C++ and Massively Parallel Computers

DANIEL J. LICKLY AND PHILIP J. HATCHER

Department of Computer Science, University of New Hampshire, Durham, NH 03824

ABSTRACT

Our goal is to apply the software engineering advantages of object-oriented programming to the raw power of massively parallel architectures. To do this we have constructed a hierarchy of C++ classes to support the data-parallel paradigm. Feasibility studies and initial coding can be supported by any serial machine that has a C++ compiler. Parallel execution requires an extended Cfront, which understands the data-parallel classes and generates C* code. (C* is a data-parallel superset of ANSI C developed by Thinking Machines Corporation.) This approach provides potential portability across parallel architectures and leverages the existing compiler technology for translating data-parallel programs onto both SIMD and MIMD hardware.

1 INTRODUCTION

The data-parallel programming model is based on the simultaneous execution of the same operation across a set of data [1]. Most scientific and engineering problems, and many others, have data-parallel solutions. The model's single locus of control simplifies design, implementation, and debugging of programs. The model is high level, which greatly enhances the portability of programs across sequential, SIMD, and MIMD platforms. For these reasons data-parallel programming environments are a crucial requirement to allow a wide range of users to exploit massively parallel computers.

To be most effective a data-parallel programming environment must have three important characteristics.

1. It should be based on synchronous execution semantics. The programming model presented to the user should be single threaded.
2. It should support virtual processors. The programmer should be able to write programs independent of the number of physical processors available.
3. It should provide a shared-memory programming model. The cost of accessing the memory may be nonuniform, but the user should be able to access all locations by name.

We characterize systems with these three properties as strongly supporting data-parallel programming.

C* is a data-parallel superset of ANSI C developed by Thinking Machines Corporation [2]. C* strongly supports data-parallel programming. C* programs can be translated for efficient execution on both SIMD and MIMD parallel hardware [3]. However, the fact that C* is a superset of C is problematic for many potential users. First, being a superset, C* requires programmers to learn new syntax and semantics. Second, many users would

Received April 1993
Revised June 1993

This work was supported by National Science Foundation grant CCR-8906622.

Scientific Programming. Vol. 2, pp. 193–202 (1993)
CCC 1058-9244/94/040193-10

prefer that C* be based on C++ rather than C. They would like to use the object-oriented methodology when programming parallel computers. This article reports on an initial study of the feasibility of integrating the C* programming model within unaltered C++.

The template feature of C++ version 3.0 offers a powerful mechanism to emulate the data-parallel constructs of C* without leaving the syntax of C++. A template can be used with a "parallel object" class to allow instantiation of parallel objects of different types. Our goal is to exploit the potential efficiency and portability of explicitly data-parallel languages within the context of conventional C++ code.

The first step has been the creation of a hierarchy of C++ classes to support the data-parallel paradigm as presented in C*. These classes may be used to define parallel objects. The resulting aggregate objects may be manipulated elementally by most of the C++ operators in a fashion similar to the C* aggregate operators. An organization of these classes has been defined and implemented in C++ and can be executed on any machine with a C++ version 3.0 compiler.

Our future complementary implementation strategy will be to produce a C++ to C* translator. This translator will exploit knowledge of the data-parallel class hierarchy in order to generate C* code. This will provide automatic portability to any parallel platform that has a C* compiler.* The strategy also allows us to leverage the rapidly developing compilation technology for data-parallel programming languages.

2 RELATED WORK

There is currently intense interest in object-oriented programming, and even in object-oriented parallel programming. A host of research projects are underway investigating the use of object-oriented programming languages on parallel or distributed computing platforms. There is even a large number of projects studying C++ for parallel hardware. Our work can be distinguished from these projects by our design goals:

1. Support strongly the data-parallel paradigm.
2. Tolerate no changes to C++.
3. Allow "automatic" generation of implementations for parallel hardware.

Mentat [4] and CC++ [5] are both dialects of C++ designed to exploit parallel hardware. Both languages are most effective as SPMD programming environments. Neither language strongly supports the data-parallel model as they require the programmer to explicitly perform synchronization and communication through dataflow-like mechanisms. Both languages are extensions to C++.

pC++ [6] and C** from the University of Wisconsin [7] both strongly support the data-parallel model. However, both languages are extensions to C++. pC++ also differs from our work in that it is building on the high performance Fortran instantiation of the data-parallel programming model, whereas we begin with the C* programming model. C**-Wisconsin is based on a "copy-in, copy-out" semantics for the application of functions to parallel aggregates. We believe the implementation of such semantics is problematic in C-based languages, at least in the short term. We offer alternative function application semantics similar to what is proposed for pC++.

DPar is a language for which preliminary designs were investigated at Los Alamos National Laboratory [8]. DPar attempts to extract C*'s data-parallel extensions to C and apply them to C++. The key difficulty, which is also encountered in our work and the work with pC++, is deriving rules for allowing objects to have both parallel and sequential instantiations. Although the DPar approach would strongly support data-parallelism, the language is constructed by extending C++ with C* constructs. Our approach is also to try to apply the C* extensions to C++, but within the existing syntax and semantics of C++.

A number of efforts are investigating the construction of application-specific class libraries with internal parallelism [9–11]. The basic approach is to design a library with broad applicability and then have experts port its internals for each parallel machine. For the appropriate applications this approach strongly supports the data-parallel paradigm. The user may be totally unaware that multithreaded hardware is being exploited. This approach does not rely on any extensions to C++. However, the internals of the libraries must be ported "by hand" for each target

* This, of course, includes Thinking Machines' Connection Machine family. In addition, in another project, we are investigating the implementation of C* for a variety of MIMD hardware, including the Intel Delta/Paragon [14] and clusters of Unix workstations.

machine. The goal of our approach is to allow a C* compiler to be used to provide portability of user-written application-specific classes.

This goal is shared by the C** programming environment being developed by researchers at the Australian National University [12]. They also support the C* programming model in C++ by providing an appropriate set of classes. They embed macro calls in the C++ code and have instantiations of these macros for either generating "straight" C++ code or for generating C* code. Their motivation was to provide a C*-like programming environment on workstations for teaching purposes. Consequently, their approach does not combine the object-oriented and data-parallel paradigms—rather they use the object-oriented features of C++ to provide a C* environment. Users are constrained in what objects can be instantiated as parallel by the need to generate C* code by macro expansion. Our eventual strategy is to provide an extended Cfront (Cfront*) that both converts C++ features to C and implements parallel objects via C* code.

3 THE C* PROGRAMMING LANGUAGE

Our work is based on integrating the C* programming model within unaltered C++. This section describes the C* programming model and is organized around the example program in Figure 1. This program implements the Jacobi algorithm to find the steady-state temperature distribution on an insulated two-dimensional (2-D) plate, given constant boundary conditions.

C* introduces a new data type, the shape type, that describes array-like objects that are operated on in parallel. A shape specifies the rank, dimensions, and layout of explicitly parallel data. Shapes can be used in the declaration of arithmetic, structure, and union types. When a declaration includes both a base type and a shape type, then a parallel object is declared consisting of an object of the base type at each position of the shape. Line 5 of Figure 1 declares a shape and lines 6–9 declare a set of parallel variables.†

Parallel objects can be operated on in parallel via the overloading of standard C operators. The operator is applied as if simultaneously at each position of the shape. To specify a parallel operation, a current shape must first be established by a

```
1.  #include <math.h>

2.  #define SIZE    128       /* Resolution of grid */
3.  #define TEMP    50.0      /* Arbitrary cut-off  */
4.  #define EPSILON 0.1

5.  shape [SIZE][SIZE]cell;

6.  bool:cell  active;        /* 0 if cell boundary; 1 otherwise */
7.  float:cell change;        /* Change in temperature */
8.  float:cell new;           /* Newly calculated temperature */
9.  float:cell old;           /* Previous temperature */

10. main () {
11.   float maxerr;           /* Largest change in temp. over grid */
12.   int   cool_cells;       /* Number of cells with temp. < TEMP */

13.   with (cell) {           /* Initialize grid */
14.     where ((!pcoord(0)) ||  (!pcoord(1)) || (pcoord(1) ==
15.              (SIZE-1))) {
16.       active = 0;
17.       old = new = 0.0;
18.     } else where (pcoord(0) == (SIZE-1)) {
19.       active = 0;
20.       old = new = 100.0;
21.     } else {
22.       active = 1;
23.       new = 50.0;
24.     }

25.     do {                  /* Compute steady-state temperatures */
26.       where (active) {
27.         old = new;
28.         new = ([.-1][.]old + [.+1][.]old +
29.                [.][.+1]old + [.][.-1]old) / 4.0;
30.         change = fabs(old-new);
31.         maxerr = >?= change;
32.       }
33.     } while (maxerr > EPSILON);

34.     cool_cells = (+= (new < TEMP));
35.   }

36.   printf ("There are %d cells cooler than %5.2f degrees\n",
37.     cool_cells, TEMP);
38. }
```

FIGURE 1 Example C* program for the Jacobi algorithm.

with statement. A where statement is provided for masking off positions of the current shape. Lines 13–24 of Figure 1 use the overloaded assignment operator to initialize parallel variables. The pcoord intrinsic function creates a parallel value of the current shape with each position containing its coordinate along the specified axis.

C* also overloads most of the assignment operators to perform reductions of parallel values to a single sequential value. Line 34 of Figure 1 uses the +=operator to perform a sum reduction computing the number of positions whose new temperature is less than the cut-off temperature, TEMP.‡

† C* also includes a Boolean data type, used in line 6 of Figure 1.

‡ C* has added two new binary operations for min, <?, and max, >?, which also have reduction forms, <?= and >?=. The max reduction operator, >?= is used on line 31 of Figure 1 to control the enclosing loop.

C* allows parallel variables to be indexed. However, parallel variables are not arrays and indexing a parallel variable may be expensive on machines where the parallel variable is spread across the distributed memories of a set of processors. To emphasize this potential inefficiency, the index operator is moved to the left of the indexed variable. When indexed by sequential indices, the left index operator produces a sequential value. When indexed by parallel indices, the left index operator produces a parallel value.

The dot is used as a shorthand notation within a left index expression. The dot stands for a `pcoord` call on the axis being indexed. This allows for the convenient referencing of neighboring values in a shape. Lines 28–29 demonstrate the reference of the four neighboring values in a 2-D shape.

4 C++ CLASSES FOR DATA-PARALLELISM

This section describes the classes that we have designed and implemented to support a C* programming model within C++.

4.1 Shapes

A Shape class is provided to define shape objects that perform the same role as C* shapes. They contain the basic sizing and organization information needed to define parallel variables. Parallel variables are only compatible if they have the same underlying shape object. Constructors are provided for optionally defining the rank and dimensions of the shape.

Examples:

```
Shape shp1 (7);
Shape shp2 (64, 64);
Shape shp3;
```

The Shape class includes member functions that emulate C* intrinsics: `positionsof`, `rankof`, and `dimof`.

Examples:

```
int i,j,k;

i = shp1.positionsof();
j = shp1.rankof();
k = shp2.dimof(1);
```

4.2 Parallel Variables

The `Pvar` class template is the mechanism for defining parallel variables. `Pvar` takes a single template argument that specifies the type of the object being instantiated as a parallel. The constructors for `Pvar` objects take a shape as an argument, which defines the shape of the item being created.

Examples:

```
Pvar <float>      time (shp2);
Pvar <int>        distance (shp1);
Pvar <Polygon>    pg (shp1);
Pvar <Polyline>   pl (shp1);
```

The `Pvar` class includes member functions that emulate C* intrinsics: `shapeof`, `positionsof`, `rankof`, and `dimoff`. As in C* the latter three query the parallel variable's underlying shape.

Examples:

```
shp3 = pg.shapeof();
i = pl.positionsof();
j = distance.rankof();
k = time.dimof(1);
```

Additional member functions exist to emulate C* operators that do not exist in "overloadable" form in C++: the left-index operator and unary forms of the reduction operators. For sequential indices the left-index function produces a reference that can act as either an lval or an rval. (Parallel left indices will be discussed in Section 5.3.)

Examples:

```
i = distance.lindex(5);
time.lindex(8,i) = 1.7;
pl.positionsof();
distance.rankof();
time.dimof(1);
```

Note that the term "parallel variable" is a misnomer. The `Pvar` class is also used to represent intermediate results that cannot appear as lvals.

4.3 The Pcoord Intrinsic

The C* `pcoord` intrinsic is provided as a function that takes two arguments: a shape and an axis number. It returns an integer `Pvar` of the given shape. Choosing to implement `pcoord` as a function, and not as a member of the Shape class, was simply a matter of aesthetics—we prefer the function call syntax (`pcoord(shape,axis)`) in this

case over the member function call syntax ((shape.pcoord(axis)). We are still debating this issue, and in fact many of the intrinsics discussed above as member functions are replicated in the implementation as functions.

4.4 Manipulating Context

Shape objects include context information that indicates which positions of the shape are currently active (not masked off). Functions are provided for manipulating the context. In C* a syntactic construct, the where statement, allows the stacking of context information. Because we are unwilling to modify the syntax of C++, we provide a set of functions to stack and unstack contexts. The where function accepts a Pvar object, which must have an underlying integer type, and modifies the context associated with the shape of the Pvar object. As in the C* where, our where function pushes a new context that is formed from constricting the previous context according to the values in the input object. The endwhere function pops the top of the context stack. The elsewhere function pops the top of the context stack, complements the popped context, optionally constricts the complemented context, and then pushes the result.

Examples:

```
where (adub < sdub) ;
elsewhere();
elsewhere(adub > 9);
endwhere();
```

4.5 Function Semantics

C*, of course, allows functions that accept and return parallel values. These functions execute as if fully synchronous: as if there is a synchronization prior to entry; as if the body executes synchronously at the operator level (as if the operands are fetched at all positions of the active shape before the operator is evaluated at any position); and, as if there is a synchronization upon return. This functionality is easily emulated in our system by functions that accept and return Pvar objects.

In addition our system provides elemental functions. Elemental functions are executed as if there is a function call performed at every position of the active shape. Each invocation of the function proceeds as if independently with only a synchronization upon return. This requires that the elemental function have no state between calls (no static locals) and no side effects (no writing to globals). Member functions of classes that are provided as template arguments to the Pvar class have elemental semantics when applied to parallel objects. This imposes a restriction on the classes that can be instantiated as parallels: member functions that will be applied to parallels must have no state and no side effects.

We also allow functions to be declared to be elemental. This is particularly useful for providing parallel versions of the common math functions: sin, cos, etc.

4.6 The Pvar Class Hierarchy

Actually there are a collection of classes that provide the parallel variable capability. There are a set of base classes that are abstract, and concrete classes derived from them. The members of this class hierarchy are:

1. Pbase0: an abstract class that serves as the base for all parallel variables. It has all the basic operations that are common to all classes.
2. Pvar: a concrete class that is derived from Pbase0. It is used for collections of classes or structures without overloadings of the usual arithmetic operators.
3. Pbase1: an abstract class derived from Pbase0. This class has overloadings for the usual arithmetic operators.
4. Parith: a concrete class derived from Pbase1. It has full support for the arithmetic operators, and expects its template argument to also fully support the arithmetic operators. Parith is used mostly with the builtin arithmetic types, int, float, etc.
5. Pbase2: an abstract class derived from Pbase1. It supports parallel container classes for parallel arithmetic objects. It adds generic operations for groups of arithmetic objects.
6. Ptensor: a concrete class derived from Pbase2. It is a generic class for compound arithmetic types that are treated in a uniform fashion.

The abstract classes are not directly used by users. They simply support the implementation of the concrete classes.

The class hierarchy exists primarily to ease the burden on the programmer. The user of the Parith class must provide overloadings for all the

usual arithmetic operators for classes supplied as the template argument to `Parith`. A programmer solving a non-numeric problem might find the `Pvar` class more convenient. However, the `Parith` class allows arithmetic expressions involving parallel variables to be written in the C* style—a very concise notation. For example:

```
Shape shp(100);
Parith<double> x(shp);
Parith<double> y(shp);
Parith<double> z(shp);

x = y + z;
```

The overloading of the arithmetic operators requires that parallel operands be of the same shape. Sequential operands are promoted to parallel.

The user may also derive classes using this class hierarchy. We have experimented with this facility by building two additional classes. The `Pvector` class is derived from `Pbase2` and supports parallel variables containing a vector at each position. The `Pmatrix` class supports parallel variables containing a matrix at each position.

4.7 The Jacobi Example Recast

Figure 2 recasts the C* program of Figure 1. Line 1 provides access to our parallel variable class hierarchy. Line 2 provides elemental definitions of the standard math functions. (Line 30 contains an elemental application of the `fabs` function.) Lines 28–29 describe the four-point Jacobi update. We have used functions to hide the complex left-index function calls that are required. The complexity is due to the fact that we do not have a

```
1.  #include <Pvar.h>
2.  #include <Pmath.h>

3.  #define SIZE    128       /* Resolution of grid */
4.  #define TEMP    50.0      /* Arbitrary cut-off  */
5.  #define EPSILON 0.1

6.  Shape cell(SIZE,SIZE);

7.  Parith <int>   active(cell);       /* 0 if cell boundary; 1 otherwise */
8.  Parith <float> change(cell);       /* Change in temperature */
9.  Parith <float> new(cell);          /* Newly calculated temperature */
10. Parith <float> old(cell);          /* Previous temperature */

11. main () {
12.   float maxerr;              /* Largest change in temp. over grid */
13.   int   cool_cells;          /* Number of cells with temp. < TEMP */

14.     where ((!pcoord(cell,0)) ||  (!pcoord(cell,1)) ||
15.               (pcoord(cell,1) == (SIZE-1))) ;
16.        old = new = 0.0;
17.        active = 0;
18.     elsewhere (pcoord(cell,0) == (SIZE-1)) ;
19.        active = 0;
20.        old = new = 100.0;
21.     elsewhere() ;
22.        active = 1;
23.        new = 50.0;
24.     endwhere();

25.     do {              /* Compute steady-state temperatures */
26.        where (active) ;
27.           old = new;
28.           new = (west(old) + north(old) +
29.                  east(old) + south(old)) / 4.0;
30.           change = fabs(old-new);
31.           maxerr = max(change);
32.        endwhere();
33.     } while (maxerr > EPSILON);

34.     cool_cells = sum(new < TEMP);

35.   printf ("There are %d cells cooler than %5.2f degrees\n",
36.      cool_cells, TEMP);
37. }
```

FIGURE 2 Example C++ program for the Jacobi algorithm.

feature corresponding to the C* dot operand. For example, the following implements north in terms of the left-index function:

```
old.lindex(pcoord(cell,0),modmod
    (pcoord(cell,1)-1,dimof(cell,1)))
```

Note that we also must provide a `modmod` integer function, corresponding to the C* %% operator, to support the torus-like behavior expected by the Jacobi program.

Lines 31 and 34 contain the max-reduction and sum-reduction operations, respectively, expressed as functions.

Another example program is given in the Appendix.

5 LIMITATIONS AND DIFFERENCES FROM C*

Our current approach has several significant limitations. We also provide a programming model that differs from C* on a few points. This section itemizes these shortcomings and differences.

5.1 No Overloading of the Selection Operator

A significant limitation in our approach is encountered because the member selection operator cannot be overloaded in C++. When a parallel variable is created with a base type defined by a user class, the user may like to elementally access the member functions via the dot operator.

For example, suppose a user has a class named `Circle` and a function in that class named `getRadius`. A parallel variable could be declared as follows:

```
Pvar <Circle> pc (shp1);
```

The user may very well then want to write:

```
pc.getRadius();
```

The intent is to have another parallel variable created in which each position contains the result of applying `getRadius` at that position. However, because we cannot overload the "." operator, the above will fail because the member function is being applied to a `Pvar` object rather than a `Circle` object.

To compensate we require the user to declare which member functions can be applied elementally. This declaration takes the form of a macro call applied to the member function name. For our serial implementation the macro expands into an overloaded global (i.e., nonmember) function definition of the same name. The function iterates through the active positions and calls the member function on the object stored at each position. Therefore, instead of member function call syntax, the user must utilize a conventional function call:

```
getRadius(pc);
```

A benefit to this approach is that global functions can also be declared to be invoked elementally on a parallel object. These declarations, as with the case of member functions, take the form of a macro invocation. Again, for our serial implementation, a new function is generated that accepts and returns parallel values by repeatedly calling the underlying function at each position.

The macros are complicated by the need to have the return type as well as all parameter types provided as macro arguments. Currently, different macros are provided for different numbers of parameters. For the above example the macro call would be:

```
Pelemental_member_0
  (double, getRadius);
```

For a global function that takes two integer parameters and returns a double, the macro call would be:

```
Pelemental_global_2
  (double,globfuncname, int, int);
```

Elemental functions must adhere to the restrictions outlined in Section 4.5. These restrictions require that the function keep no state and modify no globals. Our current implementation has no means to enforce these restrictions however.

5.2 Lack of a Syntactic Where Construct

C* has a `where` statement, explicit syntax to support the manipulation of context. The syntax cleanly supports the nesting of context manipulating operations. For example, the popping of context is done automatically upon exit from the `where` statement.

Our system implements context manipulation via function calls, with no compile-time checking of constraints on the placement of the functions. However, to allow "clean" C* code to (eventually) be generated, the programmer must use our context manipulation functions in a manner supported by the C* `where` and `else` statements.

The user is also responsible for performing an explicit `endwhere` at the appropriate times. This is clearly a low-level and potentially error-prone operation. This is one clear situation where our refusal to introduce new syntax has burdened the programmer.

We do enforce a run-time constraint on the placement of these functions. Neither the `endwhere` nor the `elsewhere` statement take a shape as a parameter. These functions are assumed to manipulate the last (at runtime) stacked context.

5.3 Problems with Lvals

Currently, parallel left-index operations may only be used as rvals. Our implementation produces a new parallel value, rather than a reference. We cannot produce a reference because this could produce incorrect semantics in the case where a value is wanted (in the presence of other operations that might be side-effecting the variable being referenced). And, at the function call level, we cannot tell whether the result will be used as an lval or an rval.

Our eventual solution most probably will be to provide two functions: one to create a value and one to produce a reference. This again is a situation where we require the programmer to program at a slightly lower level than in C*.

A similar problem exists with elemental functions that return references. Our implementation currently produces a parallel variable containing the values, rather than the references.

5.4 Conversion Difficulties

The usual C/C++ conversions that apply to arithmetic types do not work automatically for our parallel types. The attempt to combine a parallel int and parallel double will generate an incompatible type error by the C++ compiler. It would be nice if the C++ compiler (and language) would produce the usual conversions in these cases. We are still investigating workable solutions. Extensive overloading of all the possible combinations is a potential but lengthy solution.

5.5 Shape Checking

All checking to ensure that variables have identical shapes is done at runtime. As in C*, this means that the two shapes must be derived from the same shape variable, not just that the shape has the same form. However, we do not support the C* compile-time "intermediate shape equivalence test on parallel variable usage" [2, p. 17].

5.6 C* Extensions

Our system anticipates several changes to C* that we believe to be essential. For example, elemental functions are very useful for incorporating into parallel programs code written for serial applications. In addition they provide an escape hatch into a less constrained control flow model.

We also support a limited form of parallel pointers. We do not preclude a user class that contains a pointer from being instantiated as a parallel object. Our interpretation is that the pointer will point "locally," within the same position of the shape. (Of course, in our current implementation we cannot provide any compile-time check to ensure the assumption is accurate.)

Finally, our system does not support the C* notion of "current shape." There is no `with` construct to establish the current shape, and code using different shapes can be mixed. For example,

```
Shape s1(100);
Pvar <int> x(s1);

Shape s2(50,50);
Pvar <int> y(s2);

int z;

z = sum(x) + sum(y);
```

All these changes are being discussed by the Data-Parallel C Extensions (DPCE) subgroup of the Numerical C Extensions Group (NCEG, ANSI X3J11.1). DPCE has taken the C* reference manual as its "base document" and hopes to provide a final set of recommendations based upon additions and deletions to C*. We anticipate features similar to those provided by the above changes to appear in the final DPCE report.

6 IMPLEMENTATION STRATEGY

6.1 Sequential Implementation

Our serial implementation has been done on a workstation using the Gnu C++ compiler. We are using version 2.3.3. It should be compatible with C++ version 3.0.

The serial implementation was undertaken to verify feasibility of concept and to learn how to develop a language set that was truly compatible with C++; would be compilable by a current C++ compiler; and would execute with the desired results. What we have attempted to produce is a run-time model that will cope with the myriad of C++ class and function combinations and yet be compatible with the C* execution paradigm.

The implementation is woefully inefficient and no attempt has been made to correct this. The implementation is characterized by many small loops with the results stored in temporaries that are allocated just for that purpose. No attempt has been made to manage the temporaries in a wise fashion. Likewise, no attempt has been made to reduce the many low-level loops by loop fusion or by other analytical techniques.

6.2 C* Implementation

We believe that an effective C++ to C* preprocessor can be built. The obvious reason for this belief is that we have designed our data-parallel class library with the special characteristics of C* in mind. Our shape class is the exact counterpart of the shape variable in C*. Our parallel classes are congruent to the parallel variable construct of C*. Therefore, we expect that the translation of our parallel constructs to C* to be a relatively easy process.

The nonparallel code is either straight C code or translatable to C as is done by Cfront. This should be handled in the same manner as Cfront does currently, because C* is nothing more than an extended C. So the C++ to C* translator that we envision is a Cfront with the extra capabilities to detect our special data-parallel class constructs and move them directly to C*. We call this special preprocessor, Cfront*.

The advantage of this approach lies in the leveraging of existing technology. Considerable progress has been made on the compilation of data-parallel programming languages like C*. By translating to C*, we allow the C* compiler to solve the problems of temporary management and loop fusion, at least for programs that manipulate parallel variables of the basic arithmetic types. For programs that use parallel variables of user class types, our approach inherits the compilation challenges of C++. Our code efficiency will be no better or worse than that produced by the existing C++ compilers.

7 CONCLUSIONS

We have investigated the feasibility of supporting a data-parallel programming model within unaltered C++. This is done by providing a hierarchy of data-parallel classes. Our system is based upon the C* programming model, but relies heavily on extensions to C*, such as the elemental function.

Our approach, although not changing the C++ syntax, does require the user to program at a lower level for some operations, most notably for context manipulation. We need to experiment with more test programs in order to better understand this tradeoff, as well as other advantages and disadvantages of our notation.

Work on a parallel implementation is still in the design stage. A key question to be researched is what limitations will need to be placed on the C++ programmer in order to allow the effective translation to C*.

Whether or not an explicit translation to C* is performed, we strongly believe that effective implementation of data-parallel C++ dialects on massively parallel architectures will require the incorporation of the technology that has been developed for the compilation of data-parallel programming languages. This requires the clear identification of operations that can be executed elementally, thus empowering an implementation to choose an execution order that makes best use of resources. And, in fact, this approach will also allow better code to be generated for serial machines than would be feasible using the conventional C++ compilation strategies.

APPENDIX 1

A Sample Program: Calculation of π

This program computes an approximation to π by using numerical integration to find the area under the curve $4/(1 + x^2)$ between 0 and 1 [3, 13]

```
#define INTERVALS 100000

main {
   Shape shp (INTERVALS);

   double sum = 0.0;

   Parith (double) r (shp);
   Parith (double) rr (shp);

   r = (pcoord(shp, 0) + .5)
       / INTERVALS;
   rr = 4.0/(1.0+r*r);

   sum += rr;

   sum = sum/INTERVALS
   cout << sum << " This is the PI
     sum \n";
}
```

REFERENCES

[1] W. Hillis and G. Steele Jr., "Data parallel algorithms," *Communications ACM*, vol. 29, pp. 1170–1183, 1986.

[2] J. Frankel, *A Reference Description of the C* Language*. Technical Report TR-253, Cambridge, MA: Thinking Machines Corporation, 1991.

[3] P. Hatcher and M. Quinn, *Data-Parallel Programming on MIMD Computers*. Cambridge, MA: MIT Press, 1991.

[4] A. Grimshaw, *An Introduction to Parallel Object-Oriented Programming with Mentat*, Technical Report 91-07, University of Virginia, 1991.

[5] K. Chandy and C. Kesselman, CC++: *A Declarative Concurrent Object Oriented Programming Notation*. Pasadena: California Institute of Technology, 1992.

[6] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang, *Distributed tC++: Basic Ideas for an Object Parallel Language*. Bend, OR: OON-SKI, 1993.

[7] J. Larus, B. Richards, and G. Viswanathan, *C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language*. Technical Report 1126, University of Wisconsin, 1992.

[8] S. Pope, *DPar—Data Parallel C++*. Los Alamos, NM: Los Alamos National Laboratory, 1990.

[9] T. Dennehy, *Class Libraries as an Alternative to Language Extensions for Distributed Programming. Proceedings of the USENIX Symposium on Experience With Distributed and Multiprocessor Systems*, 1992, pp. 313–321.

[10] D. Quinlan and M. Lemke, *P++, A Parallel C++ Array Class Library, OON-SKI, 1993*.

[11] A. Robinson, A. Ames, H. Fang, D. Pavlakos, C. Vaughan, and P. Campbell, *Massively Parallel Computing, C++ and Hydrocode Algorithms*. Albuquerque, NM: Sandia National Laboratories, 1992.

[12] T. Bossomaier and D. Stewart, *Abstract Data Types for High Performance Data Parallel Computing?* Canberra: Australian National University, 1993.

[13] R. G. Babb, II, *Programming Parallel Processors*. Reading, MA: Addison-Wesley, 1988.

[14] A. Lapadula and K. Herold, *A Retargetable C* Compiler and Run-Time Library for Mesh-Connected MIMD Multicomputers*. Technical Report 92-15, University of New Hampshire, 1992.