototype system in which to experiment with these ideas. There are many directions of research that we intend to pursue. Other collection types must be defined: we will probably implement an `array` template next. We also intend to investigate the issue of distribution, and will develop constructors that specify not only the `size` of an object, but also its distribution over the available processors. We also intend to provide higher-level communication operators, such as the `match` and `move` operators [17].

Our current implementation is based on the C Vector Library (CVL). Using this library allowed us to quickly port the system to a variety of machines, but limits the possibilities for optimization. Each operator in our language results in a separate call to a vector function, and the amount of synchronization performed is high. A native implementation of the base classes for each target machine would allow the opportunity to apply sophisticated optimizations like loop fusion or loop unrolling on a per-node basis [9]. As we understand the language and compilation system better, we should be able to deliver much higher performance than that achieved in this prototype system.

References

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

[2] D. Bailey, D. Browning, R. Carter, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[3] Guy Blelloch, Siddhartha Chatterjee, Jay Sippelstein, and Marco Zagha. CVL: a C Vector Library. School of Computer Science, Carnegie Mellon University, 1991.

[4] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, 1990.

[5] Guy E. Blelloch. NESL: a nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992.

[6] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[7] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of Supercomputing '93*, pages 588 – 597, November 1993.

[8] K. Mani Chandy and Carl Kesselman. The CC++ language definition. Technical Report Caltech-CS-TR-92-02, California Institute of Technology, 1992.

[9] Siddhartha Chatterjee. *Compiling data-parallel programs for efficient execution on shared-memory multiprocessors.* PhD thesis, Carnegie Mellon University, 1991.

[10] Andrew A. Chien and William J. Dally. Concurrent aggregates (CA). In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 187–196, February 1990.

[11] Rickard E. Faith, Doug L. Hoffman, and David C. Stahl. UnCVL: The University of North Carolina C vector library. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, 1993.

[12] James R. Larus, Brad Richards, and Guhan Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical Report UW 1126, Computer Sciences Department, University of Wisconsin-Madison, November 1992.

[13] Peter Mills, Lars Nyland, Jan Prins, John Reif, and Robert Wagner. Prototyping parallel and distributed programs in Proteus. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 10–19, Dallas, Texas, Dec.1-5, 1991. IEEE.

[14] Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 119–128, May 1993.

[15] John R. Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing*, May 1987.

[16] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* Series in Artificial Intelligence. MIT Press, 1988.

[17] Thomas J. Sheffler. *Match and Move, an Approach to Data Parallel Computing.* PhD thesis, Carnegie Mellon University, 1992.

[18] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.