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

EFFICIENT IMPLEMENTATION OF HIGH-LEVEL PARALLEL PROGRAMS

R. L. Bagrodia
S. Mathur

August 1990
CSD-900022

Efficient Implementation of High-Level Parallel Programs[*]

Rajive Bagrodia and Sharad Mathur
3531F Boelter Hall
Computer Science Department
UCLA
Los Angeles, CA 90024
(213) 825-0956
rajive@cs.ucla.edu

August 10, 1990

Abstract

The efficiency of a parallel program is related to the implementation of its data structures on the distributed (or shared) memory of a specific architecture. This paper describes a declarative approach that may be used to modify the mapping of the program data on a specific architecture. The ideas are developed in the context of a new language called UC and its implementation on the Connnection Machine. The paper also contains measurements on sample programs to illustrate the effectiveness of data mappings in improving the execution efficiency of a program.

[*]This research was supported by a grant from the Schlumberger Foundation

1 Introduction

In designing parallel languages, software engineering concerns dictate that the language abstract out architectural details and present a simple virtual machine to the programmer. However, experience with programming parallel architectures has indicated that optimal performance can be extracted only by exploiting the communication topology of the architecture such that communication and synchronization costs are minimized. Abstracting out architectural details in the program generally entails a performance penalty.

Consider the Connection Machine (CM) as an example. The main memory in the CM is divided into two parts: the front-end memory and the data processor memory. The former is associated with the 'front-end' machine that controls the data processors; this memory may only be operated upon sequentially. The latter consists of the local memory associated with each of the 64K data processors; data stored on this memory may be executed in parallel. Although the CM has (upto) 64K physical processors, a program may assume an arbitrary number of *virtual* processors (VP) which are interleaved transparently on the physical processors. The number of VPs that are mapped to each data processor is referred to as the VP ratio for the program.

In general, an access to the local memory of a data processor is faster than that of a remote processor. The CM provides two mechanisms to allow remote memory access[12]: a general purpose *router* which can be used by any processor to access the local memory of any other processor, and a special-purpose 2-dimensional rectangular grid called the NEWS grid which allows relatively fast communication among the processors that lie along the grid. Ignoring the difference in the cost of accessing different memories may have a significant impact on execution efficiency. For instance, if the operands in a parallel summation lie along a NEWS grid line, the summation can be computed considerably faster than if they were distributed randomly among the processors (actual measurements of the speedup that is realized is included in a subsequent section). Furthermore, if a set of elements is manipulated sequentially, storing them on the data processors rather than the front-end memoty will severely degrade performance.

In this paper, we describe our approach to parallel program design embodied in a language called UC. UC differs from most existing languages in that 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. The approach is illustrated by the following example.

Arrays are the primary data structure used by a UC program. The program is not required to specify how the arrays are to be mapped on the CM memories. The compiler uses simple heuristics to generate default mappings for the arrays in a given program. For instance, if an array is never used within a parallel construct, it is automatically stored on the front-end; otherwise it is mapped on the data processor. If an array is stored on the data processors, each element of the array will be stored on a unique (virtual) processor. Arrays that have the same dimensionality are mapped such that corresponding elements of different arrays lie on the same processor and successive elements of an array are mapped to neighboring processors. Thus if a program uses two arrays $a_{n,n}$ and $b_{n,n}$, the $(i,j)^{th}$ elements of each array are automatically mapped on the same data processor.

The default mappings may generate sub-optimal code for a specific application. Consider an application where the majority of the operations access $a[i][j]$ and $b[i][j+1]$ simultaneously. The execution efficiency may be improved by using a data mapping that will store $a[i][j]$ and $b[i][j+1]$ on the same processor. In most existing languages, the only way to modify the data mapping, is by executing an explict assignment statement to appropriately shift the data elements of array b. However, this modification may have a ripple effect and cause extensive changes in the program, forcing it to be tested and debugged all over again. In addition, the program loses the algorithmic dependence of $a[i][j]$ on $b[i][j+1]$.

In contrast, 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 primary motivation behind these mappings is to optimize the communication time consumed by the pattern of remote accesses that are used most frequently in the program. The default mappings may also be changed to reduce the VP ratio as also to more evenly spread the computation load. The data mappings are specified in a separate section of the program, and do not require direct modification to the program logic. As such changing the data mappings does not affect the correctness of a UC program.

The next section gives a brief description of the UC language. Section 3 presents the data mappings and section 4 describes how they are implemented. Section 5 provides experimental results to demonstrate their effectiveness in improving the execution time of various programs. Section 6 describes some restrictions and future directions. The paper concludes with a brief discussion of related work.

2 The UC Programming Language

UC is a simple extension of C. This section gives a brief description of a subset of the language. The description is aimed at giving a flavor of the language and is not intended to be a precise definition of its semantics. For a complete description, the reader is referred to [2, 8].

UC enhances C with a data-type called *index-set* and some new operators to specify parallel execution. An index-set represents an ordered set of integers. The most commonly used UC operators are *reduction* and *parallel assignment.* Reduction is used to perform an associative, commutative binary operation on a set of elements, and parallel assignment is used to simultaneously modify a set of elements. The set of elements used by the parallel operators is selected from a C array by using an *index-set* together with a boolean predicate.

Consider the UC program in figure 1. The program computes matrix $c_{n,n}$ as the product of two matrices $a_{n,n}$ and $b_{n,n}$. For brevity, input-output issues are ignored. The program declarations include the standard C declarations for matrices a, b and c(line 1). In addition, three index-sets I, J and K are declared (line 2). Each index-set consists of N integers from 0 to N-1. The declaration of an index-set uses two identifiers, the first one (example I) refers to the index-set itself and the second (example i) to an arbitrary element of the set.

```
1       int a[N][N], b[N][N], c[N][N];
2       index-set I:i={0..N-1}, J:j=I, K:k=I;

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

Figure 1: Matrix multiplication in UC

The program consists of a single UC parallel assignment. The UC keyword **par** (line 3) specifies parallel execution of the assignment statement in line 4. The index-set(s) contained in the **par** statement determine the number of instances of the assignment statement that are executed in parallel. In this case, N^2 instances of the assignment will be executed, one for each of the N^2 elements in the product set $I*J$. Now consider the assignment statement itself. For a given (i, j), the value of $c[i][j]$ is computed as the dot product of two vectors: the i^{th} row of a and the j^{th} column of b. The dot product is programmed in UC as a reduction (line 4). The reduction specifies the addition of N expressions, one for each value of index-element k, where the k^{th} expression is $a[i][k]*b[k][j]$.

The next example (figure 2) computes the trace of array a(the sum of elements on its main diagonal). The declarations are similar to the previous example and have been omitted. The body is a single reduction which computes the sum of the diagonal elements. This example illustrates how a boolean expression may be used to select a subset of the elements from an index-set. The predicate $(i{=}{=}j)$ selects only those elements from the index-sets that correspond to the subscripts of the diagonal elements of array a. The keyword **st** in the reduction separates the index-sets from the boolean expression. (Note that this reduction may be expressed more concisely as: $\$+(I;\ a[i][i])$). A boolean expression may also be used in a par statement to restrict the assignment operation to a subset of the array elements.

```
par(I,J)
   trace = $+(I,J st(i==j) a[i][j]);
```

Figure 2: Trace Computation in UC

The restriction of a single assignment statement in the body of a parallel statement causes programs to become verbose. For example, the program in Figure 3 initializes a and then computes the prefix partial sums; the keyword **par** and the index-set have to be repeated. To abbreviate such programs, we allow the extent of parallelism for a *sequence* of statements to be specified only once as shown in figure 4.

```
par(I) a[i] = i;
par(I) psum[i] = $+(J st (j<=i) a[i]);
```

Figure 3: Partial sums in UC

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

Figure 4: Partial sums in UC

The last example (figure 5) illustrates repeated execution of a par statement. The asterisk preceding the keyword par indicates that the statement is executed repeatedly until every instance of the predicate associated with the statement evaluates to false. In this example, the assignment statement for the i^{th} row is executed repeatedly until the first element of the row is negative.

```
*par(I) st (a[i][0]>=0)
    a[i][0] -= $+(J st (j!=0) a[i][j]);
```

Figure 5: Iterative Par statement

If the keyword **par** of a UC statement is replaced by **seq**, the corresponding statement (or statement block) is executed sequentially for each element in the index-set. The elements are selected in the order of their occurence in the original specification for the corresponding index-set. The seq statement may be thought of as an abbreviated specification of a counting loop.

3 Mappings

A UC program need not specify how an array is to be mapped on the CM memory. However, even for a relatively simple program like matrix multiplication, a modified data mapping can improve the execution efficiency of a program. In this section we describe the UC constructs to modify the default mappings generated by the compiler.

UC currently defines four types of mappings: **permute**, **fold**, **copy** and **reduce**. A **permute** mapping is used to reorder the elements of an array so that corresponding elements of different arrays that are accessed in a single assignment can be stored locally; **fold** allows part of an array to be folded over so that corresponding elements of the *same* array that are accessed together can be stored locally. A **copy** allows an array to be replicated along an extra dimension to reduce the need for broadcasts. In contrast, a **reduce** mapping allows an array to be collapsed along one dimension to reduce the VP ratio of the program as well as to make some data references local. This mapping is particularly useful for arrays that have parallel operations defined only along one dimension. Each type of mapping is explained in detail subsequently in this section.

We use the matrix multiplication program of figure 1 to illustrate mappings in UC. Figure 6 contains the program and its map section. The multiplication algorithm uses O(N^3) processors: the reduction in line 19 uses N processors to compute each element of array C and the par statement in line 18 computes the N^2 elements of array C simultaneously. However, the default mappings generated for the arrays stores the corresponding elements of arrays A, B and C on the local memory of N^2 processors. This implies that the program includes a significant number of remote data accesses. A copy mapping may be used to expand both arrays A and B onto N^3 processors such that each remote access is replaced by a functionally equivalent local access.

The map section comes at the end of a UC program and begins with the keyword *map*. The map section for this program begins at line 22. This section consists of an optional declaration section and a list of mappings. The keyword *virtual* is used to declare variables in the map section. These variables are 'typeless' and do not occupy physical memory; their only purpose is to define a template that is used to specify a subsequent mapping. The example declares a 3-dimensional array in line 24. Lines 26 and 27 respectively specify the mapping for arrays A and B. A mapping specification has five parts: the *type of mapping*,

```
1       #define M 39
2       #define N 39
3
4       index_set I:i = {0..M-1};
5       index_set J:j = {0..N-1};
6       index_set K:k = {0..M-1};
7
8       int A[M][N], B[N][M], C[M][M];
9
10      init()
11      {
12       /* Some initialization of A and B */
13      }
14
15      main()
16      {
17          init();
18          par (I,K)
19          C[i][k] = $+(J; A[i][j]*B[j][k]);
20      }
21
22      map
23      {
24          virtual M1[M][N][M];
25
26          copy(I,J) A[i][j] :- M1[i][j][k];
27          copy(K,J) B[j][k] :- M1[i][j][k];
28      }
```

Figure 6: A Matrix multiplication example

which may be one of permute, fold, copy or reduce, the *source* which refers to the array whose default mapping is being modified, the *target* which refers to the array 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 specifies how the source is to be transformed. Each array in the program may be the source array in at most one mapping.

Data mappings for UC arrays may be specified to be either absolute or relative. In absolute mappings, the target array is a virtual data structure declared in the map section (as in the example in figure 6). Relative mappings use a program array as the target. Examples of this type of mapping are included subsequently.

The mapping defined in line 26 specifies a copy mapping with the array A as the source and the virtual array $M1$ as the target. The symbol :- is used to separate the source and target arrays. The mapping function specifies that the array A be transformed into a 3-dimensional object by copying it M times along a third dimension. This ensures that each of the N^3 processors, say (i,j,k) can access $A[i]j]$ locally as $A[i][j][k]$. Similarly, the mapping for array B (line 27) allows each processor to access the corresponding element of B in its local memory. The two mappings together ensure that the multiplication in line 19 of the original program becomes a local computation for each processor in the transformed program. If matrix multiplication is used frequently in some program than this mapping may yield a significant overall benefit. The speedup achieved as a result of using this mapping is described in a subsequent section.

For completeness, we present the BNF description for the map section:

map_section	→	**map** { [*virt_decl*]* [*map_stmt*]* }
virt_decl	→	**virtual** *arr_decl* [,*arr_decl*] ; | *idx_decl* ;
map_stmt	→	*map_op* (*idxs*) *int_stmts*
int_stmts	→	*bas_map_stmt* | { [*bas_map_stmt*]* }
bas_map_stmt	→	*map_fn_spec* ; | **st** (*expr*) *map_fn_spec* ;
map_fn_spec	→	*id* [[*id*]]* :- *id* [[*expr*]]*
map_op	→	**permute** | **fold** | **copy** | **reduce**

The non-terminals *arr_decl* and *idx_decl* respectively refer to declaration statements for array objects or index-sets.

A **permute** mapping refers to any one-to-one onto mapping of the source data structure on the target. Although a permute mapping may change the shape of the source array, the number of elements in the source array remains unchanged. Figure 7 illustrates some commonly used permute mappings together with their UC specification.

Permute mappings may also be used to map dissimlar arrays relative to each other. The following fragment maps a vector $a_{1,n}$ with respect to a matrix $b_{m,m}$ $(m > n)$ such that the elements of a are left flushed along the third row of array b. Other relative mappings may be defined similarly.

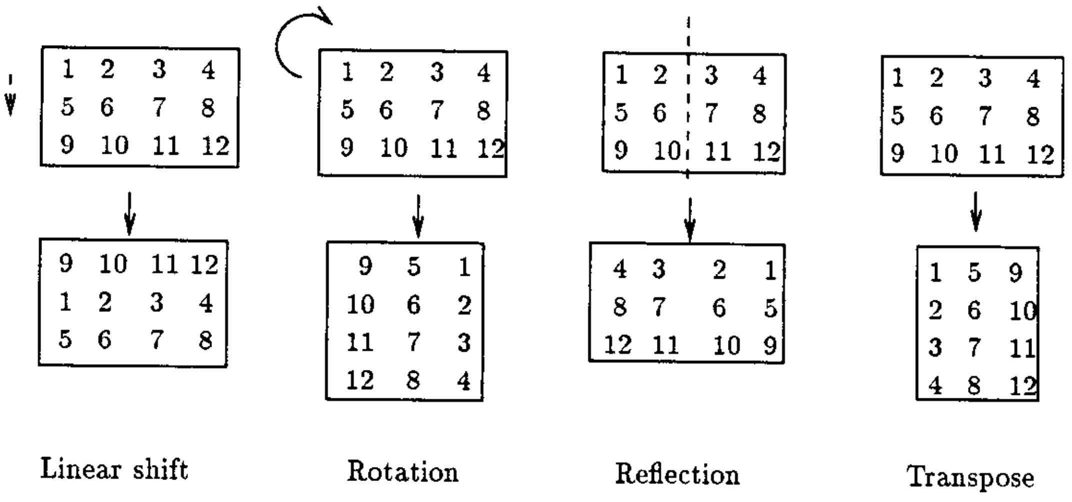
```
permute(I)
   a[i] :- b[2][i];
```

A **fold** mapping moves a subset of the array elements such that some locations in the original array will contain multiple data items and others will be 'empty'. The most commonly used fold mapping involves folding the array along some axis. Figure 8 displays a fold mapping of this type, where the array is folded along a vertical axis. Note that a fold mapping typically includes a predicate in order to distinguish between those elements that are to be moved and those that remain in their original location in the source.

A **copy** mapping specifically increases the dimensionality of the source by one and replicates it along this new dimension. An example of this is given in figure 9. Expansion mappings are useful where increasing the dimensionality of the data structure can make references local without increasing the order of complexity of the algorithm.

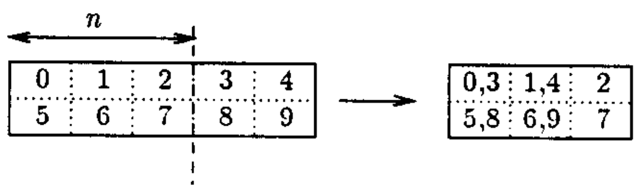
A **reduce** mapping on the other hand reduces the dimensionality of the source. It does this by collapsing one dimension and storing all the collapsed elements in a struct. A generic example of this can be seen in figure 10. In addition to making references local, these mappings reduce the size of a data structure and any extra level of parallelism associated with the contracted dimension. This decreases the number of virtual processors required for a program, but may increase the computation time for operations along its collapsed dimension.

A series of the above kinds of mappings can be composed together to define more complex mapping functions. For instance, the mapping in figure 11 uses a



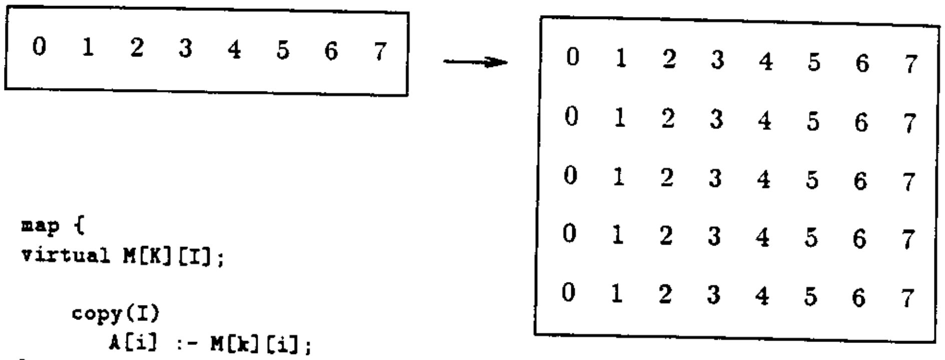
```
permute(I,J)                permute(I,J)               permute(I,J)               permute(I,J)
  A[i][j]:-A[(i+1)%M][j]      A[i][j]:-A[j][M-i-1]       A[i][j]:-A[i][N-j-1]       A[i][j]:-A[j][i]
```

Figure 7: Permute mappings



```
/* n=3 in this example */
fold(I,J)
    st (j >= n) /* this is the mapping predicate */
        A[i][j] :- M[i][j-n];
```

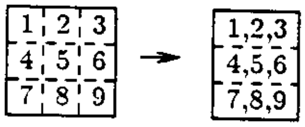
Figure 8: A fold mapping



```
map {
virtual M[K][I];

    copy(I)
       A[i] :- M[k][i];
}
```

Figure 9: A copy mapping



```
map {
virtual M[K];

    reduce(I)
       A[k][i] :- M[k];
}
```

Figure 10: A reduce mapping

```
/* A composition mapping */
map {
    virtual M1[M][N];

    copy (I)
        A[i] :- M1[i][j];
    permute (I,J)
        M1[i][j] :- M1[i-2][j];
}
```

Figure 11: A composite Mapping

copy mapping to expand A and then uses a permute mapping to shift it along the x axis.

4 Implementation

A mapping is implemented by means of a source to source transformation of the UC program such that the data in the transformed program is shifted as specified in the mapping but the program is functionally equivalent to the original program. Implementing a mapping involves one or more of the following tasks:

- If the target array in the mapping has a different size or shape than the source (example, transpose or expand), the declaration for the source must be modified in the transformed program. For instance, if an array $A_{n,m}$ is mapped to its transpose, the transformed program must include a declaration for $A_{m,n}$. Similarly, if $A_{n,m}$ is expanded using a copy mapping to a 3-dimensional array, then the declaration for array A must be modified accordingly in the transformed program.

- References to the source array in the original program must be modified in the transformed program. For instance, a permute mapping may map $A[i]$ to $A[i\text{-}1]$. A statement of the form $A[i+1]=B[i]$ must then be modified to $A[(i + 1)\text{-}1]=B[i]$ in the transformed program.

- Expressions in the modified references must be simplified (where possible) to allow compile time detection of local references in the transformed program. In the preceding example, in order that the compiler be able to detect the modified assignment as a completely local operation, the subscript expression must be simplified to yield $A[i]=B[i]$.

- Additional UC code may have to be introduced in the transformed program to ensure that it is functionally equivalent to the original program. For instance, a copy mapping creates multiple instances of a data item in the transformed program. If an element in the source array is modified in the original program, the transformed program must include a broadcast to suitably update all instances of the data element.

The next section presents the experimental results of including each of the four types of mappings in sample programs. Implementation issues that arise for a specific type of mapping are examined in further detail in the context of the sample programs.

5 Performance

This section presents a complete example for each of the four types of mappings that were described in the previous section. The examples were chosen for expository purposes and are not necessarily representative of real UC programs. Each example illustrates a specific type of mapping. The discussion for the example includes a short description of the program and the mapping and also includes experimental measurements on the speedup obtained with each mapping. Speedup is defined as the ratio of the execution time for the original program to the execution time for the transformed program. After measuring speedup, each example was modified to include a code fragment for which the default mappings generated by the compiler are optimal. The following is an example of such a fragment:

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

Program transformations to implement a specific programmer specified mappings for arrays A and B will cause the preceding fragment to be executed less efficiently than with the default mappings. A fragment with this form of access pattern is referred to as 'previously ideal references'. The measurements for each example include an estimate of the extent to which previously ideal references can be tolerated by each mapping before the inefficiency introduced by modifying the 'ideal' access pattern outpaces the efficiency gained by a specific mapping.

The results reported in this section were obtained using a CM-2 [12] with a sun/4 front end. All runs used 8k physical processors. The mapped code shown is actual code generated by the UC optimizer, with some numerical constants replaced by the corresponding symbols for readability. Speedup is defined as the ratio of the execution time for the original program to the execution time for the transformed program.

5.1 Copy

A copy mapping is used to reduce the need for broadcasts. Consider the program of figure 12 which uses an O(N^3) matrix multiplication algorithm to compute the all-pairs reachability matrix (r) for a graph. Assume that A represents the adjacency matrix. The program computes r by calculating successive powers of the adjacency matrix until a fixed point is reached. To remove the randomness from the algorithm, a linear graph was chosen.

The innermost reduction in line 11 is an obvious target for optimization, as it is the most frequently executed operation. The reduction involves remote accesses of array r and array A. The remote accesses may be reduced either by using a copy mapping to expand the arrays to 3-dimensional objects, or by using a reduce mapping to collapse one dimension and take advantage of the common index k. As the program uses an O(n^3) algorithm, it is preferable to use a copy mapping which is specified in lines 15-20 of the program.

The actual code generated by the UC optimizer is also shown in Lines 1-13 in figure 12. Note that all remote accesses to arrays r and A in the original program are replaced by local accesses of the replicated array in the transformed program. In particular, whereas the $(i,j,k)^{th}$ processor reads $r[i][k]$ remotely in line 11 of

```
1       { int A[N][N], r[N][N], tmp[N][N];
2            /* initially r=A and tmp != r */
3           init();
4
5           /* While some element of r is still changing */
6           while ($||(I,J; r[i][j]!=tmp[i][j]))
7               /* multiply r by the adjacency matrix */
8             par (I,J)
9               {
10                  temp[i][j] = r[i][j];
11                  r[i][j] = $||(K; r[i][k] && A[k][j]);
12              }
13      }
14
15      map {
16          virtual 3DMEM[I][K][J];
17
18          copy(I,K) r[i][k] :- 3DMEM[i][k][j];
19          copy(K,J) A[k][j] :- 3DMEM[i][k][j];
20      }
                 ||

1       {int A[N][N][N], r[N][N][N], tmp[N][N];
2
3           init();
4           while ($|| (I, J; r[i][j][j] != temp[i][j]))
5           {
6           par(I, J)
7           {
8               temp[i][j] = r[i][j][j];
9               r[i][j][0] = $|| (K; r[i][k][j] && A[i][k][j]);
10          }
11          par(J, K, I) r[i][k][j] = r[i][k][0];
12          }
13      }
```

Figure 12: An example of copy mapping

the original program, it reads it locally as $r[i][k][j]$ in line 9 of the transformed program.

As array A is not modified by the par statement, its replication does not introduce significant overhead. However, r is both modified and referenced within the par statement in the original program. This implies that the modification must be broadcast to update all copies of array r. As seen in line 11 of the transformed program, this broadcast may be implemented efficiently by using a single parallel assignment. The overall efficiency gain achieved by this transformation is shown in table 1. The results show considerable speedups within the range of array sizes tested in our experiments.

Matrix Size	*Original Timing*	*Mapped Timings*	*speedup*
19	1.36	0.67	2.03
25	5.11	1.59	3.21
31	19.44	3.85	5.05
39	59.13	9.92	5.96
49	154.1	23.93	6.44

Table 1: Timings for Transitive Closure

Tests show that for this example, the transformed program outperforms the original program as long as the ratio of the number of previously ideal references (defined in the beginning of this section) to the number of references optimized by the mapping is less than 2.5 : 1. The primary overhead for this mapping are the broadcasts required for mapped arrays that are both modified and referenced in the program. An increase in the number of broadcasts will reduce the speedup.

5.2 Reduce

Reduce mappings improve execution efficiency of a program by decreasing remote access of data and also by reducing the VP ratio for the program. Figure 13 presents an example that computes the trace of the product of two matrices without computing the product matrix. The reduction on line 5 computes the diagonal elements of the product, and the reduction on line 6 sums the elements to compute the trace. The reduction in line 5 is the most expensive operation in the program. A copy mapping cannot be used to improve the efficiency of this program as expanding B or C will dramatically increase the VP ratio and expanding A does not reduce the remote access. Instead, a reduce mapping is used to collapse each array along its smaller dimension. The mapping in lines 9-14 in the figure specifies that B be collapsed along its second dimension and C along its first. A transpose mapping will also improve the efficiency of this program, but this mapping is discussed in a separate section.

The transformed code is shown in lines 1-12 at the bottom of figure 13. Lines 1-2 show the modified declarations for B and C. Note that the reduction in line 5 of the original program is now computed sequentially (line 8) for each element of the diagonal (although different elements of the diagonal are still computed in parallel).

The performance gains obtained with this mapping are shown in table 2. While considerable speedups result from this mapping, as expected, the gain decreases as the size of the dimension being contracted is increased. The main inefficiency of the transformed program is that each element of the diagonal must now be computed sequentially. However, the last entry in the table shows

```
int B[M][N], C[N][M];

{
    par (I)
        A[i] = $+(K; B[i][k] * C[k][i]);
    trace = $+ (I; A[i]);
}

map {
    virtual M1[M];

    reduce (I,K) B[i][k] :- M1[i];
    reduce (I,K) C[k][i] :- M1[i];
}
       ||
struct { int val[N];} C[M];
struct { int val[N];} B[M];
int   _0_tmp_[N];
{
    par(I)
    {
    _0_tmp_[i] = 0;
    seq(K) _0_tmp_[i] += B[i].val[k] * C[i].val[k];
    A[i] = _0_tmp_[i];
    }
    trace = $+ (I;    A[i]);
}
```

Figure 13: A reduce mapping

that even for square matrices of reasonably large size, the speedup is significant. Resource constraints prevented our being able to run experiments for larger size arrays. The additional experiments will be useful in determining the largest array that may be contracted before the performance of the transformed program becomes worse. *We hope to include timings for larger configurations in the final version of the paper.*

*Size:M*N*	*Original Timing*	*Mapped Timings*	*speedup*
250*32	0.74	0.08	9.25
250*64	0.92	0.16	5.75
250*128	1.22	0.33	3.69
350*350	3.61	0.89	4.05

Table 2: Timings for Trace

Tests show that for this example, the transformed program outperforms the original program as long as the ratio of the number of previously ideal references to the number of references optimized by the mapping is less than 3 : 1. The reduce mapping causes all elements in the reduced dimension to be stored on one processor. This implies that the memory available per processor may also constrain the use of this mapping for larger arrays.

5.3 Fold

Fold mappings are useful for programs that include multiple references to the same data structure in a single operation. An example program that benefits from such mappings is described in figure 14. The assignment statement in line 14 is the most frequently executed statement in the program. As the statement contains a symmetric reference to array A, the array is a suitable candidate for a fold mapping. The mapping is specified in lines 17-23 in the original program. The program includes a code fragment (line 6-8) that is mapped optimally using the default mappings. Implementing the fold mapping for array A will make execution of this fragment inefficient. All previous examples included similar code which was used to determine the ration of previously ideal references that could be tolerated by the corresponding mapping.

Folding an array implies that some locations of the initial array will contain multiple elements. In this example, a temporary array is declared in line 1 of the transformed program to store the 'doubled up' elements. Further, as a fold mapping moves only a subset of the elements in the array being mapped, every reference to the array in the original program generates a conditional reference in the transformed program, where one branch of the conditional refers to the unmapped elements of the original array, and the other refers to the mapped elements. For instance, the assignment statement in line 8 of the original program is divided into two parts in the transformed program (lines 7-10).

Other things remaining the same, the speedup achieved for this transformation will increase linearly with the size of the array. More interestingly, as seen from table 3, the speedup is independent of the shape of the array as long as the toatl number of elements are kept constant. This is counter-intuitive because folding an array with a larger number of columns should result in a smaller VP ratio and hence better performance. Unfortunately, the VP ratio does not improve. Although the size of the 'folded' array in the transformed program does decrease (line 1), the **par** statements in the transformed program (lines 5,12) are still executed over the old index sets I and J, preventing the number of virtual processors from being reduced. Ideally, the transformed code should look like this:-

```
/* S is a new index set for j < N/2 */
par (I,S)
    B[i][s] = A[i][s] + _dupA_[i][s];
```

```
int A[M][N];
int B[M][N/2];

{
    /* A previously ideal reference */
    par (I,J)
    seq(R)
        A[i][j] = i *j;

    /* A symmetric reference to A */
    par (I,J)
    seq (K)
    st (j<N/2)
        B[i][j] = A[i][j] * A[i][j+N/2];
}

map {
    virtual M1[M][N/2];

    fold (I,J)
        st (j>=N/2)
            A[i][j] :- M1[i][j-N/2];
}
```

||

```
int   A[M][N/2], _2cc60_A[M][N/2];
int   B[M][N/2];

{
    par(I, J)
    seq(R)
    st(j >= N/2)
        2cc60_A[i][j - N/2] = i * j;
    st(!(j >= N/2))
        A[i][j] = i * j;

    par(I, J)
    seq(K)
    st(j - N/2 < 0)
        B[i][j] = A[i][j] * _2cc60_A[i][j];
}
```

Figure 14: An example for fold mappings

The preceding code not only contains local accesses but also reduces the VP ratio as compared to the original program. However, generation of the preceding code involves more complex code simplifications than are currently implemented. For the current implementation, a ratio of 1.5:1 for previously ideal references yields overall performance gains.

*Size:M*N*	*Original Timing*	*Mapped Timings*	*speedup*
8000*16	48.45	27.3	1.77
2000*64	48.47	27.3	1.77
500*256	48.48	27.3	1.77

Table 3: Timings for fold mappings

5.4 Permute

Permute mappings can improve execution efficiency by replacing a previously remote access by a local access. In some cases a permute mapping may also reduce the VP ratio of a program. For example, consider the case when the code contains declarations for two arrays, $A[M][N]$ and $B[N][M]$, $M \neq N$. With the default mapping, the program would require $(\max(M, N))^2$ virtual processors. If a transpose mapping is used to map A onto B, the virtual processor requirement drops to $M * N$.

Figure 15 presents an example of the use of a transpose mapping for an $O(n^2)$ matrix multiplication algorithm. The N^2 elements of the product matrix are computed in parallel (line 4); however the summation to compute each element is performed sequentially (lines 7-8). The major computation in this algorithm is the sequential loop in lines 7-8. This code will not benefit from a copy mapping as it only uses n^2 processors. A reduce mapping of both A and B would leave C as a two dimensional array and would not result in any gain. However, a transpose mapping may reduce the VP ratio as described earlier. More significantly, the transpose mapping will cause all data items involved in the multiplication on line 8 of the original program to be aligned along a NEWS grid axis in the transformed program (line 8).

Table 4 shows the timings obtained for this example for various array sizes. The improvement in execution time are broken up into two parts, one due to the reduction in VP ratio and the other due to the use of the NEWS grid. The use of NEWS grid reduces the execution time by a factor of 5, and the reduction in the VP ratio yields an additional speedup of about 1.8, giving an overall performance gain of a factor of 10.

```
int A[M][N], B[N][M], C[M][M];

{
    par (I,K)
    {
        C[i][k] = 0;
        seq(J)
            C[i][k] += A[i][j]*B[j][k];
    }
}

map {
    permute (J,K)
        B[j][k] :- A[k][j];
}
```

||

```
int A[M][N], B[M][N], C[M][M];

{
    par(I, K)
    {
    C[i][k] = 0;
    seq(J)
        C[i][k] += A[i][j] * B[k][j];
    }
}
```

Figure 15: A permute mapping

*Size:M*N*	*Original Timing*	*Mapped Timings original VP ratio*	*Mapped Timings reduced VP ratio*	*speedup*
125*250	64K,107.58	64K, 20.66	32K, 11.63	1.78
90*175	32K, 42.58	32K, 7.47	16K, 4.26	1.75
60*125	16K, 9.87	16K, 3.16	8K, 1.87	1.69

Table 4: Timings for permute mappings

6 Discussion

An approach to generate efficient mapping of high-level programs on a specific parallel architecture has been developed. The initial program does not specify the mapping of program data structures on the parallel memory. Instead the mapping is generated automatically by the compiler. Subsequently, the programmer can override the default mapping of some structures to improve the performance of the algorithm. Mappings specified by the programmer leave the correctness of the algorithm invariant.

The current implementation places certain restrictions on the data structures whose default mappings can be modified by the programmer. In particular, an array that is specified as the source in a mapping cannot be aliased in the program, unless the alias is also mapped in an identical manner. Furthermore, it is not currently feasible to map a specific array element on multiple processors. This facility may be useful for arrays of structs, where each struct itself contains an array. The current implementation will always force the entire struct to be mapped on a single processor memory.

The research described in this paper may be extended along different directions, some of which are currently being explored. In particular, this paper assumed that the data mappings are static. If the data access patterns of a program change dynamically, a static mapping will typically be unable to optimize all remote accesses executed by the program. Dynamic mappings require that each mapping be treated as an executable statement rather than as a declarative specification. Dynamic mappings have higher overheads and are considerably more complex to implement than static mappings. However, preliminary investigations indicate that they may significantly improve execution efficiency. As a simple example consider an array that is manipulated both sequentially and in parallel by some program. Such an array will benefit from being dynamically shifted among the front-end and data processor memory in response to the access pattern for its elements. Similarly the elements of an array may be reordered dynamically to optmize its current access pattern. Efforts to include dynamic mappings in a UC program are in progress.

Automatic generation of the data mappings is another avenue for further exploration. Automatic mappings may be generated on the basis of built-in heuristics (example, use copy mappings for matrix multiplications) as well as from a compile-time dependency analysis of the access patterns for program data. If the array subscripts use variables, a compile-time analysis will not yield sufficient information to generate useful mappings. In our limited experience with programming scientific applications in UC (particle diffusion in fluids, SVD applications, diagonalization of symmetric matrices ...) subscript expressions generally include only index-sets and literal constants. For such programs, automatic generation of data mappings is feasible and will be investigated. A hybrid approach where obvious data mappings are automatically generated by the compiler and others are specified by the programmer will also be investigated.

7 Related Work

The UC research was motivated by UNITY[3], a specification and programming notation for parallel programs. The UNITY work clearly enunciates the advantages of separating the programming concerns from efficiency considerations. It encourages an initial design of an architecture-independent program and suggests how the initial prototype may be refined in a step-wise manner for eventual implementation on diverse architectures.

SETL[6] seems to have been among the first languages to separate programming issues from performance concerns. SETL is a high-level language based on set theoretic concepts. A SETL program is developed almost entirely as a sequence of set operations. The program does not explicitly indicate how the sets are implemented (arrays, linked list, hash table, ...). This may either be specified separately by the programmer or be generated automatically by the compiler[10]. To the best of our knowledge, SETL has not been implemented on a parallel architecture.

More recently, a functional notation called Crystal[4] has been designed to separate algorithmic issues from performance concerns. Crystal defines the notion of a data field as a data-set together with a description of its shape and topology. Code optimizations are specified as transformations (called morphisms) on the data fields and are used to generate efficient data implementations on a parallel architecture. Crystal has been implemented on multicomputers like the NCUBE and the Intel iPSC.

CM Fortran[9] provides a restricted facility to declaratively change the mapping of an array on the CM memory. In particular, the *layout* directive may be used to force an array to be stored on the front-end or cause the array to be stored on the parallel memory such that certain axes lie along the NEWS grid. The *align* directive may be used to specify the layout of an array relative to another array declared in the program. Together these directives allow a programmer to chnage the mapping of an array in a manner similar to the permute mappings defined for UC.

This project was also influenced by the considerable body of existing research in the area of parallelizing compilers[1, 5, 7]. Parallelizing compilers allow sequential programs to be automatically executed on parallel architectures primarily by transforming sequential loops into a sequence of parallel assignments. A number of sophisticated dependency analysis techniques have been developed for these compilers. Similar techniques are useful in both implementation and automatic generation of the data mappings. A reduce mapping may cause a parallel UC operation to be implemented sequentially. As an example consider a reduce mapping that causes a statement like par(I) a[i]=a[i-1] to be sequentialized. A naive sequential implementation may execute this statement for increasing values of i. This would require that temporary storage be used to save intermediate values of the array. In contrast, if the statement is executed for decreasing values of i, it may be implemented without using any temporary storage. The required information may be easily generated from a dependency analysis of the program.

References

[1] J.R. Allen and K. Kennedy. *PFC: A Program to Convert Fortran to Parallel Form*, pages 186–205. IEEE Computer Society Press, 1985.

[2] R. Bagrodia, K.M. Chandy, and E. Kwan. UC: A Language for the Connection Machine. In *Supercomputing '90*, New York, November 1990.

[3] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, Inc., 1988.

[4] M. Chen, Young-Il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *JOurnal of Supercomputing*, (2):171–207, 1988.

[5] D.A.Padua, D.J.Kuch, and D.H.Lawrie. High Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers*, C-29:763–776, September 1980.

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

[7] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Eigth ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[8] Edmund Kwan. The UC Programming Language. Master's thesis, UCLA, 1990.

[9] 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.

[10] E. Schonberg, J. Schwartz, and M. Sharir. An automatic technique for the selection of data representation in SETL. *ACM TOPLAS*, 3(2), April 1981.

[11] Thinking Machines Corporation. *C* Reference Manual*, August 1987. version 4.0.

[12] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*, April 1987.