Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

RELATIONS + REDUCTIONS = DATA-PARALLELISM

V. Austel
R. Bagrodia
M. Chandy
M. Dhagat

March 1993
CSD-930009

Relations + Reductions = Data-Parallelism*

Vernon Austel+, Rajive Bagrodia+, Mani Chandy** and Maneesh Dhagat+

Abstract

The thesis of this paper is that a data-parallel language can be designed around two concepts: relations and reduction operations. There is a wealth of literature on the theory and implementation of relational database languages. Many parallel machines provide hardware support for reduction operations (such as summing all elements of an array), and these operations are widely used in parallel scientific computing. Our claim is that a data-parallel language can be created by coupling well-tested ideas on relations with well-tested ideas on reduction operations. This paper contains a description of a data-parallel extension of C based on these ideas. The extension, called UC, has been used for a variety of applications. UC compilers are available for the Connection Machine CM-2 and for sequential workstations. Compilers for the CM-5 and for multicomputers (such as networks of workstations and the Intel Paragon) are being developed. This paper expands on its central thesis, describes UC and presents performance measurements comparing programs written in UC with programs written in CM Fortran, C*, and *Lisp, executing on the CM-2.

+Computer Science Department, UCLA, Los Angeles, CA 90024

**Computer Science Department, Caltech, Pasadena, CA 91125

*This research was partially supported under NSF PYI Award No. ASC-9157610, ONR Grant No. N00014-91-J-1605, Rockwell International Award No. L911014, and Rome Laboratories Contract No. F30602-91-C-0061.

Contents

1 Introduction

The central theme of this paper is that a data-parallel language can be constructed by coupling well-tested concepts about relations with well-tested concepts about reduction operations.

1.1 Relations

There is a wealth of literature on relational databases. Here, we summarize the tiny bit that we need to describe our ideas. A relation is a table in which each row is unique, and in which each column has a type (eg. integer, string, float) and a name. All entries in a column are of the type specified for the column. A relation may have a *key* which is a column such that each row has a unique entry in the corresponding column. For example, a table of employees may have the social security number as a key. A relation may be ordered by giving an ordering on keys.

A linear array A of type T, indexed by i, $0 \leq i < N$, is a relation with two columns; the first is an integer *index* which is the index into the array, and the second, *value*, has type T. The index field is the key of the relation. There are N rows in the table, and there is a row in the table where the index field is i, and the value field is $A[i]$, for all i. The relation can be ordered in increasing order of index. Indeed, the relation can be stored as an array, with an implicit index field.

Similarly, a sparse linear array is a relation in which there are rows with index field i and value field $A[i]$ for only non-zero values. Multi-dimensional arrays are relations, in which the keys are tuples of all the indices. Sets of points, particles, vertices, or employees, can be represented as relations.

Relations can be implemented in a variety of ways. If the *key* of a relation is a tuple of integers or characters, then the relation can be stored as a multi-dimensional array, with one dimension for each field of the key. This array can be very sparse, in which case a hashing or linked-list implementation is more efficient. A user can suggest an implementation scheme to the compiler by providing a program annotation.

Array Notation for Relations Conventionally, we refer to element i of an array A as $A[i]$. We would like to use the same notation if array A is sparse, and is stored as a relation with two columns: index i and nonzero value $A[i]$, where the index is the key. In this case, $A[i]$ is the non-key data of the relation A in the row with key i. This suggests that, given a relation employee with key social_security_number, we should refer to an employee with social security number 111111111 as employee[111111111]. This is merely a notational convenience, and does not imply that the implementation of the employee relation requires 1,000,000,000 locations. In general, the row with key k in a relation R is denoted by $(k, R[k])$ where $R[k]$ is the non-key data.

Sparse Arrays A one-dimensional sparse matrix A with non-zero values for A[2] = 3 and A[150]= 5 and zeros elsewhere can be stored as a table with rows (2,3) and (150,5). Now, if we lookup the table for key 6, and don't find it, we conclude that A[6] = 0. To use array notation for an arbitrary relation we have to define the meaning of $R[k]$ if there is no row in relation R with key k.

For each relation R we define a zero element, and the value of $R[k]$ is the zero element of the relation if there is no entry in the relation with key k. So, if we have no employee with social security number m then employee[m] = zero_employee, where zero_employee is the zero value defined for the relation employee. The introduction of zero elements for relations allows us to treat arrays (particularly sparse arrays) and relations using a uniform notation. If a zero element is not specified, a default zero element is assumed: for example, as the default zero element, numbers (integers and floats) have the value 0, characters have the value 'a', and strings have the empty string("").

Relations can also be manipulated using standard operators from relational algebra like selection, union, difference and intersection. Some of these operators are described in the next section. The reason for introducing array notation is to keep the relational notation close to the notation used in scientific applications by programmers today.

1.2 Reduction

Quantification Quantification is used extensively in mathematics. For instance,

$$(\forall i \in I : 0 \leq i < 3 : c[i] = 0)$$

means that for all i in set I such that $0 \leq i < 3$, the equation $c[i] = 0$ holds. The notation for quantification is extended to get reductions with associative, commutative operations such as sum $(+)$ and product $(*)$; for example:

$$(+\ i \in I : 0 \leq i < 3 : e[i]) = e[0] + e[1] + e[2]$$

and

$$(*\ i \in I : 0 \leq i < 3 : e[i]) = e[0] * e[1] * e[2]$$

For every operator op, we introduce an identity element **unity** such that, for all w,

$$w\ op\ \textbf{unity}\ =\ w$$

Parallel Composition The meaning of s **par** t, where s and t are statements, is execute s and t in parallel. The precise meaning of parallel composition is not important at this point, except that the parallel composition operator is associative and commutative.

The concept of reduction is extended to control flow operators such as the parallel composition operator **par**, in the obvious way.

$$(\textbf{par}\ i \in I : 0 \leq i < 3 : f(i)) = f(0)\ \textbf{par}\ f(1)\ \textbf{par}\ f(2)$$

where $f(i)$ is a statement with parameter i.

For an arbitrary boolean expression $b(i)$, the meaning of

$$(\textbf{par}\ i \in I : b(i) : f(i))$$

is execute statements $f(i)$ in parallel, for all $i \in I$ for which $b(i)$ holds.

Sequential Composition The meaning of s **seq** t, where s and t are statements and **seq** is the sequential composition operator, is the program fragment $s;t$ in C-syntax, i.e. s is executed first, and after s terminates successfully (i.e., without aborting) t is executed. If s does not terminate, or s aborts, then t is not executed.

For an arbitrary boolean expression $b(i)$, and an ordered set I, the meaning of

$$(\textbf{seq } i \in I : b(i) : f(i))$$

is given by the program fragment:

$$\textbf{for } (i \in I) \textbf{ if } (b(i))\ f(i);$$

where the **for**-loop steps over all elements of the ordered set I starting with the smallest and ending with the largest: for each i, if $b(i)$ holds then $f(i)$ is executed, otherwise that value of i is skipped.

For example, if I is the ordered set of integers $0..N$, A is an array of integers indexed $0..N$, and sum is an integer, then:

$$\text{sum} = 0;\ (\textbf{seq } i \in I : \text{A}[i]{>}0 : \text{sum} = \text{sum} + \text{A}[i]);$$

puts the sum of all positive elements of A in sum.

Dummy Variables Quantifications in mathematics distinguish the set over which the quantification is being taken from the dummy variable of the quantification; for example

$$(+\ i \in I : 0 \leq i < 3 : c[i])$$

distinguishes the set I from the dummy variable i. For notational convenience, we associate dummy variables with a set, for instance we associate dummy variable i with set I, and we shorten the notation for a reduction $(op\ i \in I : b(i) : e(i))$ to $(op\ I : b(i) : e(i))$ where the dummy $i \in I$ is implicit.

In most of our examples, we use upper-case names for relations and sets, and the corresponding lower-case for the dummy variables used to quantify over sets and relations. It is possible to associate more than one dummy variable with a relation.

1.3 Data-Parallelism

In this section we give an overview of a language fragment that describes constructs for manipulating relations and reductions. For specificity, we describe the fragment using C-like syntax.

Declaring a Relation A relation is declared in a manner similar to a record (or struct). The declaration is of the form:

relation *name* { *non-keys* } *dummy* [*keys* : *max_size*];

The declaration consists of a *name* for the relation, an enumeration *non-keys* of the name and type of each of its non-key fields, the name *dummy* of the dummy variable associated with the relation, an enumeration *keys* of the name and type of its key fields, and *max_size*

the maximum size of the relation. (In future implementations, the max_size field need not be specified.)

A relation, PIXELS, of at most N elements, with two non-key fields called *color* and *intensity*, and two key fields x and y, and a maximum size of N where *pixels* is the dummy associated with the relation PIXELS, is declared as follows:

relation PIXELS { **int** color, intensity; } pixels[int x, y:N];

This relation can be used to store a set of pixels where x and y are the coordinates of the pixel. The pixel at location (0,1) is denoted by pixels[0,1]; if there is no entry for key (0,1) in the relation then pixels[0,1] is the zero element of the relation.

Alternately, the key can be specified as a struct which is a tuple of names:

struct position { **int** x, y; };

and the relation declared as:

relation PIXELS { **int** color, intensity; } pixels[position :N];

In this case, if q is declared and initialized as follows:

struct position q = {0, 1};

then pixels[q] is the pixel at location q.

Index-sets A relation that does not contain any non-key fields is referred to as an *index-set*. Thus an index-set is a set of tuples, where the elements of a tuple are ordered, and the tuples themselves are ordered lexicographically. The following declares index-set P with at most 4 elements, and with dummy variable p associated with it, where the set is initially {(0,0), (0,1), (1,0), (1,1)}.

index_set P:p[4] of position = {(0,0), (0,1), (1,0), (1,1)};

P refers to the name of the index-set and p refers to an arbitrary member of the set. In general, the declaration of an index-set must specify a name for the set, an identifier that may be used to range over elements in the set, and the size of the set. Optional attributes include the type of elements in the set (if omitted, this is assumed to be integer) and the initial values of elements in the set (if the elements are listed, the size may be omitted). The following statement declares an index-set I which consists of N integers from 0 to N-1.

index_set I:i = {0..N-1};

An index-set (or its subset) may also be used as a key to reference elements of a relation in a reduction. For instance, the following fragment counts the number of elements in relation *pixels* whose *intensity* is greater than some threshold d:

(+ p ∈ P : (pixels[p].intensity > d) : 1)

Reductions Our syntax for reductions is a compromise between the syntax for quantifications in mathematics and C-syntax. A reduction:

$$(op\ i \in I : b(i) : c(i))$$

is written as

$$\$op\ (I\ \textbf{st}\ (b(i))\ c(i))$$

Here **st** stands for such that. Dummy variable i is declared to be associated with relation I. For example, the sum of i * A[i] where the sum is taken over the positive elements of an array A is

```
$+ (I st (A[i] > 0) i * A[i])
```

If the **st** clause is *true*, it can be removed and a semicolon substituted as a separator:

```
$+ (I st (true) i * A[i])
```

can be written as

```
$+ (I; i * A[i])
```

Selection on relations or index-sets can be implemented using the **select** operator in a reduction. The following fragment selects even elements from index-set I and assigns them to index-set J.

```
J = $select (I st (i%2 == 0) i)
```

A slightly different syntax is used for the control-flow operators **par** and **seq**.

$$(\textbf{par}\ i \in I : b(i) : s(i))$$

where *b(i)* is a boolean expression and *s(i)* is a statement is written in our notation as:

$$\textbf{par}\ (I)\ \textbf{st}\ (b(i))\ s(i)$$

For instance, decrement all positive elements of array A by one is:

```
par (I) st (A[i] > 0) A[i] = A[i]-1;
```

The following fragment sets all elements of array A to A[0].

```
seq (I) st (i > 0) A[i] = A[i-1];
```

Example We complete the overview with a simple example that is a data parallel implementation of the sieve of Eratosthenes used to identify the prime numbers in a sequence of integers. Given a sequence of integers starting with 2, the algorithm is as follows: compute the smallest integer in the sequence, say *nextp*, and mark it as a prime. Subsequently remove all multiples of *nextp* from the sequence. This is repeated until the sequence is empty. The program is shown in figure 1. The reduction (line 9) computes the smallest element in the sequence. If the sequence is empty, the reduction returns *MAXINT* (the **unity** value for the minimum operator) and the loop terminates. The loop body marks nextp as a prime number (line 7) and removes its multiples from the index-set (line 8).

```
1   main (void) {
2       int prime[N]; /* initialized to 0, prime[i] is set to 1 if i is prime */
3       index_set I:i = {2..N-1}; /* assume N > 2 */
4       int nextp = 2;
5
6       while (nextp < MAXINT) {
7           prime[nextp] = 1;
8           I = I - $select(I st (i % nextp == 0) i);
9           nextp = $<(I; i)
10      }
11  }
```

Figure 1: Sieve of Eratosthenes

1.4 Paper Overview

Our hypothesis is that the combination of relations and reduction is sufficient for a data-parallel language for scientific and commercial applications. The language can be based on existing relational database languages, with the addition of array notation and reduction. To demonstrate proof of concept, we have implemented a data-parallel language called UC that integrates relations and reduction: UC executes on the CM-2[Hil85] and on single workstations. Since our goal was only proof of concept, we implemented a simple extension of C, rather than a relational language such as SQL with array notation and reduction operators. This extension is described next.

The extension is not a complete implementation of relational calculus; we only implement sets of integers in place of relations. Our plan is to evaluate the idea in terms of an implementation of the simple extension to C, and then implement the idea using existing relational languages if experiments with the initial implementation are positive. The experiments conducted so far are positive both in expressivity and in efficiency.

The rest of the paper is organized as follows: section 2 describes the UC language together with its mapping facilities. Section 3 presents performance results from existing UC implementations and compares its performance with other langauges such as CM Fortran, *Lisp and C*. Section 4 describes a number of UC examples. Section 5 discusses related work. Section 6 is a brief description of the UC implementation and section 7 is the conclusion.

2 UC Constructs

UC is a data-parallel language that adds the following features to C: a data-type called *index-set*, dynamic array bounds, *reductions*, and four composition operators: **par**, **seq**, ***solve** and **arb**. The primary restrictions that it imposes on C is in the use of pointers and explicit control flow statements (goto, break, etc.). This section is a brief description of the preceding constructs. The reader is referred to [BA92] for a detailed description.

```
index_set I:i = {0..9}, J:j = I;
int min, first_x, arb_max, a[N];
float avg;

avg = $+(I; i) / $+(I; 1.0);
min = $<(J; a[j]);
first_x = $<(I st (a[i] == x) i);
arb_max = $,(I st (a[i] == $>(J; a[j])) i);
```

Figure 2: Reduction in UC

2.1 Index Sets

The current implementation restricts index-sets to ordered set of integers. The following illustrates commonly used forms for their declaration.

```
index_set I:i = {0..N-1}, J:j = I, idx2:k = {4,3,9};
```

Index-set I is an example of a *range* index-set and consists of N integers from 0 to N-1. Idx2 contains the 3 elements listed explicitly and is an example of an *enumerated* index-set. A number of operations including *union*, *intersection* and *difference* are supported on index-sets. Each of these operations is implemented by an appropriate library routine. Once an index-set has been initialized, the number of elements in the set cannot be increased. However the values in the set can be modified either by using the preceding operations or by using the reductions defined in the next section. The bounds of an index-set are not restricted to compile-time constants. If they are specified by variables, its size is determined by the value of the corresponding variable when the index-set is initialized.

In the remainder of this paper, we use upper-case identifiers to refer to an index-set and the corresponding lower-case identifier to refer to an arbitrary element of the index-set. Unless otherwise noted, all index-sets are assumed to range from 0 to N-1 and all array axes are assumed to be of size N.

2.2 Reduction

A reduction operator performs an associative, binary operation on a set of expressions, and returns the resulting value. The expression being reduced may be guarded, and more than one guarded expression may be used. The operators that may be used in a reduction include **$&&** for logical-and, **$||** for logical-or, **$>** for maximum, **$<** for minimum, **$*** for multiplication, **$+** for addition, and **$,** to return the value of an arbitrary operand.

The code fragment in figure 2 contains several simple examples of reductions. In particular, first_x computes the index of the leftmost occurrence of x and arb_max uses the **$,** operator to return the position of any occurrence of the maximum value. Notice that arb_max uses a nested reduction, where the inner reduction ($>(J; a[j])) computes the maximum value in the array.

It is sometimes useful to specify reductions that return a *sequence* of values rather than a single value; such operations are referred to as tuple reductions in UC. As a simple example, consider a computation that requires both the maximum value as well as its

position in a vector. Such operations may be specified concisely using tuple reductions. Tuple reductions differ from simple reductions in that the operands in the former may be a sequence of expressions separated by a semicolon; in addition, a separate operator may be used to reduce each expression. However, the set of enabled index-elements is the same for all expressions in the tuple: if a tuple reduction has multiple clauses, the tuple operands in all clasues must have the same arity. The following example illustrates the use of tuple reductions in computing the greatest element together with the position of its leftmost occurence for a vector a.

```
(max;pos) =$>$<(I st (a[i]==$<(I; a[i])) (a[i];i));
```

2.3 Parallel Assignment

Parallel assignments are the most common UC constructs. A parallel assignment is executed *synchronously*: all instances evaluate the same expressions at the same time. An *instance* of a program fragment is one where every occurrence of a dummy variable is replaced by an element of the corresponding index-set. The following assignment transposes array c by synchronously executing N^2 instances of the assignment statement:

```
par (I,J)
  c[i][j] = c[j][i];
```

The body of a **par** statement may be a compound statement that includes variable declarations. In this case, each statement is evaluated synchronously. For instance, the following fragment swaps the contents of arrays a and b.

```
par (I) {
    int tmp[N];
    tmp[i] = b[i];
    b[i] = a[i];
    a[i] = tmp[i];
}
```

A par statement may use a predicate (called the *guard*) to select a subset of *enabled* elements from the index-set. An index element is enabled if on substituting its value in the predicate, the predicate evaluates to *true*. The par statement is executed in parallel for all enabled elements. The guard together with the statement to be executed is called a ***guarded command*** or a *clause*. A par statement may contain multiple clauses, which are executed asynchronously. For instance, the following fragment sets the even elements of b to 0 and the odd elements to 1.

```
par (I)
  st (i%2 == 0) b[i] = 0;
  st (i%2 != 0) b[i] = 1;
```

As the multiple clauses of a par statement are executed asynchronously, they are a potential source of race-conditions: if a variable is modified in two differnet clauses, its final value is

no longer deterministic. To guarantee a deterministic execution model, the multiple clauses are required to be *non-interfering*. Two clauses are said to interfere if a variable modified by one clause is referenced by the other clause. Note that each array element is considered a distinct variable.

A par statement may contain nested **if** statements. The semantics of such a statement is identical to that of a par statement with multiple clauses, where each clause is equivalent to an arm of the corresponding if statement. The following is an example of an if statement nested within a par statement and the corresponding par statement:

```
par (I) {                    par (I) {
  if (b(i))                    st (b(i))
    S1;                          S1;
  else                         st (!b(i))
    S2;                          S2;
}                            }
```

Note that statements *S1* and *S2* are required to be non-interfering.

A par statement may also include C iterative constructs. For instance, consider a par statement with a nested **while** loop.

```
par (I)                      while ($||(I; b[i]))
  while (b[i])                 par (I) st (b[i])
    S1;                          S1;
```

For every iteration, the loop condition is evaluated for all elements in I. The loop is terminated if the condition evaluates to false for every element; otherwise statement *S1* is executed synchronously for every enabled element. The operational semantics of the par statement is specified by the fragment on the right-hand side. The semantics of other iterative constructs, including the **do-while** and **for**, are defined similarly.

2.4 Sequential Execution

The **seq** statement can be used for repeated sequential execution of a nested statement, once for each element of an index-set, in the order determined by the definition of the index-set. The syntax of a seq statement is identical to that of a par statement except for the obvious substitution of keyword **seq** for keyword **par**. As a simple example, consider the following statement which prints out the pairs (0, 9), (0, -2), (1, 9), (1, -2) in that order:

```
index_set I:i = {0..1}, J:j = {9, -2};
seq (I, J)
    printf("(%d, %d) ", i, j);
```

As another example, the following fragment computes the all pairs shortest path using a simple algorithm described in [CM88]. For a given k in 0..N-1, the parallel assignment computes the shortest path between every pair of nodes such that no intermediate node is labeled greater than k. It follows that when k=N-1, dist[i][j] will contain the shortest path in the graph from node i to node j. The seq statement ensures that the parallel assignment is executed N times for consecutive values of k in 0..N-1. In the program, we omit the initialization and definition of the macro MIN which returns the smaller of its two arguments.

```
#define N 32
index_set I:i = {0..N-1}, J:j = I, K:k = I;
int dist[N][N]; /* initialized appropriately */
seq (K)
  par (I,J)
    dist[i][j] = MIN(dist[i][k]+dist[k][j], dist[i][j]);
```

2.5 Asynchronous Execution

Parallel assignments require that the specified statements be evaluated synchronously for all enabled index elements. We define another composition operator called **arb** for arbitrary order which allows the statement instances to be executed using an arbitrary interleaving. Thus unlike the **par** statement, given a fragment **arb** (I) $\{s_i; s_2;\}$, an instance of s_2 could begin execution *before* all instances of s_1 have been completed. Asynchronous execution is particularly useful when compiling UC programs for execution on asynchronous architectures. The asynchronous execution requires that the enabled instance of an arb statement be non-interfering as defined in the previous section. Note that an arb statement may always be written as a par statement, but not vice-versa. The arb statement may be viewed as a refinement of an equivalent UC program with par statements, where the former introduces greater asynchrony in its execution.

2.6 Fixed-point Computation

The solution to a number of scientific and graph problems may be expressed succinctly by using fixed-point semantics. UC provides a construct called ***solve** that may be used for this purpose. A *solve statement executes iteratively until its computation reaches a fixed-point, i.e. further execution of its body will not alter the value of any variable modified by the statement. We illustrate the use of *solve to solve the graph reachability problem: given a directed graph G, it is required to compute r, the all-points-reachability matrix for the graph. Assume that the edge connectivity of the graph is represented by matrix e, where $e[i][j]$=1 if G contains an edge from i to j, and is 0 otherwise. We develop a UC program that sets $r[i][j]$ to 1, if j is reachable from i, and 0 otherwise. A node j is said to be reachable from i if

- an edge exists from i to j in G or
- some k is reachable from i and an edge exists from k to j in G.

The preceding specification can be directly implemented as the UC program of figure 3. Lines 1-2 of the program declare arrays e and r and the appropriate index-sets. The *init* function in line 6 initializes the arrays such that the adjacency information is read into the array e and the array r is initially set equal to e. The *solve statement in line 6 contains a single assignment (line 7) that simply codes the definition of reachability between a pair of vertices in UC syntax. The reduction in line 7 determines if node j can be reached from node i based on the preceding definition of reachability and the current set of reachable pairs that have been computed. The enclosing *solve construct specifies that the assignment statement should be executed in parallel for all elements in the index-sets I and J; additionally, it also specifies that the statements be executed repeatedly until they reach a fixed-point. In

```
    /* a prototype program */
1   index_set I:i = {0..N-1}, J:j=I, K:k=I;
2   int e[N][N], r[N][N];
3
4   main(void) {
5       init();
6       *solve (I,J)
7           r[i][j] = $||(K; r[i][k] && e[k][j]);
8   }
```

Figure 3: Graph Reachability using *solve

```
float a[N][N], epsilon;
index_set I:i = {1..N-2}, J:j = I;
index_set K:k = {-1,1};
*solve (I, J) to epsilon
    a[i][j]= $+(K; (a[i+k][j] + a[i][j+k])) / 4.0;
```

Figure 4: Iterative Convergence for Real Numbers

other words, the *solve statement terminates only when further execution of the statements enclosed within its scope will not modify the value of any variable referenced within the statement.

For programs that use floating-point arithmetic, the computation may not necessarily reach a fixed-point; instead the computation may converge such that successive execution of the program perturbs the value of each data item within an arbitrarily small interval. The *solve construct may optionally specify this convergence interval, as demonstrated by the example in figure 4. This program computes the value of the interior points in a grid as the average of the values of its North, South, East and West neighbors. The computation terminates when the successive value of every data-item is within some *epsilon* of its previous value. The *epsilon* is referred to as the convergence interval and may be specified in a program in the optional to clause of the *solve statement. If omitted, as it was in the shortest path example, the convergence interval is assumed to be zero, in which case execution terminates only when the computation reaches a fixed-point.

2.7 Parallel Functions

Functions in a UC program may be invoked in parallel; however the different instances of the call are executed *asynchronously*. Thus functions provide a simple mechanism for UC programs to weaken the expression level synchronization assumed for parallel assignments. Asynchronous execution of the function calls also implies that there is a similar possibility for non-determinism as in the execution of par statements with multiple clauses. As before, UC avoids non-determinism by declaring parallel function calls *semantically ill-formed* if

the instances of the function call *interfere* with each other. A parallel function can cause interference only if it modifies shared variables. Sharing of variables by parallel instances may occur either by sharing global variables, sharing static variables or by passing pointers or array parameters. The compiler uses a conservative check and issues an appropriate warning if a function uses any of the preceding forms of sharing.

2.8 Data Mappings

A UC program need not specify how each program data structure is to be mapped on the memory hierarchy of the parallel architecture. The compiler uses built-in heuristics to distribute the data on the architecture and *dynamically* change it, if necessary. However, there are a variety of complex data mappings that may significantly improve program efficiency, but may not be easy to extract from a local analysis of the program. UC provides a general mechanism that may be used to describe different forms of data distributions[BM91]. This facility may be used by the programmer to explicitly specify many types of data mappings. The following types of data mappings are currently supported: **permute**, **fold** and **copy**. A **permute** mapping is used to reorder the elements of an array so that corresponding elements of different arrays that are accessed in a single assignment can be stored locally; **fold** allows part of an array to be folded over so that corresponding elements of the *same* array that are accessed together can be stored locally. A **copy** allows data to be replicated. In particular, it may be used to replicate an array along an extra dimension to reduce the need for broadcasts.

Mappings may be used for a variety of purposes in a UC program:

- Separation of program logic from data distribution and hence improved code portability.
- Reducing remote references and hence improving program performance.
- Support for (dynamic) data partitioning and hence for load-balancing.

Remote references may be reduced by changing the layout of the array. Alternately, the array elements may be replicated to make them available at all locations of use. The former can be achieved with the permute mapping, while the copy mapping is useful for the latter.

Mapping declarations may also serve as a tool to partition data. The fold mapping can be used to map several data element to each processor. This mapping may also be used to dynamically change how the data is partitioned.

As map declarations exist separately from the program body, they can be used to tune the program to different architectures. Ideally, the user may write the program independently of the architecture and map declarations may then be used to provide architecture-specific information such as partitions, alignments, etc.

Each map declaration removes or adds a number of expensive operations to the program. The cost of such operations may be available as published timings or they can be determined by running an appropriate test-bed. This knowledge can then be used to estimate the effect of the mapping on performance. A performance estimation facility is being developed to facilitate a compile-time evaluation of the impact of different mappings on the performance of a UC program.

A mapping specification has five parts: the *type of mapping* which may be one of *permute*, *fold* or *copy*, the *source* which refers to the data structure that is being mapped, the *target* which is the data structure that the source is modified with respect to, a *predicate* which may restrict the mapping to a subset of the source and a mapping function which describes how the source array is to be transformed. The UC data mappings may be relative or absolute; the former use a program data structure as the target while the latter use a virtual data structure. A virtual data structure is typeless and does not occupy physical memory; its only purpose is to define a template that is used to describe a subsequent mapping. A mapping is syntactically similar to the par and seq statements of UC.

For **relative** mappings, the layout of the source array is modified relative to the current layout of the target array. The target array may itself have been mapped previously. In this case, the source array is mapped relative to the mapped target array. If the source and the target are the same, the array is being mapped relative to its own previous layout.

Mappings may be **dynamic**. In general, a mapping statement may occur at any place in a program where a declaration would be legal; however the source and target data structures must have been declared prior to the map statement. Dynamic mappings imply that a given array may be re-mapped several times in deeper nested scopes. This raises the possibility that several, variously mapped versions of an array may exist in the program. For this reason, the compiler needs to generate operations at the beginning and end of new scopes to copy the data to and from newer versions of the array respectively.

We now describe each of the three types of mappings supported by UC and provide experimental measurements that indicate the performance improvements that have been realized for a variety of programs.

Permute

Permute is a one-to-one mapping. Each element of the source array is mapped to one location in the target array. The compiler ensures that the target array is at least as large as the source array.

Arbitrary functions may be used to reference the target array in the map specification. However, experience suggests that a mapping specification is most effective when the subscripts in the transformed array can be simplified sufficiently to distinguish local and remote references. This analysis facilitates optimal code generation for the mapped data structures. For subscript expressions to be evaluated at compile-time, identifiers in the expression must be restricted to constants or index-elements, and operators to +, −, and *mod*. If the preceding restrictons are not observed, the compiler must generate code that *potentially* refers to remote data; efficiency of the generated code then depends on the sophistication of the communication library in detecting and optimizing for local communications. Under certain conditions, it is possible to generate code that uses run-time analysis to decide whether data is local or remote and executes appropriate code segments.

As a simple example of a permute mapping, consider the following code fragment:

```
int a[N][M], b[N][M];
perm (I,J)
    a[i][j] :- b[(i+1)%N][j];
par (I,J) st (i > 0)
    b[i][j] = a[i-1][j];
```

```
{
    int c[M][N];
    perm (I,J)
        a[i][j] :- a[j][i];
    par (I,J) st (i > 0)
        a[i][j] += c[j][i-1];
}
```

The arrays a and b initially have identical layouts (perhaps one element per virtual processor). The first map declaration changes the layout of a so that the $(i,j)^{th}$ element of a is aligned with the $(i+1,j)^{th}$ element of b (note that this is a circular shift). In the next deeper scope, a is re-mapped (a transpose) with respect to its previous layout. As a result, the $(i,j)^{th}$ element of a is now aligned with the $(j,i+1)^{th}$ element of c.

We complete the discussion of permute mappings by presenting measurements on the effectiveness of these mappings on the CM-2 (16K processors) for a variety of simple examples.

operation	permute mapping	unmapped (millisecs)	mapped (millisecs)
a[i][j] += d[j][i]	d[j][i] :- a[i][j]	1.793	0.217
a[4][j] = c[j]	c[j] :- a[4][j]	0.831	0.203
mat[i][i] = dia[i]	dia[i] :- mat[i][i]	0.843	0.322
a[i][j] = b[i-1][j]	b[i][j] :- a[(i+1)%N][j]	0.331	0.205

Copy

Copy is a one-to-many mapping that may be used to replicate parts of a data structure. The compiler is responsible for ensuring that the multiple copies of a data item are *coherent*. The most common use of this mapping is to replicate an array along one or more additional axes. Due to the overhead of maintaining coherence, efficient implementations for this mapping require that data items that are being replicated be identified at compile-time. If not, the compiler may need to generate code that must ensure that all data elements are kept coherent, even though only a subset of the elements are replicated. Once again, this condition may be enforced by requiring that the operands in the predicate and the expressions in the source and target for the mapping be restricted to constants and index-elements.

The utility of the copy mapping in improving execution efficiency of a program is illustrated in the context of a UC program that models mosquito populations. The example and the mappings together with measurements that indicate the performance improvement are presented in section 4.

Fold

Fold is a many-to-one mapping. It is primarily a tool to partition arrays when the number of physical processors available is less than the number of data elements. It can also be thought of as a way to control how virtual processors are simulated on each physical processor. A fold mapping requires that source and the target array have the same number of dimensions.

Once again, optimal code generation requires that the subscript expressions be known at compile-time. This may be enforced by requiring that all operands be constants or index-elements and that operators used to subscript the target array be restricted to *mod* or *div* which may be used to respectively specify a **cyclic** or **block** partition.

As examples of simple decompositions, consider the following:

```
/* cyclic partitioning */
int a[N];
virtual M[N/F];
fold (I)
    a[i] :- M[i%f];
```

```
/* block partitioning */
int a[N];
virtual M[F];
fold (I)
    a[i] :- M[i/F];
```

In contrast with standard block and cyclic decompositions supported by most languages, a UC fold mapping can specify a variety of partitions, whose size and distributions can be changed dynamically. For instance, consider the following fold mapping:

```
int f (int a[], K, F1, F2)
{
  fold (I)  st (i<K)
      a[i] :- a[i/F1]
            st (i>=K)
      a[i] :- a[K/F1 + (i-K)/F2]
  :
}
```

The preceding example allows parts of array a to be blocked in different sizes. For simplicity, assume that K is a multiple of F. The mapping specifies that that the first K elements are decomposed into blocks of F1 elements and the rest into blocks of F2 elements. As F1, F2 and K are function parameters, their values may change with different invocations of the function allowing the blocking size to be modified dynamically, perhaps in response to the relative density of computation in different parts of the array over different phases of execution.

The mosquito application is also used to demonstrate the use of a copy mapping in improving execution efficiency of a program. The example and the mappings together with measurements that indicate the performance improvement are described in section 4.

3 Performance

In this section, we summarize the comparative performance of a number of applications when written in UC, C*, and CM Fortran. Some of the examples are described in greater detail in the next section and where feasible a complete listing of the UC program has been included. For brevity, complete listings of the programs in the three languages have been included for only one application – graph reachability in Appendix A. Figure 5 contains the comparative timings for some small graph algorithms and numerical applications. All runs were done on a 16K partition of a CM-2 and the reported times refer to the elapsed wall-clock time to execute the complete program. As seen from the timings, the performance of the UC program is comparable with both the C* and CM Fortran versions of the program.

Program	CMF	C*	UC
Reachability	3.2	3.0	3.4
Matmul	9.8	10.4	9.5
Primes	3.0	3.1	4.2
Spath	2.0	2.0	2.3
Diffusion	12.3	*N/A*	11.5
Gauss-Jordan	12.8	12.6	16.1

Figure 5: Performance comparisons (time given in seconds)

4 Examples

The current UC implementations have been used to design applications in a variety of areas that include Computational Fluid Dynamics, Partial Differential Equation Solvers, Computational Biology, Image Processing and Graph Algorithms. The compilers have been used by researchers to develop more efficient programs for specific applications as also by students interested in learning parallel programming. Student programs have been developed for graph algorithms (minimum spanning tree, shortest path, maximum flow, traveling salesman problem etc.), numerical applications (Fast Fourier Transforms, Gaussian elimination, parameter estimation, etc.), search applications (8 queens problem, optimal strategy for playing Othello, etc.), and miscellanous applications that include DNA-matching and unstructured grid computations.

In this section, we describe a variety of examples across different application spectrums that have been developed in UC; measurements are reported where meaningful. Unless stated otherwise, all measurements reported here were taken on a CM-2 with 16K processors with floating-point accelerators, and a SUN front-end.

4.1 Graph Reachability

As described in section 2.6, the UC program for computing all nodes that are reachable from a given node is straightforward and may be written as follows:

```
*solve (I)
    r[i] = $||(J; r[j] && e[i][j]);
```

where r contains the vertices reachable from vertex zero and e is the incidence matrix of the graph.

The complete programs that implement this algorithm in UC, CM Fortran and C* are given in appendix A. The listings include the input-output statements as well as the instructions to measure the performance of the programs. As seen from the listings, the UC program is the shortest, with Fortran coming in a close second and C* a distant third. The relative timings for the three applications were given in the previous section. Figure 6 includes the timings for a slightly different configuration and includes both the CM time and total time. The CM time refers to the amount of time during which the CM actually executed instructions, and total time measures the elapsed wall-clock time; any difference

Language	CM time (secs)	Total time (secs)
CMF	13.4	15.5
C*	12.9	14.8
UC	14.3	16.2

Figure 6: Performance of reachability

between them is time that the CM spent waiting for instructions from the front-end. All programs were executed on a 8K partition of a CM-2; the input was a graph with 512 vertices that required 511 iterations of the loop. As seen from the timings, the performance of the UC program is always within 10-15% of the others.

4.2 Matrix Multiplication

If $A = (a_{ij})$ is an $m \times n$ matrix and $B = (b_{jk})$ is an $n \times p$ matrix, then the matrix product AB is defined to be the matrix $C = (c_{ik})$ with the elements

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

Assuming appropriate initializations, a straightforward program that uses $O(N^2)$ processors is as follows:

```
par (I,J)
    seq (K)
        c[i][j] += a[i][k] * b[k][j];
```

However, each multiplication involves expensive remote accesses. An efficient variation on the preceding algorithm (that also uses $O(N^2)$ processors) for the CM-2 is described in [Tic89]. This algorithm uses commutativity of addition to modify the standard systolic algorithm for matrix multiplication. The algorithm periodically permutes the operand arrays (using efficient global communication primitives) before every multiplication such that the multiplicands are available locally. In the UC program given in this section, variables tmp1 and tmp2 are used to store the permuted operand arrays. Relative performance for the UC, C* and CM Fortran programs were presented in the previous section. The definitions of input/output routines have been omitted for brevity.

```
#define N 8
int a[N][N], b[N][N];
int acc[N][N], tmp1[N][N], tmp2[N][N];
index_set I:i = {0..N-1}, J:j=I, K:k=I;

int main (void) {
    par (I,J) {
      a[i][j] = i;
      b[i][j] = j;
    }
```

```
    par (I,J) {
      /* Skew the input matrices. */
      tmp1[i][j] = a[i][(i+j)%N];
      tmp2[i][j] = b[(i+j)%N][j];

      acc[i][j] = 0;
      /* N times, add in the product of this pair, then shift */
      /* each matrix over one in the appropriate direction. */
      seq (K) {
        acc[i][j] += tmp1[i][j] * tmp2[i][j];
        tmp1[i][j] = tmp1[i][(j+1)%N];
        tmp2[i][j] = tmp2[(i+1)%N][j];
} } }
```

4.3 Stable Marriage

This problem as described in [And91] is as follows: let *Man* and *Woman* each be arrays of n processes. Each man ranks the women from 1 to n, and each woman ranks the men form 1 to n. A *pairing* is a one-to-one correspondence of men and women. A pairing is *stable* if, for two men m_1 and m_2 and their paired women w_1 and w_2, both of the following conditions are satisfied:

- m_1 ranks w_1 higher than w_2 or w_2 ranks m_2 higher than m_1; and
- m_2 ranks w_2 higher than w_1 or w_1 ranks m_1 higher than m_2.

Put differently, a pairing is unstable if a man and woman would both prefer each other to their current pair. A solution to the stable marriage problem is a set of n pairings, all of which are stable. The program in Figure 7 is the UC code to solve the stable marriage problem. Once again, input/output and initialization functions have been omitted for brevity. The program generates a stable pairing as follows[1]: in a given iteration, each woman tries to find the lowest ranked mate (note that a lower rank implies a higher preference) such that the pairing is stable. A stable pairing is discovered when all men have been assigned to a woman.

4.4 Simulation of Mosquito Control

A biological model of the life of *Culex pipiens* mosquitoes[FTD89] was programmed in *Lisp. This model was executed on the CM-2 to study the growth of mosquito populations in Orange County[JT92]. The model studies mosquito development during the four primary stages: *egg*, *larva*, *pupa*, and *adult*. Each life stage can have three different types of organisms (genotype *aa*, *Aa*, and *AA*). The area comprising Orange County was subdivided into a grid of *locations*. Each location is associated with a varible number of breeding *sites*, each of which is identified as a gutter, pool, drain, channel, or container. Each site may have

[1] In general, many stable pairings are possible.

```
#define UNASSIGNED -1

index_set I:i={0..N-1};     /* used for the women */
index_set J:j={0..N-1};     /* used for the men   */

int rank[N][N],      /* rank[i][j] : rank assigned by w_i to m_j */
    prefer[N][N];    /* prefer[j][-] : preference list for m_j   */
int unassigned[N],   /* unassigned[j]=1 ==> m_j is unassigned    */
    next[N],         /* next[j] : current top preference for m_j */
    fiancee[N];      /* fiancee[i] : current mate for w_i        */

int main(void)
{
 int best_rank[N],    /* rank of best man suitor for w_i */
     new_fiancee[N];  /* new man selected as mate by w_i */

 init();              /* initializations of rank and prefer       */
                      /* all w_i and m_j are initially available */

 while ($+(I; unassigned[i]))  /* while at least one man is unassigned */
   {
     par (I)  /* try to assign each w_i a 'stable' mate */
     {
       /* Each w_i chooses the suitor with lowest (hence best) rank.  */
       best_rank[i]                                    /* for all m_j, such that */
        = $<(J st((unassigned[j] == 1) &&              /* m_j is available and */
                  (prefer[j][next[j]] == i) &&         /* w_i is most preferred by m_j */
                  ((fiancee[i] == UNASSIGNED) ||       /* and w_i is unassigned or */
                   (rank[i][fiancee[i]] > rank[i][j]))) /* prefers m_j to current mate */
                 rank[i][j]);                          /* choose the m_j with lowest rank */

        if (best_rank[i] < N)                          /* if w_i found a suitor */
        {
            new_fiancee[i]                             /* extract new suitor for w_i */
                = $,(J st (best_rank[i] == rank[i][j]) j);

           if (fiancee[i] != UNASSIGNED)
              unassigned[fiancee[i]] = 1;    /* unassign old husband and */
           fiancee[i] = new_fiancee[i];      /* and assign a new one     */
           unassigned[fiancee[i]] = 0;
        }
     }

     par (J) st (unassigned[j] == 1)          /* for each unassigned man          */
       next[j] = next[j]+1;                   /* update index into preference list */
   }
 print_fiancee();
}
```

Figure 7: Stable Marriage in UC

populations of all 12 different types, with each population containing organisms at different *tiers* of development.

The simulation is initialized by assigning a certain number of sites to each location and identifying one of four weather stations with each location. The atmospheric conditions (air temperature, precipitation, etc.) at a location are determined from data read from the corresponding weather station. Finally, the egg, larva, pupa, and adult female populations are initialized at each site. The simulation is subsequently executed for a number of *steps*, with each step responsible for simulating one day.

The computation at each step is as follows: initially, the weather conditions at each location are determined from the input data. This is followed by the calculation of the survival, development, and reproduction rates for each different type of mosquito as described in [FTD89]. The development rates are used to age each of the life stages in three phases: in the first phase, the number of organisms that will move from one stage to another are computed; the second phase ages organisms that will progress only within their current stage; and in the last stage, new-entrants to each stage are introduced. Populations at each stage are subsequently reduced as determined by the corresponding survival rate. The last part of the simulation step uses the reproduction rate for the females to determine if eggs are laid either at the home site, at a new site in the same location, or at a site in a neighboring location.

The performance summary for the UC[2] and *Lisp program was as follows: The runtime to simulate mosquito development for 365 days for a UC program was 40 minutes, as compared with over 2 hours for *Lisp. The UC program was approximately 950 lines of code, whereas the *Lisp program was 12% *longer* – 1075 lines.

Subsequently, the UC program was refined by introducing map statements to improve the execution efficiency of the simulation. Both *fold* and *copy* mappings were used as described below. The primary data structure used in the simulation are multi-dimensional arrays representing various attributes of the mosquito population: dimensions 0 and 1 represent the grid coordinates of the location, dimension 2 represents the sites and dimension 3 is used for tiers in the stages of development. In particular, four such arrays are used, one each for the four major phases in the development of the mosquito.

The following program fragment illustrates a simple instance of the fold mapping:

```
1  int eggs[ROWS][COLUMNS][SITES][TIERS],
2      larvae[ROWS][COLUMNS][SITES][TIERS],
3      pupae[ROWS][COLUMNS][SITES][TIERS],
4      females[ROWS][COLUMNS][SITES][TIERS],
5      egg_count, larva_count, pupa_count, female_count;
6
7  fold (R,C,S,T) {
8      eggs[r][c][s][t]     :- eggs[r][c][s/4][t];
9      larvae[r][c][s][t]   :- larvae[r][c][s/4][t];
10     pupae[r][c][s][t]    :- pupae[r][c][s/4][t];
11     females[r][c][s][t]  :- females[r][c][s/4][t];
12 }
```

[2]This program was developed in collaboration with Professor Chuck Taylor of the Department of Biology, UCLA.

```
13
14 void count_populations(void) {
15   ...
16     egg_count    = $+(R,C,S,T; eggs[r][c][s][t]);
17     larva_count  = $+(R,C,S,T; larvae[r][c][s][t]);
18     pupa_count   = $+(R,C,S,T; pupae[r][c][s][t]);
19     female_count = $+(R,C,S,T; females[r][c][s][t]);
20   ...
21 }
```

The computation in lines 16-19 is a series of reductions that compute the total number of mosquitoes in each of the four phases. As stated earlier, the fold mapping allows the user to control how virtual processors are simulated on each physical processor. Here, we block each array along the third axis (corresponding to mosquito sites) such that each virtual processor simulates four array elements. On the CM-2, this mapping resulted in a considerable improvement in the performance of the model, reducing the execution time of this routine to 13.25 msecs as opposed to the 23.9 msecs that was required without the mapping. Folding was done along the third axes instead of the fourth, because other parts of the program perform nearest-neighbor moves along this last axis; performance of these operations would have suffered had the fourth axis not been retained parallel.

We next illustrate the use of the copy mapping in improving execution efficiency of this simulation. Figure 8 contains a program fragment to age the female population.

The array site_used is a mask which is used as a guard in several operations. But note that, while it has three dimensions, in the last two par operations it is used to guard operations over all four dimensions. Using a copy mapping, we can make it available at all locations of use.

Similarly, surv_rate is a two dimensional array which is used several times in operations involving all four dimensions. It can be copied along the last two dimensions to make all these operations local. Similar optimizations were made in other routines of the program. The measurements given in figure 9 demonstrates the *significant* reduction in execution time that was obtained by using the map statements.

4.5 Diffusion Aggregation in Fluid Flow

This application was developed in collaboration with Dr Indranil Chakravarty, Schlumberger Laboratory for Computer Science, Austin.

The purpose of this simulation is to develop computer models that account for changes in microstructure and porosity in carbonate rocks resulting from fluid flow. The simulation begins with a model in which there is a collection of unhydrated material clusters scattered over a plane. These clusters represent the original unreacted rock grains and are separated by pore space. The proportion of the plane occupied by unhydrated material is referred to as the *packing fraction*. The proportion of the plane that is pore space is the *porosity*. The packing fraction and the initial geometric configuration of the unhydrated materials are strong determinants of the simulation result. The starting configuration for the simulation is typically an approximation to rock structures. The various implementations we report use a random placement of circles.

```
void age_females( int females[ROWS][COLUMNS][SITES][TIERS],
                  int female_jump,
                  int site_used[ROWS][COLUMNS][SITES],
                  float surv_rate[ROWS][COLUMNS])
{
  int i;
  float jumpers[ROWS][COLUMNS][SITES];
  copy (R,C,S,T)
       surv_rate[r][c] :- surv_rate[r][c][s][t];

  par (R,C,S) st (site_used[r][c][s])
  {
     jumpers[r][c][s]
          = $+(T st (t >= (TIERS - female_jump))
               surv_rate[r][c] * females[r][c][s][t]);
  ...
  }
  {
    copy (R,C,S,T)
       site_used[r][c][s] :- site_used[r][c][s][t];

    ...
    for (i = 0; i < female_jump; i++)
       par (R,C,S,T) st ((t+1 <= TIERS-1) && site_used[r][c][s])
            {
                females[r][c][s][t+1] = females[r][c][s][t];
                females[r][c][s][0] = 0;
            }

    par (R,C,S,T) st (site_used[r][c][s])
       females[r][c][s][t] = surv_rate[r][c]
                             * females[r][c][s][t];
  }
}
```

Figure 8: Copy Mapping – Mosquito Simulation

	unmapped	mapped	improvement
CM busy Time	31.578 mins	24.971 mins	20.92 %
Total Time	39.270 mins	29.442 mins	25.02 %

Figure 9: Performance Improvement for Mosquito Simulation

Dissolution takes place at the surface layer of the unhydrated materials, i.e., those unhydrated materials that are in contact with the pore space. The selection of the surface layer material that dissolves is random. The dissolved unhydrated materials become diffusing particles. The particles diffuse (perform a random walk) in pore space until they encounter a solid (unhydrated material or previously hydrated product) whereupon they become an immobile hydrate product. When all diffusing particles have settled, the process repeats again starting from the dissolution step. Taken to the extreme, the dissolution/diffusion cycle would be repeated until there were no more surface level unhydrated material. This is not necessarily the way the simulation would be used in practice.

From the physics of the problem, we must augment the number of diffusing particles immediately after the dissolution step. This new phase, which we will refer to as *volume correction*, consists of the random injection of particles into the pore space. The number of additional particles has been determined empirically [WL83] and is a proportion of the number of particles that dissolved.

An entire simulation can be viewed as an iteration consisting of three distinct steps. A random dissolution step selects surface unhydrated material to dissolve. Volume correction is then applied to augment the diffusing particle count. The diffusion phase moves the particles in random walks until they all have hydrated. The number of diffusion rounds is determined by the length of the random walk followed by the last diffusing particle to hydrate. All particles are assumed to move at the same speed. The model is assumed to admit a periodic boundary condition for the simulation grid.

Rather than include the entire UC code for the problem, we only include the code that implements the most complex operation – diffusion. A complete listing of the UC code may be found in [CKW+90].

For simplicity, we represent the simulation grid as a one-dimensional array $d[0..N-1]$ where $d[i].s$ represents the state of particle i and $d[i].select$ is the index of a neighboring D particle. Array $dest[0..N-1]$ is used to store the next step of the random walk, where $dest[i]$ denotes the potential destination of the particle in position i. Some commonly used UC reductions have been defined as C macros: *count_D* counts the total number of diffusing particles and *count_ND* counts the number of D-neighbors for a given particle. We have omitted the definition of constant identifiers like U, H, etc. The code fragment executes the diffusion step until the number of D particles is 0. In each iteration, a D particle whose potential destination is an H or U particle immediately becomes hydrated. Subsequently, all P-particles randomly select a D-neighbor (if it exists) and swap positions. If some i particle does not have a D-neighbor, $d[i].select$ will be computed to be *INF* (the identity value for the $, operator) and the corresponding particle does not move. The CM Fortran version of this code fragment was considerably longer.

```
#define count_D ($+(I st (d[i].s == D) 1)
#define count_ND ($+(J st ((d[i+j].s == D) && (dest[i+j]==i))) 1))

struct D {int s, state;} d[N];
int dest[N];
index_set I:i = {0..N-1}, J:j = {-1,1};

phase = diffusion;
```

```
while (count_D != 0) {
    par (I) st ((d[i].s == D) && ((d[dest[i]].s == H) || (d[dest[i]].s == U)))
        d[i].s = H;
    par (I) st (d[i].s == P)
    {
        d[i].select = $,(J st ((d[i+j].s == D) && (dest[i+j] == i)) i+j);
        if (d[i].select != INF) {
          d[i].s = D;
          d[d[i].select] = P;
} } }
```

The running time of the UC program was found to be almost identical with that of an equivalent CM Fortran program written for this application: for a 128 x 128 grid size with a porosity of 30 percent, each program took about 30 seconds to complete on an 8K CM-2[CKW+90].

5 Related Work

A large number of parallel languages and notations have been designed. Bal et al.[BST89] is a recent survey of parallel languages. Parallel langauges may be broadly classified into languages with explicit concurrency and synchronization and those where the parallelism is implicit and is typically extracted by the compiler with the aid of optional programmer annotations. We compare our approach with those of other explicitly parallel languages.

Explicitly parallel languages may embody either synchronous or asynchronous execution models. The language may use a universal shared memory model[BCZ90, LLG+92] where any thread can access any variable or a local memory model where each thread has exclusive access to its private data. Asynchronous languages with a local address space are perhaps the most common paradigm for parallel languages. Typically, such langauges provide explicit primitives for the specification of parallelism, interprocess communication and process synchronization. Within the asynchronous programming paradigm, example languages include CSP[Hoa78], Occam[May85], Ada[Ada83], PCN[CT91], Linda[Gel85], Maisie[BL90] and C+-[Sei90]. Although efficient implementations have been developed for most of the preceding languages, programming in these notations is complicated by the need to express the required global synchronization explicitly in the programs. Efficiency of programs written in the preceding languages is primarily determined by the process decomposition and process to processor mapping that is specified for the program. Recent notations like Strand[Tay89], PCN[CT91] and TDFL[PSJ90] attempt to simplify the problem of explicit distributed synchronization by allowing the synchronization to be specified implicitly based on an underlying data-flow graph or other mechanisms.

Asynchronous parallel programs with distributed synchronization have multiple loci of control which may complicate the programming task. Synchronous or data-parallel languages have a single locus of control and deterministic semantics making them conceptually easier to program. Data-parallel languages use the universally addressable memory model and typically provide optional data distribution primitives to specify how the program data is distributed over the memory hieracrhy of the parallel architecture. Although some languages attempt to do this purely on the strength of sophisticated compilation tools to

deduce optimum data distributions (e.g. ParallelDistributedC[HQL+91a], Crystal[CCL88] and in particular the work in [KLS90, Who91, GB92]), most provide explicit language facilities to specify data distribution[CK88]. A large number of data-parallel languages have been designed including C*[RS87], CM Fortran[Lab89], DINO[RSW91], High-Performance Fortran[For92], SUPERB[ZBG86], Paragon[Ree90], Pandore[APT90], Kali[MR90], Parlation Lisp[Sab87] and pC++[Gan93]. Some languages like C*[RS87] specify strict synchronization at the expression level, while other languages weaken the synchronization granularity and are synchronized at the block level[LR91]. The code executed between synchronization points is not allowed to access non-local data. FortranD[HKK+91] and Kali[MR90] are good examples of such langauges, which are sometimes referred to as block SIMD languages. Considering the langauge constructs, synchronization granularity, and the underlying communication model of UC, perhaps the projects that are most closely related include Kali, Dino, HPF and C*.

High Performance Fortran[For92] extends the specification of Fortran 90 with features designed to improve performance on a variety of parallel machines. The pricipal additions are data distribution features and parallel statements. The ALIGN directive allows coallocation of data, and DISTRIBUTE allows partitioning of data among memory regions. HPF provides the FORALL statement to express parallel assignment to array sections and the INDEPENDENT directive to identify statements that do not exhibit any sequentializing dependencies.

In Kali[MR90], the compiler depends on the user for data and iteration distribution. It provides a restricted set of data distribution primitives (block and cyclic) and the forall statement. Communication, which is allowed only at the beginning and end of forall loops, is generated by compile-time analysis and optimized using run-time analysis (*inspector* forall loops are generated to collect information for optimizing subsequent *executor* forall loops). Other approaches to run-time optimization of communication can be found in [DSB91, HKK+91].

In DINO[RSW91], a program consists of a host (or front-end) program which makes calls to node routines that run on each processor. Although DINO provides a more flexible SPMD model similar to that of UC, unlike UC, it requires that local and remote reference to data be distinguished statically[3]. Data distributions are specified with declarations, and may be one from among a set of predefined types (block, cyclic, replicate).

The primary difference between UC and existing data-parallel languages is the uniform treatment of arrays and relations in UC. The notion of relations and index-sets together with reductions provides a simple mechanism to express a wide variety of data-parallel computations. Our experience indicates that efficient implementations of these ideas is possible when the key for a relation is restricted to integers. Extension of this notation to a complete relational calculus is in progress. Secondly, UC allows the granularity of synchronization to be varied arbitrarily under programmer control. Langauges like C* define strict expression-level synchronization. Efficient C* implementation on asynchronous architectures requires that the compiler be able to costruct control dependency graphs that minimize global synchronizations while maintaining the semantics[HQL+91b]. On the other hand, although block SIMD synchronizations can be implemented more efficiently, they may force an unnatural program decomposition on computations where the communictaion does not follow

[3] A later version, Dino2[RS92], proposes removal of this restriction. Another variant, DYNO[WS92], uses a graph-based approach to perform dynamic load balancing.

this model[RSW91]. UC programs may be synchronized at the expression level within par statements, at the function level within parallel function calls and with arbitrary granularity using the arb statement. Thus an initial program written only using par statements may be refined subsequently by replacing some of the par statements by functionally equivalent arb statements to reduce global synchronizations. Lastly, UC provides modular and dynamic data mapping primitives that could be used to change the distribution of specific arrays in subsequent invocations of a given function.

6 Implementation Notes

UC compilers are currently operational on the CM-2 and on UNIX workstaions; compilers for networks of workstations and multicomputers are in progress. The current implementation does not support tuple reductions or dynamic index-sets. Implementation of these constructs is in progress. This section gives a brief overview of the compiler; a detailed description is in [Aus91].

The compiler is divided into three main phases: in the first phase, the compiler determines attributes of arrays and functions used in the program, the most important being whether the array or function is ever used in parallel. Next the program is transformed into an equivalent simpler UC program that only uses very simple SIMD-like assignments and reductions. Finally, this translated UC program is printed in a target language, such as C/Paris[Thi91] or C*(for data-parallel implementation), C(for sequential implementation) or PVM[BDG+91] for implementation on workstation networks.

The heart of the compiler is the rewrite rule based translator that rewrite the source UC program into an intermediate language prior to the emission of object code. The intermediate language is called UC_{simd} and was desiged to mimic the assembly language for a data-parallel architecture. A UC_{simd} program can be translated in an almost trivial manner into existing data-parallel langauges like C/Paris and C*. The rule system consists of rewrite rules that translate the source code and *primitive* rules that match a statement that belongs to UC_{simd}. Taken together, the primitive rules describe all possible constructs in the intermediate language.

A UC construct is parsed and stored internally as a tree, which is henceforth referred to as t. We rewrite a tree t using algorithm *rewrite*:

```
algorithm rewrite t
  while true
    while some rewrite rule r matches t
      rewrite t using r;
    recursively call rewrite on the subtrees of t;
    if no subtree was rewritten then
      exit the loop;
  fail if no primitive rule matches t:
```

The rewrite process succeeds only if the input tree is eventually transformed into a tree that matches a primitive rule; otherwise a diagnostic is printed and the compiler aborts. If all rules fail to match the tree, the algorithm *rewrite* is called recursively on the subtrees of the input tree; this allows rules that match a complex tree to fire before it is decomposed

by recursive calls. Most rewrite rules decompose complex constructs into simpler ones, but we also use rules to perform simple optimizations (such as loop fusion).

7 Conclusion

This paper proposes that data-parallel programming languages can be languages for relational databases extended (a) to treat arrays and relations using the same notation, (b) with reduction operators and (c) data-mapping constructs.

The novel features of the approach are:

1. The introduction of relations and index-sets, and reduction operations extended to the program composition operators **par** and **seq**.

2. Synchronization (or barriers) at different levels of granularity: either at the assignment or function level.

3. The introduction of the ***solve** command to simplify calculations leading to fixed-points.

4. A set of declarative mapping facilities which can be used by the programmer to describe the relative distribution of the program data structures on the local memories of the individual processors. If the mappings are omitted, the compiler uses simple heuristics to generate default mappings. As changes in the data mappings do not affect correctness of the UC program, efficient and portable programs may be designed for different architectures simply by modifying the data mappings.

UC compilers are available for the CM-2 and for sequential workstations running ANSI C. Measurements of the compiler for a number of applications indicates that the compiler delivers performance that is comparable to those of exiting languages on the CM-2 including C* and CM Fortran. An implementation on a network of workstations is in progress. Since the performance and expressivity results are positive, we plan to continue with an implementation of an extended relational database language.

References

[Ada83] United States Department Of Defense. *Reference Manual for the Ada Programming Language*, 1983.

[And91] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjaming Cummings, 1991.

[APT90] F. Andre, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 International Conference on Supercomputing, ACM*, 1990.

[Aus91] V. Austel. Compiling UC using source-to-source transformations. Masters report, Computer Science Department, UCLA, 1991.

[BA92] R. Bagrodia and V. Austel. *UC User Manual.* Computer Science Department, University of California at Los Angeles, 1992.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *1990 ACM Conference on Principles and Practice of Parallel Programming*, March 1990.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, B. Manchek, and V. Sunderam. A users' guide to PVM: Parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, March 1991.

[BL90] R. Bagrodia and Wen-toh Liao. *Maisie User Manual.* Computer Science Department, University of California at Los Angeles, 1990.

[BM91] R. Bagrodia and S. Mathur. Efficient implementation of high-level parallel programs. In *ASPLOS-IV*, April 1991.

[BST89] H.E. Bal, J. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Survey*, 21(3):261–322, September 1989.

[CCL88] M. Chen, Young-Il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *Journal of Supercomputing*, pages 171–207, 1988.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[CKW+90] I. Chakravarty, M. Kleyn, T. Woo, R. Bagrodia, and V. Austel. UNITY to UC: Case Studies in Parallel Program Construction. Technical Report TR-90-21, Schlumberger Laboratory for Computer Science, November 1990.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[CT91] K.M. Chandy and S. Taylor. *Introduction to Parallel Programming.* Jones & Bartlett, 1991.

[DSB91] R. Das, J. Saltz, and H. Berryman. *A Manual for PARTI runtime primitives.* Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23665, May 1991.

[For92] High Performance Fortran Forum. High Performance Fortran Language Specification. DRAFT, November 1992.

[FTD89] J. Fry, C. E. Taylor, and U. Devgan. An expert system for mosquito control in orange county california. In *Bulletin of the Soceity of Vector Ecology*, December 1989.

[Gan93] D. Gannon. pC++: High-Perfromance data parallel programming in C++. Technical report, CS Department, Indiana University., 1993.

[GB92] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):170–193, March 1992.

[Gel85] Dave Gelernter. Generative communication in Linda. *ACM TOPLAS*, 7(1), January 1985.

[Hil85] D.W. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An Overview of the Fortran D programming system. Report CRPC-TR91121, Center for Research on Parallel Computation, March 1991.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HQL+91a] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Trans. on Parallel and Distributed Systems*, July 1991.

[HQL+91b] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Robert R. Jones, and Ray J. Anderson. A production-quality C* compiler for a hypercube multicomputer. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 73–82, Williamsburg, VA, April 1991. ACM SIGPLAN. Available as SIGPLAN Notices 26, 7, July 1991.

[JT92] Fry J. and C. E. Taylor. Mosquito control simulation on the connection machine. In *Proceedings of California Mosquito and Vector Control Association*, 1992.

[KLS90] K. Knobe, J. Lucas, and G. Steele. Data Optimizations: Allocation of arrays to reduce communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8, 1990.

[Lab89] Argonne National Laboratory. Using the Connection Machine System (CM Fortran). Technical report anl/mcs-tm-118, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, June 1989.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, March 1992.

[LR91] M. Lam and M. Rinard. Coarse-grain parallel programming in JADE. In *Third Symposium on Principles and Practices of Parallel Programming*, April 1991.

[May85] David May. Communicating processes and occam. Technical note 20, INMOS Corporation, 1985.

[MR90] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. Report 90-69, Institute for Computer Application in Science and Engineering, Hampton, VA, 1990.

[PSJ90] K.M.Korner P.A. Suhler, J.Biswas and J.C.Browne. TDFL: A task-level dataflow language. *Journal of Parallel and Distributed Computing*, 13:103–115, 1990.

[Ree90] A. Reeves. The Paragon programming paradigm and distributed memory compilers. Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca NY, June 1990.

[RS87] J.R. Rose and G.L. Steele. C*: An Extended C Language for Data Parallel Programming. Technical report PL-87.5, Thinking Machines Corporation, March 1987.

[RS92] R. Rosing and R. B. Schnabel. Efficient language constructs for large parallel programs – an overview of Dino2. Technical Report Technical Report CU-CS-578-92, University of Colorado at Boulder, January 1992.

[RSW91] M. Rosing, R. B. Schnabel, and R. B. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[Sab87] G. Sabot. The paralation model as a basis for parallel programming lnguages. Technical report, Harvard University, April 1987.

[Sei90] C.L. Seitz. Multicomputers. In C.A.R. Hoare, editor, *Developments in Concurrency and Computation*. Addison-Wesley, 1990.

[Tay89] S. Taylor. *Parallel Logic Programming Techniques*. Prentice Hall, 1989.

[Thi91] Thinking Machines Corporation, Cambridge, Massachusetts. *Paris Reference Manual.*, 6.0 edition, Febuary 1991.

[Tic89] Walter F. Tichy. Parallel matrix multiplication on the Connection Machine. In Horst D. Simon, editor, *Scientific Applications of the Connection Machine*, pages 174–187. World Scientific, 1989.

[Who91] S. Wholey. *Automatic Data Mappings for Distributed Memory Parallel Computers*. PhD thesis, School of Computer Sciences, Carnegie-Mellon University, 1991.

[WL83] T.A. Witten and L.M..Sander. Diffusion Limited Aggregation. In *Physical Review*, pages 5686–5697, 1983.

[WS92] R. P. Weaver and R. B. Schnabel. Automatic mapping and load balancing of pointer-based dynamic data structures on distributed memory machines. Technical Report Technical Report CU-CS-584-92, University of Colorado at Boulder, February 1992.

[ZBG86] H. Zima, H-J Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

A Programs for Reachability

Reachability in UC

```
#define N 512

void CM_timer_clear(), CM_timer_start();
void CM_timer_stop(), CM_timer_print();

int main(void)
{
  index_set I:i = {0..N-1}, J:j = I;
  int e[N][N], r[N];

  /* initialize e so each node is connected to next higher node */
  par (I,J) {
    e[i][j] = (i==j);
    if (i<(N-1))
      e[i+1][i] = 1;
  }

  /* r[i] is true if node 0 is connected to node i */
  par (I)
    r[i] = e[i][0];

  CM_timer_clear(1);
  CM_timer_start(1);
  *solve (I)          /* continue until r stops changing */
    r[i] = $||(J; r[j] & e[i][j]);
  CM_timer_stop(1);
  CM_timer_print(1);
}
```

Reachability in CM Fortran

```
      program reachability
      implicit none
      integer, parameter  :: n=512
      logical, array(n  ) :: r, temp
      logical, array(n,n) :: e
      integer i,j

C     each node is connected to itself and the next node
      e = .FALSE.
      forall (i=1:n) e(i,i) = .TRUE.
      forall (i=1:n-1) e(i,i+1) = .TRUE.

C     r contains the nodes that node 1 is connected to
      r(1:n) = e(1,1:n)

      call CM_timer_clear(1)
      call CM_timer_start(1)
      do while (.TRUE.)
         temp = any(spread(r, DIM = 2, NCOPIES = n) .and. e,
     $        DIM = 1)
         if (all(temp .eqv. r)) then
            goto 100
         end if
         r = temp
       end do
  100  continue
       call CM_timer_stop(1)
       call CM_timer_print(1)

       end
```

Reachability in C*

```
#include <cscomm.h>
#include <stdlib.h>
#include <stdio.h>

#define N 512
#define N_PROCS 8192

int main(int argc,char *argv[])
{
  shape [N_PROCS]s1; /* shapes must have more than 512 elements */
  shape [N][N]s2;    /* 512*512 > N_PROCS, so no padding needed */
  int:s2 e, conn, s2mask;
  int:s1 r, temp, s1mask;

  /* initialize e so each node is connected to next higher node */
  with (s2) {
    e = 0;
    [pcoord(0)][pcoord(0)]e = 1;
    where (pcoord(0) < (N-1))
      [pcoord(0)][pcoord(0)+1]e = 1;
  }

  /* these masks prevent collisions in sends */
  with (s1) s1mask = (pcoord(0)<N);
  with (s2) s2mask = (pcoord(0)<N && pcoord(1)<N);

  /* r[i] is true if node 0 is connected to node i */
  with (s2)
    where (s2mask & pcoord(0)==0)
      [pcoord(1)]r = e;

  CM_timer_clear(1);
  CM_timer_start(1);
  while (1) {
    with (s1) /* send r to 2d shape */
      where (s1mask)
[pcoord(0)][0]conn = r;

    with (s2)
      where (s2mask) {
conn = copy_spread(&conn, 1, 0);  /* spread */
conn &= e;                        /* combine */
reduce(&conn, conn, 0,            /* reduce */
       CMC_combiner_logior, 0);
```

```
where (pcoord(0)==0)
  [pcoord(1)]temp = conn;          /* return result */
      }
    with (s1) {  /* stop if r hasn't changed */
      if (&=(temp==r))
break;
      r = temp;
    }
  }
  CM_timer_stop(1);
  CM_timer_print(1);
}
```