WG14/N336

X3J11/94-021

Adding Shapes to Iterators
WG14/N336
X3J11/94-021

Bill Homer
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121

April 26, 1994

Abstract

The notion of "shape" has been previously proposed as the basis of a parallel programming model for C. It was used to add arrays as first-class objects to C, with support for both declarations of multi-dimensional, distributed array objects, and operations on those objects. This paper explores an alternative approach, which combines the "declaration part" of shape with the a previously proposed *for-all* construct, using iterators. The goal is a relatively small language extension which offers an effective parallel programming model.

1 Introduction

In the interest of brevity, this paper does not attempt to be complete and self-contained. Examples rather than formal syntax rules will be used for the exposition.

The previous proposal involving shape mentioned in the abstract may be found in "Data Parallel C Extensions" (X3J11.1/93-034).

A complete description of iterators may be found in "Using Iterators to Express Parallel Operations in C" (X3J11.1 93-050). Recall from that proposal that there are two new storage classes, spelled `iter` and `unord`. When used in a statement, each variable with one these storage classes implies an iteration, which is either ordered (as in an ordinary C `for` statement), or unordered (so that the different iterations of the loop can be executed in any order, or in parallel). Multiplication of a 100×300 array by a 300×200 array can be expressed as in Figure 1. The last statement is the body of three implied nested loops, with an ordered loop for the accumulation of the sums, but unordered loops over the elements in the result array.

315

Figure 1 *matrix multiply using iterators*

```
extern double a[100][200], b[100][300], c[300,200];
unord int I = 100, J = 300;
iter int K = 200;

a[I][J] += b[I][K] * c[K][J];
```

2 The notion of shape

There is a new type definition, using a new keyword spelled shape.

Figure 2 *Definition of a shape*

```
shape S[100][200];
```

A shape may also include distribution information. This is an important option for distributed memory environments, but for simplicity it will be omitted from the examples hereafter. The syntax is based on that proposed for arrays in "Distributing Data Using the "block" Qualifier in C" (X3J11.1 92-033).

Figure 3 *Definition of a distributed shape*

```
shape dS[100 block(10)][200 block(20)];
```

3 Shaped arrays

A shape is an aggregate object which may be used only in a limited number of contexts. One of them is a new variant of an array derivation.

Figure 4 *Definition of a shaped array*

```
double a[S];
```

The declaration of a reserves storage for a 100×200 array of doubles, much as would two ordinary array derivations. Unlike the ordinary array derivations, there is no guarantee that the storage for the elements of a shaped array is contiguous. Furthermore, a reference to a shaped array in an expression is not converted to a pointer to its "first" element. In fact, in most contexts, a reference to a shaped array must be fully indexed to yield a reference to an element, as in a[0][0].

4 Shaped iterators

Rather than extending arithmetic and other operators to accept shaped arrays as operands, a shape can have an associated iterator. As might be expected, the iteration is over the

316

positions in the shape. The syntax allows a shape as the initializer for an iterator.

Figure 5 *Definition and use of a shaped iterator*

```
shape S[100][200];
double a[S], b[S];
unord int i = S;

a[i] = b[i] + 1;
```

Note that the assignment is the body of two implied nested unordered loops, so that the assignments of elements may proceed in parallel. This gives a more concise form of iteration than is available for an ordinary "array of array" type. The more verbose form is also available for shaped arrays, and in fact is needed to express operations that do not "respect" shape. For example, they must be used to express an assignment that transposes an array.

Figure 6 *Transpose of shaped arrays*

```
shape S2[100][100];
double c[S2], d[S2];
unord int i = 100, j = 100;

c[i][j] = d[j][i];
```

An interesting feature of this proposal is that it highlights the potential communication costs implied by moving data to new positions in a distributed memory environment. In Figure 5, the use of a shaped iterator makes it clear that there is no communication between different positions in the shape. In Figure 6, the use of multiple, ordinary iterators is required to express such communication.

5 Pointers and function calls

It is possible to have a pointer to, but not *into*, a shaped array. In other words, the pointer points to the whole shaped array object rather than any of its elements.

To provide access to a shaped array in a called function, a shape can be declared as a function parameter, and passed as a function argument. In the example, the rank of the shape is unspecified, but it would be easy to extend the syntax to indicate the rank.

Figure 7 *Shape function parameter*

```
void f(shape S, double (*a)[S]) {
    unord int i = S;
    (*a)[i] += 1;
}
```

The "extra" level of indirection in Figure 7 is unattractive, and can be avoided by allowing a shaped array argument to be passed to a shaped array parameter. Note that there is no notion of a shaped value, only a shaped lvalue, and so the argument must always be an lvalue. The array is still passed by reference, but now the indirection is implicit.

Figure 8 *Shaped array function parameter*

```
void f(shape S, double a[S]) {
    unord int i = S;
    a[i] += 1;
}
```

Finally, it seems that a general programming model must include some notion of a slice, i.e., a shaped array subobject. This is useful both for function parameters and for declaring a new object that is aligned with a subobject of a shaped array. For this we introduce another new keyword, `slice`, and borrow some syntax from "Array Syntax and Tensor Values" (X3J11.1/91-002). In Figure 9, `b` is aligned on the positions in `a` having both index values even. The assignment moves values from positions in `a` to corresponding positions in `b`. Like a shape, a slice can be passed to a function, and used in the declaration of other parameters.

Figure 9 *Slices*

```
shape S[100][200];
slice S[0;50;2][0;100;2] S1;
double a[S], b[S1];
unord i = S1;

void g(slice S1, double c[S1]) {
    unord int i = S;
    c[i] += 1;
}

b[i] = a[i];
g(S1, a);
g(S1, b);
```

6 Concepts avoided

By not introducing shaped arrays as first class objects, we get to avoid some things that would make the language extension much larger. In particular, we avoid the notion of shaped values, with the attendant new expression syntax and semantics. We also avoid contextualization and the associated questions about the semantics of control flow.

7 Conclusion

Obviously, this is just a sketch, not a complete proposal. Nevertheless, I believe that these ideas could be the basis of a fully specified proposal that would provide a relatively simple and attractive parallel programming model.