UC: A Set-Based Language for Data-Parallel Programming

Rajive Bagrodia* Mani Chandy+ Maneesh Dhagat*

*Computer Science Department, UCLA, Los Angeles, CA 90024
+Computer Science Department, Caltech, Pasadena, CA 91125

Abstract

This work was motivated by the success of set-based languages and methodologies: SETL allows succinct mathematical representation of algorithms; specification notations like UNITY that use index-sets together with a few composition operators to design parallel programs for a large class of applications. Van de Velde[dV93] has developed a stepwise refinement methodology for parallel programming based on index sets; he has applied these methods to a variety of scientific applications. Williams [Wil93] has developed applications using voxel databases which are relations defining points in space. This literature suggests further exploration of the use of sets in parallel programming. 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-2 and for sequential workstations. Compilers for the SP1, a distributed memory machine, and and a network of workstations 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.

1 Introduction

This work was motivated by the success of set-based languages and methodologies. The set-based languages SETL[FSS83] allows for succinct mathematical representations of algorithms. The value of relations in data-parallel computing has been demonstrated by Paralation Lisp[Sab88]. A large class of distributed algorithms have been developed and proved correct using UNITY[CM88], a set-based notation. Van de Velde[dV93] has developed a stepwise refinement methodology for parallel programming based on index sets; he has applied these methods to a variety of scientific applications. Williams[Wil93] has developed applications using voxel databases which are relations defining points in space. This literature suggests further exploration of the use of sets in parallel programming.

We have designed a parallel language called UC that incorporates relations and sets into C. Significant features of UC include:

- the introduction of relations and index-sets into C together with composition operators that use the standard mathematical notation for reductions.

- the introduction of synchronization, or barriers, at multiple levels of granularity. The parallel composition operators provided by UC allow a programmer to *syntactically* specify the synchronization granularity of a set of statements to be at the expression-level, at the level of compound statements, or at the function level where multiple threads are synchronized only at the point of call and return, with the multiple function instances executed asynchronously.

- the use of dynamic and modular mapping primitives that can be used for data decomposition and code allocation.

UC has been used to design a diverse set of applications that include search algorithms, graph algorithms, computational fluid dynamics applications, and numerical applications. Data-parallel UC programs can be executed on a CM-2 array processor; experimental measurements of the compiler indicate that performance of the UC compilers is comparable to that of commercial languages like C∗, CM Fortran and *Lisp[ABCD93]. Implementations on SP1, a distributed memory machine, and a network of workstations are in progress.

The next section describes the UC language. Section 3 briefly introduces its mapping facilities. A brief description of the UC implementation is presented in section 4. Section 5 describes a number of UC examples and presents performance results from existing UC implementations and compares its performance with other languages such as CM Fortran, and C∗. Section 6 discusses related work; in particular we discuss the relationship of our work to previous work on languages like HPF, Parlation LISP and SETL. Section 7 is the conclusion.

2 Sets and Quantification

The central concepts of the notation are the well-known mathematical concepts of sets and quantification which have been used in programming languages[FSS83, CM88, Sab87, dV93]. We use an abstract notation for describing concepts, because the specific C-like syntax we use in the implementation is not central to the ideas described in this paper.

Range A *range* is a special case of a set of integers with constant lower and upper bounds on values of members of the set; the bounds are specified at the point of declaration of the range. The declaration of a *range* is of the form:

```
range R:r = {0..N-1};
```

which declares `R` to be a *range*, and `r` to be a dummy variable that runs over `R`, where all members of `R` lie in the closed interval `[0, N-1]`. Operations on sets can be carried out on ranges with the restriction that all elements of a range must always lie within the constant bounds specified in its declaration.

Relations and Index-Sets A *relation* is a table with several columns, each containing elements of a specific type. One (or more) columns must be distinguished as the *key*: each row in the table must contain a unique entry for this column. For example, a table of employees may have the social security number as the key. A relation is unordered.

A one-dimensional array may be viewed as a relation with two columns: the first containing the index, and the second containing the value. If the array is sparse, it may be represented by a relation which only stores rows with non-zero values. Multi-dimensional arrays are relations in which the keys are tuples of all the indices. Sets of points, vertices, or employees can all be represented as relations.

For each relation `R` we define a *zero element*, and we adopt the convention that the value of `R[k]` is the zero element of the relation if there is no entry in the relation with key `k`. 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.

Declaring a Relation The definition of a relation-type is similar to that of a record and has the form:

relation *name* { *non-keys* } **key** { *key-fields* }: *dummy_variables*;

Example:

relation `PIXEL` { **int** `color, intensity;` } **key** { **int** `x, y`} : `p,q;`

is a relation type, `PIXEL`, with two non-key fields called `color` and `intensity`, and two key fields `x` and `y`; the dummy variables for its key are called `p` and `q`. A dummy variable may be used in the program to refer to an arbitrary tuple in the relation.

The declaration of a variable of type relation specifies its maximum size, as in:

```
PIXEL pixel[N];
```

which defines `pixel` to be a variable of type `PIXEL` and with a maximum of `N` rows. (A more sophisticated implementation will not require the maximum size to be specified.) We adopt the convention of using upper-case identifiers to refer to index-sets and the corresponding lower-case identifier to refer to a dummy variable of the index-set.

An *index-set* is a relation that has no non-key fields. Index-set types and variables are declared in a way similar to relations. Index-sets are commonly used as a key to reference a subset of the elements in a relation. Every relation is implicitly associated with an index-set called **ALL**; this set includes the key field of every tuple in the corresponding relation.

Unlike a range, an *index-set* does not have constant bounds. It is a dynamic set used when the number of values stored in the set is small relative to the total number of possible values. For example, an application may need a dynamic set in which values are drawn from the universe of all possible nine-digit social security numbers.

2.1 Quantification

Quantification A quantification is of the form[CM88]:

$$\langle op\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \mathtt{s(i)} \rangle$$

where *op* is an associative, commutative operator, `I` is a set, `i` is a dummy variable, `b(i)` is a boolean expression, and `s(i)` is an expression or statement. For instance,

$$\langle \forall\ \mathtt{i} \in \mathtt{I} : \mathtt{0} \leq \mathtt{i} < \mathtt{3} : \mathtt{c[i]} = \mathtt{0} \rangle$$

means that for all `i` in set `I` such that $0 \leq \mathtt{i} < 3$, the equation `c[i] = 0` holds. Likewise,

$$\langle + \ \mathtt{i} \in \mathtt{I} : \mathtt{0} \leq \mathtt{i} < \mathtt{3} : \mathtt{e[i]} \rangle$$

is the sum of `e[i]` over all `i` in set `I`, where $0 \leq \mathtt{i} < 3$. The notation supports addition, multiplication, maximum, minimum, universal quantification, and existential quantification over sets. The notation also has a non-deterministic operator that returns an arbitrary member of a set.

Quantification is extended to control-flow operators that are associative and commutative.

$$\langle \mathbf{par}\ \mathtt{i} \in \mathtt{I} : \mathtt{0} \leq \mathtt{i} < \mathtt{3} : \mathtt{s(i)} \rangle$$

where `s(i)` is an assignment statement with parameter `i`, is the parallel execution of `s(i)`, for all `i` in set `I`, where $0 \leq \mathtt{i} < 3$. The **par** operator is discussed later.

Operations on subsets are obtained using **select**:

$$\langle \mathbf{select}\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \mathtt{e(i)} \rangle \ = \ \{\mathtt{e(i)} | (\mathtt{i} \in \mathtt{I}) \wedge \mathtt{b(i)}\}$$

For example:

$$\langle \mathbf{select}\ \mathtt{i} \in \mathtt{I} : \mathtt{i} > 0 : \mathtt{i} * \mathtt{i} \rangle$$

where `I` is the set {`-3, 1, 4`} is the set {`1,16`}.

In programs, we use the keyword **in** for $\in$, and the logical *and* operator `&&` for $\forall$, the logical *or* operator || for $\exists$, < for minimum, and > for maximum. In programs we use:

$$\$op(\mathtt{i}\ \mathbf{in}\ \mathtt{I}\ \mathbf{st}\ (\mathtt{b(i)})\ \mathtt{s(i)})$$

for

$$\langle op\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \mathtt{s(i)} \rangle$$

where *op* is an arithmetic, boolean, or selection (min, max, select) operator; we use the same notation without the **$** if *op* is a program composition operator. Comments are enclosed between `/*` and `*/`, as in C.

Example The following algorithm to calculate prime numbers using the sieve method of Eratosthenes illustrates the use of quantification. Given a sequence of integers starting with 2, at each iteration of the loop the algorithm is as follows: compute the smallest integer in the sequence, say `nextp`, and mark it as a prime number. Then remove all multiples of `nextp` from the sequence. This is repeated until the sequence is empty. The UC program is shown in Figure 1; performance measurements of this program are reported subsequently.

2.2 Asynchronous Composition

The keyword **arb** (for arbitrary) is the operator for composing programs asynchronously. The operation of *s* **arb** *t*, where *s* and *t* are statements, is concurrent execution of *s* and *t* using arbitrary interleaving. The operator **arb** is associative and commutative.

For example, { `a = y;` } **arb** { `b = z;` } assigns the values of `y` and `z` to `a` and `b` respectively, if the memory locations of `a` and `b` are different from each other, and different from those of `y` and `z`. However, the effect of the execution of {`z = y;`} **arb** {`b = z;`} is

```
main (void) {
   int prime[N];            /* initialized to 0 */
   range I:i = {2..N-1};   /* assume N > 2 */
   int nextp = 2;

   while (nextp <= N) {
      prime[nextp] = 1;                             /* prime[i] is 1 if i is prime */
      I = $select(i in I st (i%nextp != 0) i);      /* I = {i | (i mod nextp) ≠ 0 } */
      nextp = $<(i in I st (true) i);               /* nextp = minimum element of I } */
   }
}
```

Figure 1: Sieve of Eratosthenes

not specified, because the atomicity and interleaving of operations between the concurrently executing statements `z = y;` and `b = z;` are not specified. For example, both interleavings `z = y; b = z;` and `b = z; z = y;` are permissible, and each of these interleavings leads to different results.

Syntactic Sugar In programs, for

$$\langle \mathbf{arb}\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \mathtt{s(i)} \rangle\ \mathbf{arb}\ \langle \mathbf{arb}\ \mathtt{i} \in \mathtt{I} : \neg\mathtt{b(i)} : \mathtt{t(i)} \rangle$$

we write

arb(i **in** I) **st**(b(i)) s(i); **others** t(i);

If the dummy element has been declared together with the corresponding index-set or range, it may be omitted in the statement. For instance, given the declaration

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

the statement

arb(i **in** I) **st**(b(i)) s(i);

can be written as:

arb(I) **st**(b(i)) s(i);

and if the boolean expression `b(i)` is true for all `i`, it can be omitted, as in

arb (I) s(i);

Example 1 A program to transpose a matrix `A` is:

```
arb (I,J) temp[i][j] = A[i][j];
arb (I,J) A[i][j] = temp[j][i];
```

This program fragment consists of the sequential composition of two statements, each of which is an asynchronous composition of $N \times N$ assignments, where `A` is an $N \times N$ matrix; the first of the two composed statements assigns matrix `A` to a matrix `temp`, and the second assigns the transpose of `temp` to `A`.

The result of the fragment:

```
arb (I,J) A[i][j] = A[j][i];
```

is not specified because the atomicity and interleaving of each of the $N \times N$ statements is not specified.

Sequential composition and asynchronous parallel composition do not commute. For instance, the following program fragment has a different effect than the fragment in example 1.

Example 2

```
arb (I,J) {
   temp[i][j] = A[i][j];
   A[i][j] = temp[j][i];
}
```

Unlike example 1, here there is no barrier between the execution of the two statements.

2.3 Parallel Assignment Composition

Assignment statements can be composed using the parallel composition operator **par**. The execution of { x = e; } **par** { y = f; } is as follows: the expressions e and f are evaluated, and then their values are assigned to x and y in parallel. Thus the values of x and y after the execution of this is the same as their values after the execution of:

```
{ temp_x = e;} arb { temp_y = f;} ;
/* barrier */
{ x = temp_x;} arb { y = temp_y;} ;
```

where temp_x and temp_y are distinct fresh variables not named in expressions e and f.

The parallel composition operator is associative and commutative. We use the convention that the parallel composition operator has higher binding power than the asynchronous composition operator. The syntactic conventions adopted for the **arb** operator also apply for statements composed with the **par** operator.

Parallel composition is a convenience rather than a necessity, since parallel composition can be implemented in terms of sequential composition. The convenience is, however, extremely valuable in programs for SIMD machines.

Syntactic Sugar In programs,

$$\langle \mathbf{par}\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \mathtt{s(i)} \rangle\ \mathbf{arb}\ \langle \mathbf{par}\ \mathtt{i} \in \mathtt{I} : \neg\mathtt{b(i)} : \mathtt{t(i)} \rangle$$

is represented by

$$\mathbf{par}(\mathtt{i}\ \mathbf{in}\ \mathtt{I})\ \ \mathbf{st}(\mathtt{b(i)})\ \ \mathtt{s(i)};\ \mathbf{others}\ \mathtt{t(i)};$$

We adopt the convention that the body of a **par** statement can be a block consisting of variable declarations followed by a sequence of assignment statements. Each assignment statement in the block is evaluated synchronously. For instance, the following fragment

```
par (I,J) {
   a[i][j] = a[j][i];
   a[i][j] = a[(i+1)%N][j];
}
```

is equivalent to:

```
par (I,J) a[i][j] = a[j][i];
par (I,J) a[i][j] = a[(i+1)%N][j];
```

The iterative construct **while** within a **par** quantification has the following meaning:

$$\langle \textbf{par}\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} : \textbf{while}(\mathtt{c(i)})\ \mathtt{s;}\rangle$$

is the same as:

$$\textbf{while}(\langle \exists\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} \wedge \mathtt{c(i)}\rangle)\ \ \langle \textbf{par}\ \mathtt{i} \in \mathtt{I} : \mathtt{b(i)} \wedge \mathtt{c(i)} : \mathtt{s}\rangle$$

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 `s` is executed synchronously for every enabled element. The meaning of other iterative constructs is similar.

Example: All-Pairs Shortest Path 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`. In the program, we omit the initialization and definition of the macro `min` which returns the smaller of its two arguments.

```
int k, dist[N][N]; /* initialized appropriately */
range I:i = {0..N-1}, J:j = I;
for (k = 0; k < N; k++) {
   /* ⟨∀i, j : i ∈ I ∧ j ∈ J : */
   /*      dist[i][j] = min(dist[i][k] + dist[k][j], dist[i][j]); ⟩ */
   par (I,J)
      dist[i][j] = min(dist[i][k]+dist[k][j], dist[i][j]);
}
```

If a **par** statement includes a function call, multiple instances of the function execute asynchronously. The program fragment

```
par (I) f(x[(i+1)%N], y[i]);
```

is semantically equivalent to the following fragment:

```
arb {
   copy_x[i] = x[(i+1)%N];
   copy_y[i] = y[i];
};
arb (I) f(copy_x[i], copy_y[i]);
```

3 Data Mappings

UC provides a *map* construct to describe various types of data distributions[BM91]; UC data mappings are similar to the mappings provided by HPF. Three types of data mappings are supported: **perm**, to align a data structure with respect to another; **fold** to decompose aggregate data structures (e.g. to specify block partitioning for arrays), and **copy** to replicate parts of a data structure. Any declaration block of a UC program may contain a map statement, allowing UC mappings to be dynamic.

A map statement is syntactically similar to a **par** statement and has the following parts: the *type of mapping* which may be one of **perm**, **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 target may be a virtual data structure that does not occupy physical memory; its only purpose is to define a template that is used to describe subsequent mappings. For instance, the following statement specifies that the transpose of array `a` should be aligned with array `b`.

```
int a[N][M], b[M][N];
perm (I,J)
   a[i][j] :- b[j][i];
```

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 at compile-time. 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*.

For the **copy** mapping, 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; this may 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 **fold** mapping is a tool to partition arrays when the number of available physical processors 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. The following example uses a virtual data structure to partition an array:

```
/* 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];
  ⋮
}
```

For simplicity, assume that K is a multiple of F1. 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 – a way of perfoming load balancing. Note that use of dynamic mappings may cause data movement and hence may require communication.

We use a UC program that models mosquito populations to demonstrate the use of data mappings in improving execution efficiency of a program. The example and data mappings together with the experimental measurements are described in section 5.

4 Implementation

UC compilers are currently operational on the CM-2 and on UNIX workstations; compilers for networks of workstations and multicomputers are in progress. The current implementation does not support sparse relations, dynamic mappings. A detailed description of the UC compiler is in [Aus91]. This section gives a brief overview of the existing compiler for the CM-2 and concludes with a discussion of the proposed multicomputer implementation.

The compiler is divided into three phases: in the first phase, the compiler determines various attributes of the aggregate data structures and functions used in the program. For instance, unlike other data parallel languages (e.g. C*, HPF, or DINO), UC does not require parallel data structures to be distinguished syntactically from sequential structures. For each aggregate data structure and function, the compiler must determine whether the array or function is ever used in parallel.

The second phase is a rewrite rule based translator that rewrites the source UC program into an intermediate language, consisting of simpler assignment, communication, and synchronization operations. The rule system consists of a set of rewrite rules which are repeatedly applied to transform the source code to the intermediate language. In particular, parallel assignments with complex expressions that involve a number of message communications are decomposed into simpler assignments that use one of the standard communication patterns like reduction, scan, spread, broadcast, etc. In the current compiler, most rewrite rules are used to decompose UC constructs into simpler ones; a few rules have been implemented to perform simple optimizations like loop fusion. Additional rules for optimizations like eliminating temporaries, reducing synchronization points, and merging communication statements are being added to the compiler.

In the third and final phase, this translated UC program is printed in a target language, such as C/Paris[Thi91] or C*(for synchronous architectures like the CM-2), ANSI C (for the sequential implementation) or Maisie[BL90], a message passing language (for implementation on asynchronous architectures and workstation networks).

4.1 Multicomputer Implementation

For multicomputer implementations, aggregate data structures like arrays are decomposed as specified by the map statements, and operations are executed using the owner computes rule.

The implementation of the **arb** statement is straightforward: each processor executes the body of an arb statement independently, and then synchronizes which other processors at the barrier. Parallel assignments are implemented by transforming them into arb statements while minimizing the number of barrier synchronizations[PDB93, HQ91]. Reductions are implemented by subdividing them into two parts. In the first part, each processor computes the partial result of combining selected values stored locally. In the second part, the final value is computed by doing a reduction on the partial results of all processors. For instance, the *sum*-reduction, `x = $+(I st (b[i]) v[i]);` is transformed into the following code:

```
range P:p = {0..PROCS-1};
arb (P)
   /* processor p stores values v[start[p]] to v[stop[p]-1] */
   for (index[p] = start[p]; index[p] <= stop[p]; index[p]++)
      if (b[index[p]])   partial_sum[p] += v[index[p]];
x = $+(P; partial_sum[p]);
```

Ranges and Index-sets Efficient implementation of index-sets is key to efficient UC implementations. As a relation may be viewed as an index-set that also includes non-key data, strategies for its implementation are closely linked with implementations of index-sets. We outline the implementation for index-sets for distributed memory architectures.

A range can be implemented efficiently simply by storing its lower and upper bound; if the range is dynamic, it is implemented using a bit array. For instance, a statement such as

```
I = $select(i in I st (i%next != 0));
```

is translated as

```
par (I) st (mask[i] && (i%next != 0))
   mask[i] = 1;
```

where `mask` is the bit array that represents range `I`.

Sparse relations (and index-sets) can be implemented using a conventional hashing scheme like that used in SETL[FSS83]. For simplicity, assume that the number of buckets is equal to the number of physical processors in the architecture and that a default system-defined hashing function is used. For a given index-set (and relation) each processor stores the set of primary keys that hash to its local memory. Let k_i refer to the set of keys for the relations that are stored in the local memory of processor p_i. Set k_i will depend on the specific hashing function that is used for the relation.

For many operations, including some commonly used data parallel operations, the access time for elements of a sparse relation is reasonably close to that of dense index-sets. Reductions and parallel assignments that update a subset of tuples in a relation are particularly efficient provided that the subset is selected based on some *local* characteristic of each tuple. For instance, execution of following statement does not entail additional (computation) overhead even if the index-set `S1` is sparse: each p_i will execute the statement for the susbset of its k_i that satisfy the predicate.

```
par (S1) st (emp[s1].salary >= 50000)
   emp[s1].salary += 0.1 * emp[s1].salary;
```

Random access to specific elements is more expensive in the case of sparse sets, but only by a constant factor proportional to the additional time to execute the hashing function. An example of such a fragment is

```
par (S1) st (emp[s1].salary >= emp[emp[s1].boss].salary)
   emp[s1].salary -= 0.1 * emp[s1].salary;
```

where for each element e, the location of the record for e.`boss` must be computed. Of course, if the operation occurs frequently, the implementation could transparently store this information to reduce computation overhead at the cost of increased memory consumption.

5 Examples and Performance

The current UC implementations have been used to design applications in a variety of areas that include Computational Fluid Dynamics[CKW+90], Partial Differential Equation Solvers[MACV93], Computational Biology[JT92], 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. In particular, UC 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 UC programs across different application spectrums and provide measurements on their performance. In addition, comparative timings are presented for C∗ CM Fortran, or *Lisp versions of the program as appropriate. Unless stated otherwise, all measurements reported here were taken on a CM-2 with 16K processors (slicewise compilation) with floating-point accelerators, and a SUN front-end.

5.1 Computing Primes

The UC implementation of parallel sieve was described earlier in figure 1. A slightly different version of that program was also developed, where the index-set is constant, and a separate array is used to mark multiples of the identified prime numbers. The graph in figure 2 plots the execution time of the two programs. It shows that **$select** and index-set operations can be supported by UC without significant performance degradation. Additionally, the graph in figure 3 plots the execution time of the UC, C∗ and CM Fortran programs for this algorithm as a function of N, the size of the input sequence.

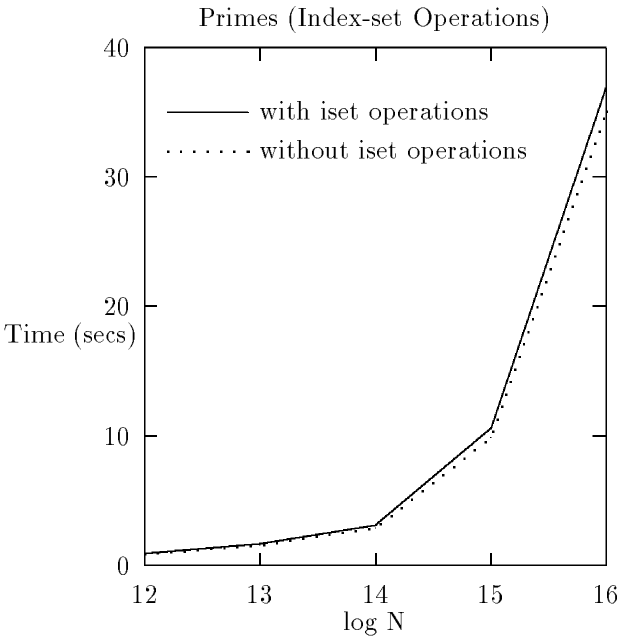

Figure 2: Performance of Index-set Operations

5.2 Graph Reachability

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 graph in figure 3 shows the execution time of the three programs with increasing problem size; the largest arrays in the programs are of size N^2.

5.3 Shortest Path

The UC code for this example was presented in section 2.3. The graph in figure 3 plots the execution time of the UC, C∗ and CM Fortran programs for this algorithm with increasing values of N; the largest arrays in the programs are of size N^2.

5.4 Matrix Multiplication

Using N^2 processors, a simple program for matrix multiplications is

```
for (k = 0; k < N; k++)
   par (I,J)
      c[i][j] += a[i][k] * b[k][j];
```

However, each multiplication involves expensive remote accesses.

An alternative systolic algorithm for the CM-2 that uses N^2 processors is described in [JHM89]. The algorithm has two distinct phase - *alignment* and *systolic multiplication*. Alignment permutes the arrays such that the multiplicands are available locally. The data motion in the systolic multiplication phase of the algorithm requires nearest neighbor communication only.

```
int main (void) {
   int a[N][N], b[N][N], c[N][N], k;
   range I:i = {0..N-1}, J:j=I;

   par (I,J) {
      a[i][j] = a[i][(i+j)%N];         /* Alignment Phase */
      b[i][j] = b[(i+j)%N][j];

      c[i][j] += a[i][j] * b[i][j];    /* Multiplication Phase */
      for (k = 1; k < N; k++) {
         a[i][j] = a[i][(j+1)%N];
         b[i][j] = b[(i+1)%N][j];
         c[i][j] += a[i][j] * b[i][j];
      }

      /* Re-alignment Phase -- omitted */
} }
```

The graph in figure 3 plots the execution time of the UC, C* and CM Fortran programs for this algorithm with increasing values of N.

5.5 Gaussian Elimination

We implemented the Gaussian elimination[CLR91] scheme for solving a set of linear equations of the form $Ax = b$, where A is a square matrix of coefficients, and x and b are column vectors of unknowns and constants, respectively. The following code implements the LU decomposition phase:

```
int main(void) {
   double A[N][N], tmp[N];
   range L:l = {0..N-1}, I:i=L, J:j=L;
   int z, pivot;
   double max;

   for (z = 0; z < N; z++) {
      max = $>(I; fabs(A[i][z]));
      pivot = $<(I st (max == fabs(A[i][z])) i);    /* good pivot chosen */

      if (pivot < N)
         par (L) {
            tmp[l] = A[z][l];
            A[z][l] = A[pivot][l];
            A[pivot][l] = tmp[l];
         }
      else break;
```

```
        I = I - z;  J = J - z;
        par (I) A[i][z] /= A[z][z];    /* Calculation of Multiplier Column */
        par (I,J)  A[i][j] -= A[i][z] * A[z][j];     /* Elimination */
    }
}
```

For each of the three implementations, we measured the execution times for two different input matrices: one which required no pivoting; and another that required pivoting at every step. The graph in figure 4 plots the execution time of the UC, C$*$ and CM Fortran programs (for both the pivoting and non-pivoting cases) with increasing values of N; the largest arrays in the programs are of size N^2.

5.6 Simulation of Mosquito Control

A biological model of the life of *Culex pipiens* mosquitoes[FTD89] was originally programmed in $*$Lisp. The model was executed on the CM-2 to study the effect of differnet level of pesticides on mosquito populations in Orange County, California[JT92]. Subsequently the model was programmed in UC with the expectation that the program could be executed more efficiently without risking a significant increase in the code size. The performance summary for the UC 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. The published results on the *Lisp program contained data for only one configuration which was used for the preceding comparison. Subsequently, the UC program was refined by introducing data mappings which improved the performance of the model by another 25%. Figures 5 and 6 respectively illustrate the type of *fold* and *copy* mappings that were used.

In figure 5, the primary data structure in the model are a set of four multi-dimensional arrays called `egg`, `larva`, `pupa` and `adult`. The reductions in lines 12-15 compute the total number of mosquitoes in each stage of evolution. 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 this axis because other parts of the program perform nearest-neighbor moves along the other axes.

In Figure 6, a 3-D array `site_used` is used as a guard in several operations that involve 4-D arrays. The *copy* mapping allows this array to be replicated such that it is available 'locally' when needed. Similarly, `surv_rate` is a 2-D array which is used several times in operations involving 4-d arrays. 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 7 demonstrates the *significant* reduction in execution time that was obtained by using the map statements.

5.7 Diffusion Aggregation in Fluid Flow

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.

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. This section includes 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`. A commonly used UC reduction has been defined as a C macro: `count_D` counts the total number of diffusing particles. 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)

struct D {int s, state;} d[N];
int dest[N];
range 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].

6 Related Work

This paper builds on work done on data-parallel languages like Kali[MR90], DINO[RSW91], FortranD[HKK+91] and HPF[For92], as well as on collection-oriented languages like SETL[SDDS86] and Paralation Lisp[Sab88]. We briefly consider its relationship with languages in each category.

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 hierarchy of the parallel architecture. 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. The code executed between synchronization points is not allowed to access non-local data. FortranD and Kali are good examples of such languages, which are sometimes referred to as block SIMD or SPMD languages. The primary difference between UC and existing data-parallel languages is its support for variable granularity of synchronization and its use of sets to support data-parallelism.

The synchronization granularity specified by a program determines the computation model presented to the programmer and the efficiency with which the languages can be implemented on asynchronous architectures. Thus, languages that are synchronized at the operation level have simple semantics, as the programmer has access to the correct global state of the program before executing any operation. However, implementing such a strict notation on an asynchronous distributed memory architecture can, in general, be expensive as a barrier may potentially be required before the execution of every operation. This is particularly the case if the language uses a universal shared memory model where no syntactic distinction is made between local and remote data. In contrast, a language with a weaker synchronization model forces the programmer to restrict remote data access to specific points in their code. In other words, this places the burden of maintaining a coherent view of the shared data explicitly on the programmer. However, an implementation of this model on an asynchronous architecture requires a smaller number of barrier synchronizations as compared with the previous model. Reducing barrier synchronizations improves execution efficiency by reducing both the blocking and communication times. By supporting synchronization at multiple levels of granularity, UC supports an iterative approach to program design where an initial program can be designed using expression-level synchronization and can subsequently be refined for efficient implementation on asynchronous architectures.

SETL[SDDS86] is a high-level prototyping language that uses dynamic, heterogeneous sets as its primary data type. Control-flow is specified in SETL using the standard sequential constructs; presumably parallelizing compilers may be used to parallelize SETL programs much like they are used with Fortran loops, although the dynamic typing may make compile-time dependence analysis less effective for SETL loops. A parallel variant of SETL called *Parallel SETL*[HK93] has been proposed recently. Parallel SETL attempts to exploit parallelism in a SETL program by explicitly specifying parallel executions of loops over both ordered and unordered collections. Using different keywords, the language can specify simultaneous SIMD or MIMD execution of multiple iterations of a loop that roughly corresponds to using the UC par or arb composition operators. The language also provides a standard par-begin par-end construct to specify asynchronous execution of a set of statements; however all communication requires global synchronization. Another prototyping effort is *PROSET*[Has93] that integrates the generative communication constructs of Linda[Gel85] into SETL to design a task-parallel variant of SETL with multiple tuple spaces.

Paralation Lisp[Sab88, BS90] is a data-parallel language which is based of notion of a group of related collections, called a *paralation*. A single communication mechanism is provided and it is syntactically divided into two parts: setting up a communication pattern and actual data movement. A communication pattern in Paralation Lisp is created by matching values of elements in two fields. For this reason, communication is harder to optimize by the compiler as architecture specific code can not be generated. In UC, the nature of the communication – whether it is a neighbor communication, broadcast, reduction, etc. – is usually visible at compile time by inspecting array subscripts or the parallel construct being used. Additionally, Paralation Lisp collections are dynamically typed, hence complete data-structure information is not available until run-time.

7 Conclusion

This work was motivated by the success of set-based languages and methodologies: SETL allows succinct mathematical representation of algorithms; specification notations like UNITY that use index-sets together with a few composition operators to design parallel programs for a large class of applications. Van de Velde[dV93] has developed a stepwise refinement methodology for parallel programming based on index sets; he has applied these methods to a variety of scientific applications. Williams [Wil93] has developed applications using voxel databases which are relations defining points in space. This literature suggests further exploration of the use of sets in parallel programming.

We have designed a parallel language called UC that incorporates relations and sets into C. The results are encouraging. Many parallel programming problems can be expressed well using sets. Performance of programs is comparable to performance of programs in languages that are not designed with sets as the basis. 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 existing languages on the CM-2 including C$*$ and CM Fortran. An implementation on a network of workstations is in progress.

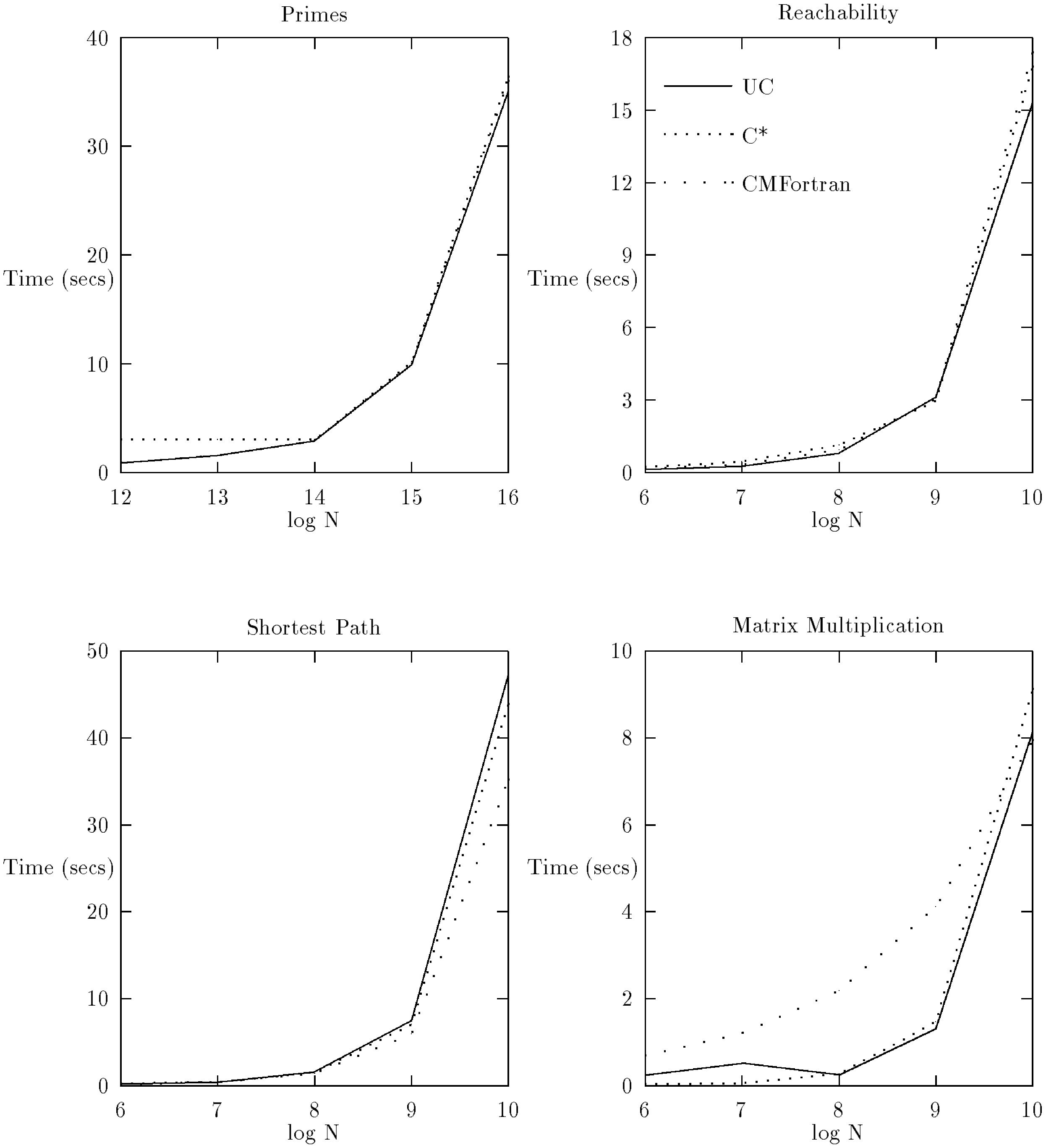

Figure 3: Performance Comparison

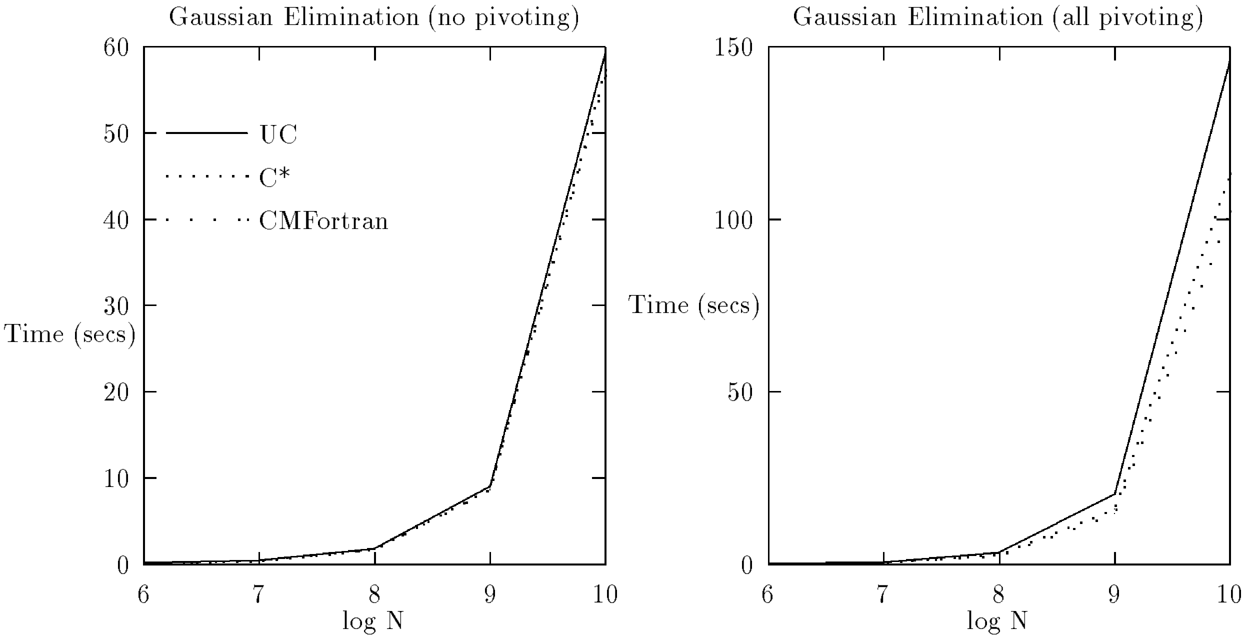

Figure 4: Performance Comparison (continued)

```
int egg[RW][CL][ST][TI], larva[RW][CL][ST][TI],
    pupa[RW][CL][ST][TI], adult[RW][CL][ST][TI],
    egg_count, larva_count, pupa_count, adult_count;

fold (R,C,S,T) {
   egg[r][c][s][t]     :- egg[r][c][s/4][t];
   larva[r][c][s][t]   :- larva[r][c][s/4][t];
   pupa[r][c][s][t]    :- pupa[r][c][s/4][t];
   adult[r][c][s][t]   :- adult[r][c][s/4][t];   }

void count_populations(void) {
   egg_count   = $+(R,C,S,T; egg[r][c][s][t]);
   larva_count = $+(R,C,S,T; larva[r][c][s][t]);
   pupa_count  = $+(R,C,S,T; pupa[r][c][s][t]);
   adult_count = $+(R,C,S,T; adult[r][c][s][t]);
 ...
}
```

Figure 5: Fold mapping in Mosquito Simulation

```
void age_adult( int adult[RW][CL][ST][TI],
                int fj,site_used[RW][CL][ST],
                float surv_rate[RW][CL])
{
  int i ;
  float jumpers[RW][CL][ST];
  copy (R,C,S,T) {
      surv_rate[r][c] :- surv_rate[r][c][s][t];
      site_used[r][c][s] :- site_used[r][c][s][t];
  }
  par (R,C,S) st (site_used[r][c][s])
     jumpers[r][c][s] = $+(T st (t >= (TI - fj)) surv_rate[r][c] *
                           adult[r][c][s][t]);
  ⋮
  par (R,C,S,T) st (site_used[r][c][s])
    adult[r][c][s][t] = surv_rate[r][c] * adult[r][c][s][t];
       ⋮
```

Figure 6: 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 7: Performance Improvement for Mosquito Simulation

References

[ABCD93] V. Austel, R. Bagrodia, M. Chandy, and M. Dhagat. Reductions + Relations = Data-Parallelism. Technical Report 930009, University of California, Los Angeles, CA 90024, April 1993.

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

[BS90] G. E. Blelloch and G. W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[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.

[CLR91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw Hill Book Company, 1991.

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

[dV93] E. Van de Velde. *Concurrent Scientific Computation.* Cambridge University Press, 1993.

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

[FSS83] S. Freudenberger, J. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM TOPLAS*, 5(1), January 1983.

[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.

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

[Has93] W. Hasselbring. *Protyping Parallel Algorithms with PROSET-Linda*, volume 734 of *Lecture Notes in Computer Science, Parallel Computation, (Proc. Second International ACPC Conference)*, pages 135–150. Springer-Verlag, Gmunden, Austria, October 1993.

[HK93] S. F. Hummel and R. Kelly. A Rationale for Massively Parallel Programming with Sets. *Journal of Programming Languages*, 1(3):187–207, 1993.

[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.

[HQ91] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1991.

[JHM89] S. L. Johnsson, T. Harris, and K. K. Mathur. Matrix Multiplication on the Connection Machine. Technical Report NA89-3, Thinking Machines Corporation, Cambridge, MA 02142, 1989.

[JT92] Fry J. and C. E. Taylor. Mosquito Control Simulation on the Connection Machine. In *Proceedings of California Mosquito and Vector Control Association*, 1992.

[MACV93] I. D. Mishev, V. Austel, T. F. Chan, and P. S. Vassilevski. Experiments with Algebraic Multilevel Preconditioners on Connection Machine. Technical Report CAM Report No. 93-25, UCLA Mathematics Department, Los Angeles, CA 90024, 1993.

[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.

[PDB93] S. Parkash, M. Dhagat, and R. Bagrodia. Synchornization Issues in Data-Parallel Languages. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[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.

[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.

[Sab88] G. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, 1988.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

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

[Wil93] R. Williams. Voxels. Technical Report CRPC Technical Report, Caltech, Pasadena, CA 91125, February 1993.

A Programs for Reachability

A.1 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], old_r[N];
  bool continue = TRUE;

  /* 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);

  continue = $&&(I; r[i] == old_r[i]);
  while (continue)
    par (I) {        /* continue until r stops changing */
      r[i] = $||(J; r[j] & e[i][j]);
      continue = !($&&(I; r[i] == old_r[i]));
      old_r[i] = r[i];
    }
  CM_timer_stop(1);
  CM_timer_print(1);
}
```

A.2 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
```

A.3 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);
}
```