# UC: A Language for the Connection Machine

R. Bagrodia
Computer Science Dept
UCLA
Los Angeles, CA 90024

K.M.Chandy
Computer Science Dept
Caltech
Pasadena, CA 91125

E. Kwan
Computer Science Dept
UCLA
Los Angeles, CA 90024

**Abstract**

In designing parallel languages, the concern for defining a simple virtual machine must be balanced against the need to efficiently map a program on a specific architecture. UC addresses this problem by separating the programming task from efficiency considerations. UC programs are designed using a small set of constructs that include reduction, parallel assignment and fixed-point computation. The language also provides a map section that may optionally be used by a programmer to specify data mappings for the program. This paper describes the UC constructs and their current implementation on the Connection Machine. It also presents measurements of the compiler for simple benchmarks.

## 1 Introduction

The cost of developing (parallel) programs may be divided into two components: software life-cycle costs and program execution costs. The former is determined by the effort required to design, code, verify, test and maintain a program as well as the indirect costs of porting a program among different machines. Execution costs are determined entirely by the resources (memory, cpu cycles, elapsed time ...) consumed during the execution of the program on a specific machine. Henceforth we will use the terms program and architecture to mean a parallel program and a parallel architecture respectively. Life cycle costs are lower for languages that present a simple virtual machine to a programmer, in which architectural details are abstracted. In contrast, execution costs may be optimized by exploiting the interconnection topology and other architectural features of a parallel processor to reduce communication and synchronization costs. Ignoring architectural considerations in the program typically entails a performance penalty. The challenge is to design a programming environment in which simplicity of the virtual machine may be maintained, while allowing efficiency issues to be addressed separately.

In this paper, we describe our approach to parallel program design embodied in a language called UC. The project was motivated by a desire to design a language that had the simplicity of UNITY[4] and could be implemented as efficiently as existing parallel languages. A UC program clearly separates efficiency concerns from programming issues. The programming task is concerned with describing an algorithm in an abstract, high-level language. The efficiency concerns are addressed by indicating how the program data-space is to be mapped on a specific architecture. Separating these tasks allows a programmer to initially develop a *correct* prototype rapidly and follow this by incremental refinements to improve program efficiency.

Rapid program design is facilitated in UC by the use of constructs that permit program development at a higher level of abstraction than provided by most imperative languages. These include reduction, parallel assignments, fixed-point computation and non-deterministic selection. A programmer may omit the specification of data mappings in the program, further reducing the effort expended in developing a correct prototype. The compiler uses simple heuristics to generate default data mappings. However, for some applications, the execution efficiency of a program can be improved considerably by changing the data mapping such that frequently occurring communication patterns are implemented efficiently on the specific architecture. For instance, an operation that involves accessing only local memory is executed faster than one that involves remote access. Consider a program that uses two arrays $a_{1,n}$ and $b_{m,n}$ such that the primary operation in the program manipulates $b[k][i]$ as a function of $a[i]$, for a given $k$ and all $i$. The efficiency of this program may be improved by mapping the program data such that vector $a$ is stored on the same set of processors as the $k^{th}$ row of $b$. In most languages, data mappings are modified by using language constructs that were defined to transfer data among remote processors (assignment statements involving remote data, message transmissions, ...). However, such a modification directly affects the program logic and may require that the program be tested and debugged all over again. Also, the algorithmic dependence of the data is lost in the program and must be documented separately, making the program harder to maintain.

UC provides a set of well-defined *mapping* constructs that allow a programmer to *optionally* override the default mappings generated by the compiler. The mappings allow a programmer to manipulate an array in a variety of ways including changes in its shape and size. The data mappings are specified in a separate section of the program, and do not directly alter the program logic. As such changing the data mappings does not affect the correctness of a UC program.

This paper describes the UC language and its implementation on the Connection Machine using compiler generated mappings. Issues related to programmer-specified mappings are addressed briefly. A complete description on the specification, implementation and effectiveness of data mappings may be found in [2]. The next section describes related work. Section 3 describes the UC language. Section 4 briefly discusses transparent optimizations and user-specified mappings for UC programs. Section 5 describes the current implementation. A prototype compiler for UC has been developed that translates UC programs to C*, a C-based language that was designed for the Connection Machine. The section presents measurements on some sample programs written in UC and C*.

## 2 Related Work

A large number of parallel languages and notations have been designed within the imperative paradigm. Gehani[7] contains a collection of papers describing some of these efforts. Many languages are tailored to a particular parallel architecture, as is the case with C*[16] or CM Fortran[13] for the Connection Machine[9] and Cedar Fortran for the Cedar multiprocessor[10]. Other languages have been devised for a family of architectures: Ada[18], Linda[8] and Cosmic C[17] are good examples of languages for MIMD architectures. A few have attempted to bridge the gap across significantly different architectures: for instance languages like PASM C[11] and Refined C[5] were designed for implementation on both SIMD and MIMD computers. These efforts represent a large and significant body of research that has considerably advanced the state of art of parallel programming. Their major drawback lies in the amount of effort that must be devoted to efficient partitioning of data and synchronizing access to shared data, be it through message-passing, shared memory or other methods. The UC approach to program design encourages an initial program design where the mapping is left unspecified and is generated automatically by the compiler. Subsequently, a programmer may specify the mapping to improve the program efficiency.

UC shares a few characteristics with earlier language design efforts like Actus[15], SETL[6] and C*[16]. Actus is a Pascal-like language that was designed to program both vector and array processors. Parallel operations are provided via parallel arrays and index sets. A parallel array declares one of its dimensions to be parallel to indicate that the elements along that dimension may be manipulated simultaneously. An index set is a dta type used to declare a set of integers. A variable of this type is used to specify a parallel operation on a corresponding set of array elements (note that the corresponding elements must belong to a dimension that was declared as parallel). Like Actus, UC offers an index_set data type and treats arrays as the primary source of specification for parallel operations. However, unlike Actus, a UC data declaration does not inherently restrict the extent of parallelism that may be exploited to manipulate the data object.

SETL is a very high-level language based on the mathematical properties of sets. A SETL set may have an arbitrary (though finite) number of heterogeneous elements, each of which may itself be a set. Pure SETL does not include any declarations; in particular it does not indicate how specific sets that are used in a program may be implemented by traditional data structuring techniques. The set implementations may (optionally) be specified separately by the programmer or be generated automatically by the compiler. To the best of our knowledge, SETL has not been implemented on parallel architectures.

C* and CM Fortran are the two high-level languages that may currently be used to program the Connection Machine. C* is a data parallel language derived from C++. It uses the *class* concept from C++ to define a data type called *domain*. A *domain* type is a C *struct* (or record) and a collection of functions and statements that operate on the record. A *domain* type may have many instances, with each instance mapped onto a unique data processors. From an operational viewpoint, a domain type describes the structure of the local memory of a set of data processor; and domain functions describe the statements executed by it. The specification of parallelism in a C* program is thus statically linked to the declaration of data structures and their implementation on the Connection Machine. C* was also designed to be compatible with C. As a result it tries to support all features of C including the if, goto, and switch statements. The implementation of some of these statements within the synchronous execution model of C*, leads to a complex model of program control flow. CM Fortran[13] is a CM implementation of Fortran-77 extended with the array handling features of Fortran 8x. CM Fortran also provides a set of compiler directives to allow a programmer to alter the mapping of arrays on the CM memory. In general, a UC program is more concise than an equivalent program written in CM Fortran.

## 3 Language Features

UC is a simple enhancement of C: it adds a data type – *index_set*, a operator – *reduction*, and four constructs *par*, *oneof*, *seq* and *solve* to express dependencies among statements. UC imposes a few restrictions on C. It disallows goto statements and allows C pointers to be used only to pass an array (or an array slice) as an argument to a function.

### 3.1 Index Sets

An index set is a constant data item that represents an ordered set of integers, each of which must be a constant expression. The syntax for declaring index sets is as follows:

```
index-set → index_set list ;
list      → def [, list]*
          | decl [, list]*
def       → id : id = { const .. const }
          | id : id = { const [, const]* }
          | id : id = id
decl      → id : id
```

where *const* is a constant numeric expression, and *id* is any valid C identifiers.

The following fragment declares three index sets:

```
index_set I:i = {0..N-1}, idx2:j = {4,2,9},
          K:k = idx1;
```

The declaration of an index set uses two identifiers. The first identifier (eg. *I*) refers to the index set and the second (eg. *i*) to an arbitrary element of the set. In the preceding example, index sets *I* and *K* each contain N elements, from 0 to N-1 (assume that N has previously been declared to be a constant data item), while *idx2* contain the 3 elements listed explicitly. 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 set.

## 3.2 Reduction

A reduction operator is used to perform a (commutative and associative) binary operation on a set of operands. The syntax of a reduction expression is as follows:

*red-exp* → **$***bin-op* ( *idxs sc-exp*)
*sc-exp* → ; *exp*
| [;]*exp-list*
*exp-list* → *sc-exp* [*sc-exp*]* [**others** *exp*]
*sc-exp* → **st** (*pred*) *exp*

*idxs* is a list of index sets, *pred* and *exp* are C or UC expressions, and *bin-op* denotes the operation to be performed on each instance of *exp*.

The binary operator *bin-op* used in a reduction may be any one of the following:

| Operator | Meaning | Identity Value |
|---|---|---|
| $+ | addition | 0 |
| $&& | logical and | 1 |
| $> | maximum | -INF |
| $< | minimum | INF |
| $* | multiplication | 1 |
| $\|\| | logical or | 0 |
| $^ | logical exclusive or | 0 |
| $, | value of an arbitrary operand | INF |

INF is a predefined constant. The identity value is returned when the reduction operator is applied to an empty set of operands.

If a reduction contains at most one *sc-exp* (predicate together with an expression), it is referred to as a *simple* reduction. A simple reduction is computed as follows: the predicate is evaluated for every element in the index set; the elements for which it evaluates to true are said to be *enabled.* All elements are enabled, if the *sc-exp* does not contain a predicate. The expression in the *sc-exp* is then evaluated synchronously for all enabled index elements and the *bin-op* is applied to the corresponding expressions. If *idxs* contains more than one index set, the elements are selected from the Cartesian product of the index sets named in the list.

```
index_set I:i = {0..9}, J:j = I;
int s, min, first, arb, last, a[N];
float avg;

s = $+(I; i);
avg = $+(I; i) / $+(I; 1.0);
min = $< (J; a[j]);
first = $< (I st (a[i] == min) i);
arb = $, (I st (a[i] == min) i);
last = $> (I st (a[i] == $>(J; a[j])) i);
```

Figure 1: Reduction in UC

As a simple example, consider the code fragment in figure 1. In this example, *s* computes the sum of all elements in index set *I* and *avg* computes the average value of the elements. The other reductions in this example use array elements as operands: *min* computes the minimum value in array *a*; *first* computes the index of the leftmost occurrence of *min* and *arb* uses the $, operator to return the position of any occurrence of the minimum. The predicate (*a*[*i*]==*min*) is used in the computation of *first* and *arb* to select a subset of *I* that consists of the positions of all occurrences of the minimum value in array *a*. Finally, *last* computes the position of the rightmost occurrence of the *maximum* value by using a nested reduction, where the inner reduction ($>(*J*; *j*])) computes the maximum value in the array.

A reduction with more than one *sc-exp* is evaluated by evaluating the set of simple reductions, one for each *sc-exp*, and applying the specified binary operator to the results. If an *sc-exp* contains the keyword **others**, the corresponding expression is evaluated (and included in the reduction) for all index elements that were not enabled for any of the other *sc-exp* in the reduction. For instance, the following fragment computes the sum of the absolute values of the elements in array *a*.

```
abs_sum = $+ (I st (a[i]>0) a[i]
                  others  -a[i]);
```

For reductions with multiple *sc-blocks*, the order in which the constituent simple reductions are evaluated is not specified by the language (this is for compatibility with C which does not specify the order of evaluation of operands in any expression). Further, if an index element is enabled for more than one *sc-exp*, each one of the corresponding expressions is included in the computation of the reduction.

## 3.3 Control Flow in UC

The dependencies among statements in a UC program may be expressed by the following UC constructs: *par* for parallel execution, *seq* for sequential execution, *solve* for fixed-point computation and *oneof* for non-deterministic selection. The syntax for each of the preceding constructs is identical (except for the use of different keywords) and is shown below:

*sc-stmt* $\rightarrow$ [*]*sc-op* ( *idxs* ) *stmts*
*stmts* $\rightarrow$ *stmt*
  | *sc-block* [*sc-block*]* [**others** *stmt*]
*sc-block* $\rightarrow$ **st** (*pred*) *stmt*
*sc-op* $\rightarrow$ **par** | **seq** | **solve** | **oneof**

where *idxs* is a list of index sets, *pred* is a C expression that may include UC reductions, and *stmt* is any UC or C statement excluding the goto statement. Except for the *solve* statement, if any *sc-op* is preceded by a *, the corresponding *sc-stmt* is executed iteratively as long as at least one *pred* evaluates to *true*. The keyword **par** denotes parallel execution of the constituent statements, **seq** denotes sequential execution, **solve** represents fixed-point computation and **oneof** denotes non-deterministic selection. Each of the statements is discussed in the following sections.

## 3.4 Parallel Computation

A par statement is typically used to execute a parallel assignment for a set of array elements. The set is specified by the index set together with an optional boolean expression. As a simple example of the par statement, consider the following code fragment which computes array $c_{n,n}$ as the product of $a_{n,n}$ and $b_{n,n}$. Index sets *I*, *J* and *K* in the example are assumed to consist of integers from 0 to *N*-1. The par statement causes all elements of array *c* to be computed in parallel. Each element of *c*, say *c*[*i*][*j*], is computed by using a reduction to calculate the dot product of the $i^{th}$ row of *a* and the $j^{th}$ column of *b*.

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

The preceding program implicitly specifies an O($N^3$) level of parallelism: $N^2$ instances of the assignment statement are executed simultaneously, each of which requires *N* processors to compute the reduction.

A par statement may also use a predicate to select a subset of the elements from the index set. The predicate is used to determine the subset of enabled index elements (recall that an index element is enabled if on substituting its value in the predicate, the predicate evaluates to *true*) and the statement is executed in parallel for all enabled elements. The following fragment computes the reciprocal of non-zero elements in array *a*.

```
par (I)
  st (a[i]!=0)  a[i] = 1.0/a[i];
```

Similar to a reduction, a **par** statement with a single *sc-block* (a boolean predicate together with its statement) is referred to as a simple statement. A par statement with multiple *sc-blocks* is executed as a *set* of simple par statements, each of which is executed synchronously for the enabled index elements. The following fragment sets the odd elements of *a* to 0 and others to 1.

```
par (I)
  st (i%2==1)  a[i] = 0;
  others       a[i] = 1;
```

The execution order for different *sc-blocks* of a par statement is not specified by the language. The statement should be decomposed into a sequence of simple par statements to enforce a specific order.

If a par statement contains a sequence of statements, each statement in the sequence is executed synchronously for the enabled index elements. The next example sorts integers stored in an array using ranksort[4]. For simplicity, we assume that all integers in the array are distinct. In the first statement, the rank of the $i^{th}$ element is computed by counting the number of values that are smaller than $a_i$. The second statement synchronously stores each $a_i$ in its final position to yield the sorted array.

```
par (I)
{ int rank;
  rank = $+(J st (a[j]<a[i]) 1);
  a[rank] = a[i];
}
```

If a par statement is preceded by an asterisk, the statement is executed iteratively, until no index element is enabled. We use the *par statement to compute the prefix sums of elements in an array. Given array *a*, the prefix sum for the $i^{th}$ element is given by *psum*[*i*]=*a*[0]+ ... + *a*[i]. Figure 2 is a UC program fragment that computes the prefix sums for *N* elements in *log*(*N*) iterations using *N* processors. The first par statement initializes the arrays and the second par statement computes the prefix sums iteratively. The function *power2* returns $2^i$ when invoked with parameter *i*. In the $k^{th}$ iteration of the par statement ($k \geq 0$), *a*[*i*] is incremented by *a*[*i*-$2^k$]. The predicate in the statement ensures that the $k^{th}$ iteration is executed only for $i \geq 2^k$.

```
index_set I:i =  {0..N-1};
int a[N],cnt[N];

par (I)
{ a[i]=i;
  cnt[i]=0;
}
*par (I) st (i >= power2(cnt[i]) )
 {    a[i]=a[i]+a[i-power2(cnt[i])];
      cnt[i]=cnt[i]+1;
 }
```

Figure 2: Iterative par statement

Each variable in a **par** statement may be assigned at most one value. If multiple values are assigned to a single variable, they must be identical. Consider the following illegal program which incorrectly assigns b[0] ... b[N-1] to every a[*i*].

```
index_set I:i = {0..N-1}, J:j = I;
int a[N], b[N];
par (I,J)
  a[i] = b[j];
```

If the intent of the preceding program was to assign any one of the values in array $b$ to each element in $a$, the non-determinism must be specified explicitly by using the $, operator.

If par statements are nested, an ambiguity may arise with respect to the binding of an sc-block. This problem is similar to the problem of a dangling else clause that may arise in C. The solution is the same as that adopted by C for the dangling else problem: by default, the sc-block will be bound to the innermost par statement. Braces may be used to force a different binding.

A given index set may be reused in a nested UC statement. In such situations, the inner use of the index set hides the outer one in a manner analogous to redeclaration of a C variable in an inner scope. Consider the following example:

```
index_set I:i = {0..9};
int a[10];
par (I)
  st (i%2==0)  a[i] = $+(I; i);
```

The even numbered elements of array $a$ are assigned the sum of the first 9 integers . The predicate restricting the index set in the *par* statement is not visible within the reduction due to the reuse of index set *I*.

## 3.5 Sequential Execution

The **seq** operator allows iterative execution of a statement for each element in an index set, where the elements are chosen in the order that they appear in the definition of the index set. The seq statement may also be interpreted as a counting loop where the index set together with the predicate (if any) determines the successive values of the loop control variable. The code fragment in figure 3 computes the partial sums of figure 2 by using a seq statement nested within a par. The par statement first initializes array $a$ and then executes the seq statement, in parallel, for every index element in *I*. For a given $i$, the seq statement is executed sequentially for each $j$ in $0 \leq j \leq \lfloor log(i) \rfloor +1$; furthermore, the addition operation is executed simultaneously for every (enabled) element.

Figure 4 describes a complete UC program that computes the all pairs shortest path using an algorithm with $O(N^2)$ parallelism. The first par statement initializes array $d$ such that $d[i][i]$ is 0 and for $i \neq j$, $d[i][j]$ is assigned an integer value randomly distributed in the range 1 to N. The seq statement causes the nested par statement to be executed N times, for increasing values of $k$ in the range 0..N-1. For a given $k$, the par statement simultaneously updates $d[i][j]$ for every $i$ and $j$. For a given $i$ and $j$, $d[i][j]$ is updated if the distance from $i$ to $j$ via $k$ is less than the current distance from $i$ to $j$. It follows, that when $k$=N-1, $d[i][j]$ must contain the shortest path in the graph from node $i$ to node $j$.

```
index_set I:i = {0..N-1}, J:j = {0..LOGN-1};
int a[N];

par (I)
{  a[i]=i;
   seq (J) st (i-power2(j) >= 0)
     a[i] = a[i]+a[i-power2(j)];
}
```

Figure 3: Partial sums with **seq**

```
#define N 32
#define min(x,y) ((x)<(y)?(x):(y))
index_set I:i = {0..N-1}, J:j = I, K:k = I;
int d[N][N];

par (I, J) st (i==j)
      d[i][j] = 0;
   others
      d[i][j] = rand()%N + 1;

seq (K)
  par (I,J)
    d[i][j] = min(d[i][k]+d[k][j], d[i][j]);
```

Figure 4: Shortest path: $O(N^2)$ Parallelism

The preceding program executes the par statement N times even though all shortest paths may have been found for a smaller value of $k$. For instance, in a degenerate case, the original distance matrix itself may contain the shortest path for all node pairs. For such configurations, a *par statement may be used such that execution is continued only if the distance between some node pair is updated in a particular iteration.

Figure 5 computes the all pair shortest path using an algorithm with $O(N^3)$ parallelism. The initialization code has been omitted as it is identical tothat inthe preceding example. In this algorithm, the seq statement causes the nested par statement to be executed $log(N)$ times. In each execution, the par statement uses a reduction to compute the current shortest path between every pair of nodes. For a given $i$ and $j$, the shortest path is computed by first calculating the distance from $i$ to $j$ through each one of the $k$ intermediate nodes, and then taking the minimum of all distances. If the graph has $N$ nodes, the preceding algorithm must necessarily compute all shortest paths in at most $log(N)$ iterations.

## 3.6 Fixed-Point Computation

A number of scientific and other problems may be solved by finding the solution to a set of equations. UC provides a primitive called **solve** which may be used to write programs that essentially solve a set of *proper* equations[4]. Like a

```
index_set I:i = {0..N-1}, J:j = I, K:k = I;
index_set L:l = {0..LOGN-1};
int d[N][N]; /* initialized appropriately */
seq (L)
  par (I,J)
    d[i][j] = $<(K; d[i][k]+d[k][j]);
```

Figure 5: Shortest path: O($N^3$) Parallelism

*proper* set of equations, a *proper* set of assignments must satisfy the following properties:

- the value of a variable (by a variable we mean a primitive data object) may be modified by at most one assignment;
- for variables whose value is both modified and accessed in the set of statements, there must exist an ordering such that the statement that modifies the value appears before any statement that accesses the values.

The wavefront problem[1] illustrates the simplifications provided by the **solve** primitive. The problem is to build a matrix with the following properties:

$a[0,j]=1,\ a[i,0]=1,\ \ \forall i,j$ and
$a[i,j]=\ a[i\text{-}1,j]+a[i\text{-}1,j\text{-}1]+a[i,j\text{-}1];\ \ \forall i,j>0$

The specification clearly satisfies the requirements for a set of equations to be *proper*. The above specification is directly transformed into the following UC program using solve:

```
index_set I:i = {0..N-1}, J:j = I, K:k = I;
int a[N][N];
solve (I, J)
  a[i][j] = (i==0||j==0) ? 1
            : a[i-1][j]+a[i-1][j-1]+a[i][j-1];
```

The keyword solve indicates that the specified set of assignment statements constitute a proper set. It is the compiler's responsibility to execute the statements in the order of their dependency. The solve construct is implemented via source level program transformations. If the array references within a solve statement only use constants and index elements, then the statement can be translated into an equivalent UC program that use **seq** and **par** statements to execute the assignments in the order of their dependencies[14]. In the more general (though less efficient) method, the solve block is implemented as a *par statement. Each variable whose l-value is referenced in the solve statement, is assigned an 'impossible' value outside the *par. The assignment statements are then tested repeatedly within the *par statement, and a particular assignment is executed only if it has not previously been executed and if every variable on its right hand side has a well-defined value. The *par terminates if no variable is modified in a particular iteration.

Like the other UC constructs, solve may also be prefixed by an asterisk to specify repeated execution. However, the statements in a *solve block are not required to be 'single assignment' statements. A *solve statement is executed repeatedly until the computation reaches a fixed-point. In other words, execution halts only when continued execution of the statements within the *solve will not modify the value of any variable referenced within the statement. This construct is useful for writing programs without explicitly specifying the termination condition. Using *solve, the program to compute the all-pairs shortest path is as follows:

```
index_set I:i = {0..N}, J:j = I, K:k = I;
int dist[N][N];  /* initialized appropriately */
*solve (I,J)
  dist[i][j] = $<(K; dist[i][k]+dist[k][j]);
```

The fixed-point for a *solve statement may be detected by translating it into an equivalent *par. Every variable whose l-value is referenced in the statement is copied into a temporary variable prior to being modified. The temporary variables are used to set up the predicate such that the *par statement terminates only when no variable is modified in a specific iteration.

A *solve statement may directly be refined by the programmer as a *par statement. However, when using *par, the fixed-point must be explicitly coded in the predicate and any required saving of intermediate states be done by the programmer. As this is done implicitly in the *solve construct, the latter yields concise programs that may be developed rapidly. Of course, the use of *par is more efficient than *solve as the programmer need not save redundant intermediate states that might be saved by the compiler to detect the fixed-point.

## 3.7 Non-deterministic Choice

The **oneof** construct is used to execute a statement for a subset of the enabled index elements. Consider a **oneof** statement with multiple *sc-blocks* (recall that a *sc-block* consists of a predicate and statement). An sc-block with a predicate $b_i$ is said to be *enabled*, if at least one index element is enabled for $b_i$. If a oneof statement contains at most one enabled *sc-block*, its semantics are exactly the semantics of a corresponding par statement, i.e. the statement is executed for the set of enabled index elements (if any). However, in a oneof statement with multiple enabled *sc-blocks*, any (and only) one of the enabled *sc-blocks* is executed. If a oneof statement is preceded by a *, the statement is executed iteratively as long as at least one *sc-block* is enabled. The **oneof** statement does not assume a fair scheduler. In particular, even if some *sc-block* is *enabled* infinitely often, it may never be executed.

The following example illustrates the use of the *oneof statement in sorting an array of integers using the odd-even transposition sort. Assume that *swap* is defined to appropriately exchange the values of its parameters.

```
int x[N]; /* initialized appropriately */
index_set I:i = {0..N-2};
*oneof (I)
```

```
st (i%2==0 && x[i]>x[i+1])  swap(x[i], x[i+1]);
st (i%2!=0 && x[i]>x[i+1])  swap(x[i], x[i+1]);
```

In the preceding fragment, the first (second) sc-block is enabled if any of the even (odd) elements in the array is not sorted relative to its right neighbor. Execution of a sc-block causes the corresponding odd or even numbered elements that are out of order to be swapped with their neighbor. The statement terminates only when no sc-block is enabled, in other words, when the array is sorted.

## 4 Optimizations

UC optimizations are classified into three groups: code optimizations, processor optimizations and communication cost optimizations. Code optimizations attempt to reduce the computation time of a program for a given data and processor configuration. These optimizations include the standard 'peep-hole' compiler optimizations like common subexpression detection, constant folding, and replacing remote read operations by remote write operations, where possible. Processor optimizations attempt to deduce the optimal number of CM *virtual* processors needed to execute a UC program[12]. This optimization may improve the execution time of a program either by reducing the number of virtual processors needed to execute a program or by reducing the number of activations for different processor sets. For instance, consider the following fragment that computes the number of occurrences of each digit from 0..9 in an array *samples*:

```
int samples[N];
int count[9];
index_set I:i = {0..N-1}, J:j = {0..9};

par (J)
  count[j] = $+(I st (samples[i]==j)  1)
```

A simplistic implementation of the preceding fragment would use 10*N processors: 10 processors to simultaneously execute the par statement for each $j$ and $N$ processors for each reduction to compute a given *count*[$j$]. However, the predicate used in the reduction may be analyzed at compile-time to deduce that each value in array *samples* can affect the value of at most one element of array *count*. Tis implies that the reduction may be performed simultaneously for all values of $j$ and may be implemented with N processors.

Communication cost optimizations improve execution efficiency by changing data mappings of a program such that remote references of data are either minimized or executed efficiently. Of the three types of optimizations discussed in this paper, this optimization has the most significant impact on the execution efficiency of a program.

Given a UC program, the compiler generates default mappings for the program data. Arrays are the primary data structures in a UC program. Arrays that are referenced within parallel UC constructs are automatically mapped by the compiler on the Connection Machine memory so that they may be manipulated in parallel. The default mapping generated by the compiler stores corresponding elements of conforming arrays on the local memory of a common processor. Thus, if a program uses two arrays $a_{1,n}$ and $b_{1,n}$, the arrays will be stored by the compiler such that, for every $i$, the $i^{th}$ elements of the two arrays are stored on a common memory. This ensures that statements like $a[i]=b[i]$ will be executed as local assignments. However, this may not be the most efficient mapping for some application. Consider, for instance, the execution of a statement like $a[i]=b[i+1]$. For this operations, communication costs will be minimized if the two arrays are mapped such that the $i^{th}$ element of $a$ is on the same processor as the $(i+1)^{th}$ element of array $b$. Many other examples exist, where the default mappings will be sub-optimal.

UC provides a section called the **map** section which is used to override the default mappings generated by the compiler. The mappings do not directly alter the logic of the program. A programmer may initially develop a program that ignores the mapping concerns, and subsequently modify the mappings to improve the program efficiency. As these modifications do not affect program correctness, and only require a declarative specification, a number of alternative mappings may be tested quickly.

Three basic classes of mappings are currently defined for UC: *permute*, *fold* and *copy*. Permute mappings are used to reorder the elements of an array, say $a1$, relative to another array $a2$ so that corresponding elements of $a1$ and $a2$ that are accessed in a single statement are stored locally. Arrays $a1$ and $a2$ need not have the same shape or size. Fold mappings allow part of an array to be folded over so that corresponding elements of the *same* array that are accessed together can be stored locally. Finally copy mappings allow an array to be replicated along an extra dimension to reduce the need for broadcasts. The following fragment illustrates the use of a permute mapping to alter the default mapping for arrays $a$ and $b$ considered in the preceding example.

```
map (I)
{ permute (I)
     b[i+1] :- a[i];
}
```

The syntax used for specification of the mappings is similar to that of other UC constructs. The keyword permute indicates the type of mapping and arrays $b$ and $a$ respectively refer to the source and target data structures for the mapping. In this example, the map section modifies the mapping of $b$ with respect to $a$ to ensure that the $(i+1)^{th}$ element of $b$ will be mapped on the same processor as the $i^{th}$ element of $a$. Given the map section for a program, the UC optimizer executes a source-to-source transformation on the program so that index expressions are updated to reflect the modified data allocation of array $b$. For the preceding mapping, the source transformations will subtract 1 from every subscript expression for array $b$. Thus an assignment statement like $a[i]=a[i]+b[i+1]$ is transformed into $a[i]=a[i]+b[i-1+1]$ and simplified to $a[i]=a[i]+b[i]$ which can then be executed as a local operation. A complete description of the mappings together with experimental measurements to demonstrate their utility in improving the execution time of programs may be found in [2]. The execution efficiency of some programs was improved by a factor if 10, simply by specifying an efficient mapping for the program data.

## 5 Implementation

A prototype implementation of UC is operational on the Connection Machine. Other than the solve primitive, all UC constructs described in this paper are supported. The UC compiler generates C* target code which can then be compiled and executed using the C* compiler. Currently, the UC compiler is being rewritten to directly generate C-Paris (Connection Machine assembly language) code.

A restricted set of experiments were executed to measure the efficiency of the UC implementation with respect to C*. Both numerical computations and graph algorithms were used as benchmarks and the results were similar. Figures 6 and 7 present the measurements for the shortest path problem using $O(N^2)$ and $O(N^3)$ algorithms for both UC and C*. The measurements show that the performance of UC programs matches that of C*. The UC programs for these problems were described in section 3.5; the corresponding C* programs are included in the Appendix. In particular, note that a C* program must explicitly and statically specify the maximum extent of parallelism that exists in the program. This implies that in going from the O($N^2$) algorithm to the O($N^3$) algorithm, the programmer must explicitly declare a 3-dimensional array to permit parallel evaluation of the shortest path via all intermediate nodes. The automatic store management performed by the UC compiler allows this to be omitted in the UC program. As a result, the two C* programs differ significantly from each other, but the corresponding UC programs are similar.

We also present the measurements for the execution of an iterative algorithm to compute the shortest distance of every point in a grid to a specific destination, referred to as the goal (G). Each cell in the grid is assumed to be connected to its four neighbors in the East, West, North and South directions by an edge whose weight is 1. (The cells on the boundary will have less than four neighbors). Every cell is initialized to be at a distance 0 from G. The shortest path is computed by using the following iterative algorithm: in each iteration, every cell (except G) inspects the distance of each of its four neighbors to G, and selects the minimum distance. It adds 1 to this value and stores the result as its current shortest path to the goal. This computation may be expressed concisely with the *par construct of UC. An additional complexity was introduced in the problem by adding 'obstacles'. If a certain cell is part of an obstacle, it is assumed to be disconnected from its neighbors. If such a cell was initially on some shortest path, the path must now be recomputed to bypass the obstacle. The obstacles may also be moved dynamically in a random manner to simulate a dynamic graph. The UC program for this problem with a stationary obstacle is presented in figure 11. The execution time for the program for both sequential and parallel executions is presented in figure 8. The C program was executed on a SUN 4 workstation (which is also used as the front end for the CM) and the parallel code on a 16K CM. The figure also includes measurements for an optimized sequential program, which was optimized by using the -o option in the C compiler. Experiments are in progress to study the performance of UC programs for CFD applications as well as numerical computations involving SVD and Jacobi diagonalization[3].

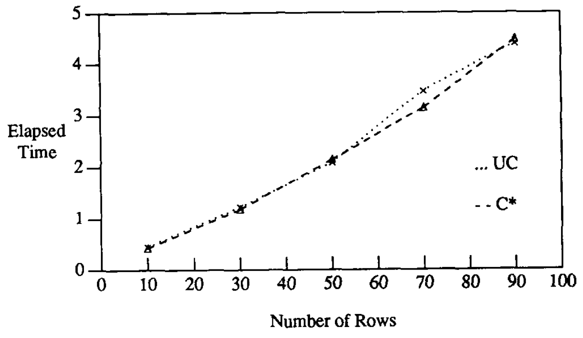

Figure 6: Shortest Path $O(N^2)$ Parallelism

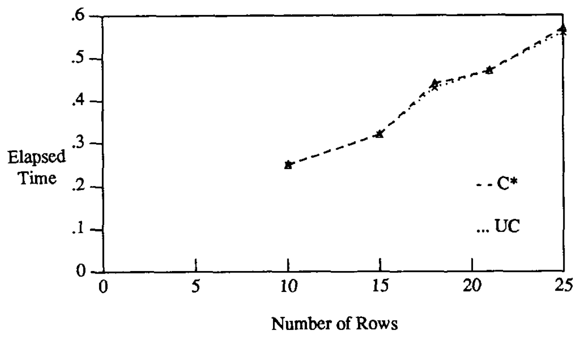

Figure 7: Shortest Path $O(N^3)$ Parallelism

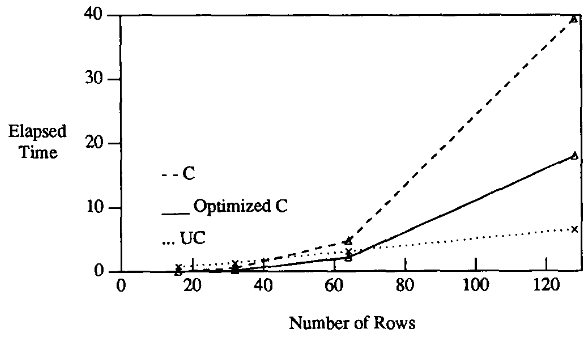

Figure 8: Shortest Path with obstacle

**Acknowledgements**

We are grateful to Jay Misra and Indranil Chakravarty for their helpful comments on an earlier version of the language. The developmental work was conducted on the Connection Machines at UCLA, which is funded by the NSF Biological Facilities Award (BBS 87 14206), and at ISI. We are grateful to the Cognitive Science Program at UCLA and to Danny Cohen at ISI for their support.

## References

[1] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. In *Workshop on Graph Reduction*, September 1986.

[2] R. Bagrodia and S. Mathur. Efficient implementation of high-level parallel programs. Technical Report CSD-900022, Computer Science Dept, UCLA, Los Angeles, CA 90024, August 1990.

[3] I. Chakravarty, 1990. Personal Communications.

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

[5] H. Dietz and D. Klappholz. Refined C: A sequential language for parallel programming. In *Proceedings of the International Conference on Parallel Processing*, pages 442–449, August 1985.

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

[7] N. Gehani and A.D. McGettrick. *Concurrent programming.* Addison-Wesley, Reading, Massachusetts, 1988.

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

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

[10] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. Parallel supercomputing today and the cedar approach. *Science*, pages 967–974, February 28 1986.

[11] J.T. Kuehn and H.J. Siegel. Extensions to the c programming language for simd/mimd parallelism. In *Proceedings of the International Conference on Parallel Processing*, pages 232–235, August 1985.

[12] Edmund Kwan. The UC implementation on the Connection Machine. Comprehensive Report, July 1990.

[13] 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.

[14] S. Mathur. Source to source transformations for UC. Master's Thesis, Computer Science Department, UCLA, August 1990.

[15] R Perrott. A language for array and vector processors. *ACM TOPLAS*, 1(2), October 1979.

[16] 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.

[17] C.L. Seitz, J. Seizovic, and Wen-King Su. The C programmer's abbreviated guide to multicomputer programming. Technical Report Caltech-CS-TR-88-1, Dept. of Computer Sciences, California Institute of Technology, Los Angeles., January 1988.

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

## A Programs

```
#define N 32

domain PATH {
    int i, j, k, len;
}path[N][N];

void PATH :: init() {
    int offset = (this- &path [0][0]);
    i = offset / N;
    j = offset % N;
    len = rand () % (N*N);
}

void main() {
 [domain PATH].{
   int k;
   int()
   for (k=0; k<N; k++)
    len <?=path[i][k].len + path[k][j].len;
    }
}
```

Figure 9: Shortest path in C* : O($N^2$) Parallelism

```
#define N 32
#define LOGN 5

domain PATH {
    int i, j, k, len;
}path[N][N];

domain XMED {
    int i, j, k;
}xmed[N][N][N];

void PATH :: init() {
    int offset = (this- &path[0][0]);
    i = offset/N;
    j = offset%N;
    len = rand()%(N*N);
}

void XMED :: init() {
    int offset = (this- &xmed[0][0][0]);
    k = (offset/N)/N;
    j = (offset/N)%N;
    i = offset%N;
}

void main() {
 [domain PATH].{ init(); }
 [domain XMED].{
   int cnt;
   init()
   for (cnt=0; cnt<N; cnt++)
    path[i][j].len <?= (path[i][k].len
+ path[k][j].len);
    }
}
```

Figure 10: Shortest path in C* : O($N^3$) Parallelism

```
#define N       32
#define WALL      INF
#define GOAL      1
#define COST(X,Y) ((X)<0||(Y)<0||(X)>=N
             ||(Y)>=N?INF:a[X][Y])
#define MIN(X,Y)  ((X)<(Y)?(X):(Y))
#define    ABS(X)       ((X)<0?(-(X)):(X))

int a[N][N];
index_set I:i = {0..N-1}, J:j = I;

main() {
   init();
   *par (I, J)
     st (a[i][j]!=GOAL && a[i][j]!=WALL
                          && a[i][j]!=min(i,j)+1)
       a[i][j] = min(i,j) + 1;
}

int
min(x,y)
int x, y;
{
   return MIN(MIN(COST(x,y+1), COST(x,y-1)),
          MIN(COST(x+1,y), COST(x-1,y)));
}

init() {
   par (I, J)
     st (i+j==N-1 && ABS(i-N/2)<=N/4)
            a[i][j] = WALL;
     others    a[i][j] = GOAL+1;
   a[0][0] = GOAL;
}
```

Figure 11: UC Program for shortest path with obstacles