Paralation Views:
Abstractions for Efficient Scientific Computing on the Connection Machine

Kenneth J. Goldman

June 22, 1989

Abstract

An ideal parallel programming language for scientific applications should provide flexible abstraction mechanisms for writing organized and readable programs, encourage a modular programming style that permits using libraries of tested routines, and, above all, permit the programmer to write efficient programs for the target machine. We use these criteria to evaluate the languages *Lisp, Connection Machine Lisp, and Paralation Lisp for writing scientific programs on the Connection Machine. As a vehicle for this exploration, we fix a particular non-trivial algorithm (LU decomposition with partial pivoting) and study code for implementing it in the three languages.

Based on our findings, we propose two extensions to Paralation Lisp for writing scientific programs. The first extension is a new mapping facililty, which reduces communication overhead from $O(\lg n)$ to $O(1)$ in many situations. The second extension, called *Paralation Views*, is an enhancement of the Paralation Lisp shape facility. By allowing the programmer to view the same set of data with multiple abstractions, this extension results in programs that are both more readable and more efficient. A possible implementation strategy is presented. Paralation Views integrates well with the existing Paralation Lisp language and provides excellent support for modularity and nested parallelism.

Keywords: programming languages, parallel programming languages, SIMD architectures, Connection Machine, scientific programming, data abstraction

This research was supported in part by the National Science Foundation under Greant CCR-86-11442, by the Office of Naval Research under Contract N00014-85-K-0168, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

1 Introduction

For a programming language to be acceptable to the scientific community, it must provide tools for writing programs that run efficiently on the target machine. This requirement, which cannot be overemphasized, partially explains why FORTRAN has remained the overwhelming favorite for writing sequential scientific programs. Of course, flexible data abstraction mechanisms and support for modular programs can result in considerable savings in initial design effort, debugging, and later modification of programs. But these features alone do not constitute a good language for scientific programming; when given the choice between a high-level language with degraded performance and a low-level language with good performance, the scientific community will invariably prefer the latter.

However, we contend that such a trade-off is not necessary. In this paper, we evaluate three languages in terms of their utility for writing scientific programs on a particular parallel architecture. The languages, all derived from a common base language, range from relatively low-level to relatively high-level. We consider the advantages and disadvantages of each language in terms of its expressive power and its efficiency. Based on our findings, we propose enhancements to one of the high-level languages that result in the ability to write programs with efficiency comparable to that of programs written in the "low-level" language. Interestingly, these enhancements do not take the form of low-level system calls or awkward hints to an optimizing compiler, but rather take the form of *additional abstraction mechanisms* that not only allow programs to run faster, but also result in code that is more modular and easier to read.

As our target machine architecture, we use the Connection Machine [5], specifically the CM-2 [13].[1] The Connection Machine is a SIMD computer (Single Instruction, Multiple Data) with 2^{16} processing elements, each having 64K bits of local memory. The processors are arranged in a 12-dimensional hypercube communication network, with 16 processors and one communications router per chip. When the network is uncongested, communication may be considered to be a unit time operation. User programs, which run on a *front-end* computer, cause instructions be broadcast to the processing elements on a bus. This bus is also used to retrieve data from the processors for delivery to the front-end.

In a SIMD computer, the processors execute the same instruction simultaneously, but the results at each processor may depend upon data values in that processor's local memory. Each processor maintains a *context bit*, which indicates whether or not it is *active*. Certain instructions are *conditional*, executed only by active processors. The remaining instructions are *unconditional*, executed by all processors. Instructions may cause local computation, communication, or a change in the set of processors. In addition, the Connection Machine provides a "wired-OR", which allows certain global conditions to be computed quickly. For example, the wired-OR is useful for determining when all processors have terminated an iterative computation.

The languages we consider are *Lisp [14], Connection Machine Lisp [11, 5], and Paralation Lisp [8, 9]. All are based on Common Lisp [10], and all have implementations running on the Connection Machine. In *Lisp, the most low-level of the three, programmers explicitly control the context bits of the processors using special functions, and no data abstraction mechanisms are provided. Programs in *Lisp tend to be monolithic, but quite efficient. Connection Machine Lisp provides a single abstraction, called the *xapping*, which permits context to be selected implicitly. However, significant communication overhead often results from manipulating xappings, especially when one wishes to work on only a portion of a xapping. In addition, the xapping may not be

[1]This paper does not address the question of whether or not the Connection Machine architecture is well-suited for particular scientific applications, but simply takes the machine as a given and concentrates on the language issues.

the most useful abstraction for many scientific applications. Finally, Paralation Lisp provides the *paralation* as the abstraction mechanism, and allows programmers to create paralations of arbitrary *shapes*, with particular locality properties and access methods. A small set of powerful operators for manipulating paralations are provided. The locality properties can result in improved efficiency, but working with portions of a paralation remains difficult and expensive. The power and generality of the communication mechanism can result in unnecessary overhead.

Based on our observations about these languages, we propose two extensions to Paralation Lisp. The first extension is a new mapping facililty that reduces communication overhead from $O(\lg n)$ to $O(1)$ in many common situations. The second extension, called *Paralation Views*, is an enhancement of the shape facility that allows a given paralation to be viewed not only as a single shape, but as multiple different shapes. With this facility, different views of the data may be efficiently created during program execution using a special set of operators (*project*, *split*, and *extract*). Thus, at each point in a program, the abstraction appropriate for that stage of the computation may be used. In addition, Paralation Views permits many operations to be performed *in place* that would otherwise incur expensive communication overhead. Therefore, programs using this facility are both more readable and more efficient. Paralation Views integrates well with the existing language, and gives excellent support for modularity and nested parallelism.

The remainder of the paper is organized as follows. In Section 2, we describe a particular algorithm that is used as a starting point for our discussion of the languages. An overview of each language is presented in Section 3, and evaluations and comparisons are made based on analysis of corresponding code for the example algorithm. In Section 4, we present the new mapping facility for reducing communication overhead. In Section 5, we present Paralation Views, and Section 6 presents the example algorithm written using Paralation Views. In Section 7, we discuss implementation. Finally, Section 8 describes a programming paradigm which is useful for achieving nested paralellism using Paralation Views. We conclude with a summary and some possible directions for further research.

2 An Example Algorithm

As a common thread throughout our discussion of the languages, we fix a particular non-trivial algorithm that is amenable to efficient execution on the Connection Machine. The algorithm, LU decomposition with partial pivoting, is non-trivial for several reasons. First, the communication pattern is data dependent. One cannot predict which processors will send messages to which other processors at a given stage of the algorithm until actually reaching that stage. Second, it requires the set of selected processors to vary throughout the execution; not all processors participate at all times.

LU decomposition is useful for solving systems of linear equations. A matrix L is *unit lower triangular* if and only if all elements on its main diagonal are 1, and all elements above its main diagonal are 0. A matrix U is *upper triangular* if and only if all elements below its main diagonal are 0. Consider the equation $Ax = b$, where A is an $n \times n$ matrix and b is a vector of length n. One way to solve for x is to perform the following steps. First, decompose the matrix A into a unit lower triangular matrix L and an upper triangular matrix U such that $A = LU$. [2] Then, solve for y in $Ly = b$ by forward elimination. Finally, solve for x in $Ux = y$ by back substitution.

LU decomposition with partial pivoting has been studied extensively as a parallel algorithm [2, 6]. The input to the algorithm is an $n \times n$ matrix A. The output matrix is obtained by

[2]We should note that an LU decomposition does not exist for some matrices [1]. The programs in this paper do not check for this situation.

```
LU-DECOMPOSITION(A: $n \times n$)
    vector I: $n \times n$, Temp: $n \times n$
    for $i \leftarrow 0$ to $n-2$
        do $v, k \leftarrow$ MAX-SCAN($i$,A)                /* max-row */
           $t \leftarrow -1.0/v$
           if (row(I) $= i$ and col(I) $\geq i$)          /* swap */
             then Temp $\leftarrow$ A
                  A $\leftarrow$ A$\langle k$,col(I)$\rangle$
                  A$\langle k$,col(I)$\rangle \leftarrow$ Temp
           if (row(I) $> i$ and col(I) $= i$)             /* normalize */
             then A $\leftarrow$ A$*t$
           if (row(I) $> i$ and col(I) $> i$)             /* update */
             then A $\leftarrow$ A + A$\langle$row(I),$i\rangle*$A$\langle i$,col(I)$\rangle$
    return A

MAX-SCAN($i$, M: $n \times n$)
    vector I: $n \times n$, R: $n \times n$
    $k \leftarrow 1$
    R $\leftarrow$ I
    while (row(I) $-k$) $\geq i$
        do if abs(M) < abs(M$\langle$I$-k,i\rangle$)
             then M $\leftarrow$ M$\langle$I$-k,i\rangle$
                  R $\leftarrow$ R$\langle$I$-k,i\rangle$
           $k \leftarrow 2k$
    return M$\langle n,i\rangle$,R$\langle n,i\rangle$
```

Figure 1: LU Decomposition Program in SIMD Pseudocode.

successively modifying A with a sequence of steps that is executed for each index i from 0 to $n-2$:

1. **Max-row:** Find v, the maximum absolute value in column i on or below the main diagonal, and let $k \geq i$ be the index of a row having v in column i.
2. **Swap:** Swap elements i through $n-1$ of rows i and k.
3. **Normalize:** Multiply all elements below the main diagonal in column i by the value $-1.0/v$.
4. **Update:** For all $i < r, c < n$, let $A[r,c] = A[r,c] + A[r,i] * A[i,c]$.

The resulting matrix (call it M) is not actually the LU decomposition of the original matrix A, but is sufficient for solving systems of equations of the form $Ax = b$. Let A' be the input matrix with whole rows reordered according to the sequence of swaps performed in the algorithm, and let $A' = L'U'$. The values on and above the main diagonal of M form U', and the negated values below the main diagonal of M (with partial rows reordered as for A') form the matrix L'.[3] By similarly reordering b, one can find x using forward elimination and back substitution as described above.

The SIMD pseudocode in Figure 1 illustrates the steps for executing this algorithm on the Connection Machine. The pseudocode syntax is due to [4]. Vectors are allocated one element per processor. "I" is a special vector that declares the maximum (and initial) set of active processors.

[3]Partial pivoting may save time and/or space on some systems [2], but there seems to be no reason to avoid swapping entire rows in Step 2 on the Connection Machine. This would enable one to read L' directly from the output matrix without reordering. However, we do the partial pivoting for purposes of illustration.

On each processor, "I" contains the id of that processor, and row(I) and col(I) denote its grid coordinates. Context is modified by the **if** statement and the **while** statement, which terminates when no processors are active. A vector variable without a subscript refers to the elements of that vector at active processors. A vector variable with a subscript denotes a "get" or "send," as appropriate. The MAX-SCAN subroutine is an example of a logarithmic *scan* computation.

Throughout the paper, we assume that the elements of the input matrix are appropriately distributed on the Connection Machine processors, one element per processor. We do not consider how I/O is handled in any of the languages. In addition, we do not address the question of whether or not this particular algorithm is the best one for solving LU decomposition on the Connection Machine, or even if LU decomposition is the best way to solve systems of linear equations on the Connection Machine. We simply take the algorithm as a useful vehicle for exploring the languages.

3 Three Languages

This section compares three languages, *Lisp, Connection Machine Lisp, and Paralation Lisp, in terms of their abiliby to express efficient scientific programs for the Connection Machine. For each language, we present a brief overview and discuss its particular strengths and weaknesses, referring to corresponding code for LU decomposition. Although glancing at the example program for each language is useful, it is not necessary to understand all the details in order to benefit from the discussion. In Section 3.4, we summarize our observations and motivate the ideas presented in later sections.

3.1 *Lisp

3.1.1 Language Overview

The parallel data structure of *Lisp is the *pvar*, which represents a "slice" of the Connection Machine memory: a pvar is an array having one element at the same address in each processor. At initialization time, one specifies whether pvars in a program are one- or two-dimensional. One cannot mix the two types in the same program. The function `*defvar` creates a permanent pvar, and `*let` creates a pvar in the current scope. In addition, a pvar can be created with a !! suffix (called the "bang-bang" operator). For example, `3!!` creates a pvar in which every element contains the value 3. (One might think of !! as the front-end shouting to the processors!!)

Individual entries of a pvar are accessible with `pref` and `pref-grid`, depending upon the number of dimensions in use. For example, if `mat` is a two-dimensional pvar,

```
(setq (pref-grid mat 3 4) (pref-grid mat 5 4))
```

copies the value of mat in processor (5,4) to the location of mat in processor (3,4). At the time a pvar is created, the currently selected set (i.e., the set of processors whose context bit is 1) determines which processors contain values for that pvar. Therefore, if processor (3,4) or (5,4) is not in the selected set when `mat` is created, the above statement causes an error. Processors may discover their own addresses with the functions `self-address!!` and `self-address-grid!!`, which takes a dimension argument.

The currently selected set is modified by a set of special functions. For example, in the statement (`*all` *body*), every processor is active at the start of *body*. Within *body*, a statement of the form (`*when` *predicate nested-body*) would cause all processors whose local variables do not satisfy the predicate to be removed from the currently selected set before *nested-body* is executed. The

```
(*defun lu-decomp (mat-pvar n)
    (*let ((temp mat-pvar)
           (result))
           (row (self-address-grid!! 0!!))
           (col (self-address-grid!! 1!!))
          (declare (type (pvar double-float) temp))              ;; optimizes operations on temp
          (declare (type (pvar double-float) result))            ;; optimizes operations on result
          (*all
              (do ((i 0 (1+ i)))
                  (= i (1- n))
                          (result)
                  (*when (logand!! (>=!! row i!!) (=!! col i!!))          ;; max-row
                          (setq v (*max (abs!! temp)))
                          (*when (=!! temp v!!)
                                  (setq k (*min row))))
                  (*when (logand!! (=!! row i!!) (>=!! col i!!))          ;; swap
                          (setf result (pref-grid!! temp k!! col :nocollisions))
                          (setf (pref-grid!! temp k!! col :nocollisions) temp))
                  (*when (logand!! (>!! row i!!) (=!! col i!!))           ;; normalize
                          (*set result (*!! temp (/ -1.0 v)!!)))
                  (*when (logand!! (>!! row i!!) (>!! col i!!))           ;; update
                          (*set temp (+!! temp (*!!
                                  (pref-grid!! result row i :many-collisions)
                                  (pref-grid!! result i col :many-collisions)))))))))
```

Figure 2: LU Decomposition Program in *Lisp.

currently selected set is dynamically scoped. After *nested-body* terminates, the currently selected set is restored to the value it had before the `*when`.

The bang-bang operator is used to express parallel computations. The statement

```
(setq pvar-a (+!! pvar-a pvar-b))
```

increments the values in pvar-a by the values in pvar-b (in the currently active processors). Communication is accomplished similarly with `pref!!` and `pref-grid!!`. To optimize communication, the programmer may specify "no-collisions," "many-collisions," or "collisions-allowed" as an optional argument, according to the number of reads from the same location. For data aggregation, *Lisp provides a set of functions which operate on the currently selected set. For example, `(*min pvar-a)` returns the smallest value in `pvar-a` among all active processors. In addition, the functions `scan!!` and `scan-grid!!` are available for performing a scan computation on a pvar with a combining function chosen from a predefined set (e.g, +, *, etc). The programmer may not supply an arbitrary function to `scan!!`.

3.1.2 Discussion: Why *Lisp programs are non-modular

Figure 2 contains the *Lisp program for LU decomposition. One striking feature about *Lisp is that writing efficient programs is fairly easy, because the machine model is completely transparent. The programmer has complete control over how the data is distributed on the processors. Processor selection is made explicit, and it is obvious where the communication is taking place. Hints to the compiler, like the types of pvars and communication collision rates, provide additional control.

This closeness to the machine model also causes some problems. In the LU decomposition example, we were fortunate enough to have a two-dimensional grid as our only data structure. However, programs that use more complicated data structures or mix different types of data structures must perform awkward address computations. A related problem, that causes *Lisp programs to be rather monolithic, is that one can't isolate a portion of the data and pass it to a function as a new structure. It is possible in *Lisp to select a set of processors and then call a procedure, but the procedure is forced to work within the coordinate system of the larger structure. One cannot create a library of routines that operate on vectors, and then pass pieces of rows or columns of matrices to them.

Also contributing to the non-modularity of programs is a peculiar interaction between the applicative style of Lisp and the nested selection operators of *Lisp. This interaction makes it difficult to use an applicative style. Instead, the comfortable programming paradigm is: set context, compute, *set variables*, set context, etc. In support of this claim, consider the following applicative style code fragment that attempts to compute Step 1 of the example algorithm.

```
(*min (*when (=!! matrix (*max
         (*when (logand!! (=!! col i!!) (>=!! row i!!)) (*defvar p matrix))))
            (*defvar q row)))
```

The `*max` expression is supposed to find the maximum value v in column i, on or below the main diagonal. The surrounding `*min` is supposed to find the smallest index among rows having v in column i on or below the main diagonal. But this is not what happens. Because the selected set is restored on return from the inner `*when`, the pvar p is not defined for all the selected processors when the maximum is taken. Therefore, the `*max` clause references undefined values! Even if we suppose that the `*max` clause returns the correct value v, the final value returned is still not the desired index, but instead is the minimum row index among all elements with value v in the *entire matrix*. This is again because context is restored upon return from the inner `*when`.

One possible solution is to avoid explicit context selection in programs by providing higher-level language features that permit selection to be accomplished implicitly. This approach is taken by the next language we consider.

3.2 Connection Machine Lisp

Connection Machine Lisp provides a single data abstraction, the *xapping* (read "zapping"), which maps a *domain* of values to a *range* of values. Since exactly one value from the range is specified for each value in the domain, a xapping is a mathematical function. Xappings are written as follows.

{ a→16 b→32}

A *xet* is a xapping in which each domain element maps to itself. A xapping whose domain is a prefix of the non-negative integers is called a *xector*, and may be abbreviated as a sequence of values in square brackets. A constant xapping maps all indices to the same value. For example, {→c} evaluates to c at all indices. The function `xref` is used to evaluate a xapping at a particular index. Complex data structures are created by nesting xappings. A matrix might be represented as a xector of xectors, where each inner xector is a row of the matrix.

Besides simply typing them in, xappings may be created with `iota` and the α operator. The function `iota` takes a integer length as input and returns a xector xet of that length. For example, `(iota 4)` returns the xector `[0 1 2 3]`. The α operator is a powerful tool that takes a value and produces a constant xapping with that value. Extending Lisp to allow xappings in the position of function calls enables the α operator to be used for performing functions element-wise on a xapping or set of xappings, to produce a new xapping. For example,

```
(α* (iota 4) α3) ⇒ [0 3 6 9]
(α+ (iota 4) (iota 5)) ⇒ [0 2 4 6]
```

Notice that the function is applied only to the values whose indices are in the intersection of the two domains. The α operator may be factored out of expressions, using • as its inverse. The first expression above is equivalent to `α(* •(iota 4) 3)`.

Parallel communication is accomplished with the β operator, which takes a combining function f, a destination xapping d, and a source xapping x. One can think of the elements $p \rightarrow q$ of destination xapping d as saying, "send `(xref x p)` to processor q." If q occurs more than once as a value in d, then the values sent to q are combined according to f. So, in the resulting xapping, the domain is determined by the values of d, and the range is determined by the values of x. For example,

```
(β* '{ a→0 b→1 c→1 d→3} '{ a→37 b→4 c→16 d→9 } ) ⇒ { 0→37 1→64 3→9}
```

Aggregation on a single xapping is also accomplished with the β operator. An expression of the form $(\beta f\ x)$ returns the value produced by combining all the values of xapping x with function f. For example, `(βmin x)` would return the minimum value in xapping `x`. Any combining function may be supplied.

CM Lisp provides a number of operators for xappings. For example, `xunion` takes two xappings and a combining function, and returns the union. The combining function is applied when the same index occurs in both xappings. A special combining function, `@`, signals an error if it is called. The function `over` is defined in term of `xunion`:

```
(defun over (a b) (xunion #'(lambda x y) x) a b)
```

It simply takes the union of the two xappings, taking the value of the first when an index is defined in both. The function `in`, defined in terms of α, intersects the domains of two xappings and takes the values from the first. Other functions include `shift`, which subtracts a specified integer from all the indices of a xector. In our example program, we use the functions `outer-product` and `transpose`, which work as their names suggest.

3.2.1 Discussion: Why xappings are not enough

Processor selection is implicit in CM Lisp. By operating on a particular data structure, one implicitly selects the relevant processors. Therefore, it is relatively easy to use an applicative style in CM Lisp. In addition, the ability to create new xappings containing portions of old xappings and passing them as arguments means that programs can be structured in a modular way. These claims are supported by the example program in Figure 3. (The input matrix to the `lu-decomp` function is a xector of xectors, whose inner xectors form the rows of the matrix.)

Xappings are intented to hide the machine model. The programmer thinks only in terms of abstract operations on xappings. An unfortunate side-effect of this particular design, however, is that things that should be easy to do on the Connection Machine become very difficult and expensive in CM Lisp. Selecting groups of processors for in-place parallel operations is one of the strengths of the Connection Machine. Yet most of the code in our example algorithm is concerned with manipulating the xappings in order to extract the right pieces of data before operating on them, and then putting the computed data back in the right place. This makes it difficult to glance at the program and find the underlying algorithm.

Take a deep breath and consider the `norm` routine in Figure 3. The function takes an index i, a size n, and a square matrix *mat* (organized as a xector of rows), and is supposed to multiply

```
(defun mask (i n)
       (shift (iota (- n i)) -i)))
(defun bigger (x y)
       (if (< (abs (car x)) (abs (car y))) y x))
(defun max-row (i n mat)
       (let ((col (αlist (xref (transpose mat) i) (iota i))))
                 (cadr (βbigger (in col (mask i n))))))
(defun swap(i n mat)
       (let* ((rowi (xref mat i))
              (k (max-row i n mat))
              (rows (if (eql i k) rowi
                       (β@ '{,i→,k ,k→,i} (xunion #'@ rowi (xref mat k)))))
                 α(over •(αin rows (mask i)) •mat))))
(defun norm (i n mat)
       (let* ((v (/ -1 (xref (xref mat i) i)))
              (tmat (transpose mat))
              (coli (xref tmat i))
              (newcol (over (α* (in coli (mask (+ 1 i) n)) αv) col)))
                 (transpose (over '{,i→,newcol} tmat)))
(defun update (i n mat)
       (let ((m (mask (+ i 1) n))
             (a (in (xref mat i) m))
             (b (in (xref (transpose mat) i) m)))
                α(xunion #'+ •(outer-product a b) •mat)))))
(defun lu-decomp (matrix n)
       (let ((mat matrix))
            (do ((i 0 (+1 i)))
                (eql i (1- n)
                    (mat)
                (setq mat (update i n (norm i n (swap i n mat)))))))
```

Figure 3: LU Decomposition Program in Connection Machine Lisp.

the elements in column i below the main diagonal by $-1.0/mat[i,i]$. First, two levels of indexing are required to obtain $v = mat[i,i]$. Then, mat is transposed in preparation for extracting the i^{th} column as variable $coli$. Next, a mask xapping is computed for intersection with $coli$ to produce a xapping containing only the elements below the main diagonal. The real work of multiplication is then done. (One could easily miss the $\alpha*$ and αv in the fourth line.) The resulting xapping is placed `over` the old column to form *newcol*. Finally, a nested xapping of one element is created to place the *newcol* `over` the transposed matrix to form a new matrix, which is then re-transposed and returned. Besides being awkward for the programmer, these manipulations are expensive at run-time. The program performs two `transpose` operations, two `over` operations, one mask creation, and one mask application. The `over` operations could be inexpensive, provided that the pairs of xappings are aligned properly. However, CM Lisp makes no guarantees about locality. And any high-level implementation of `transpose` would certainly incur communication costs. Just creating the mask takes two $O(\lg n)$ steps (iota and shift). And communication cost is incurred in order to apply it. Thus, language problems cause an algorithmic step that should take constant time to actually take $O(\lg n)$ time.

Another problem with the xapping abstraction is that multidimensional structures must be created by nesting. This makes operating on an entire structure rather awkward. The programmer must keep track of how many levels deep the α operator should be applied. For example, consider these four matrices, each represented as xappings of xappings:

$$A = \begin{bmatrix} & & \\ & e & f \\ g & h & \end{bmatrix} \quad B = \begin{bmatrix} a & b & c \\ d & & \\ & & i \end{bmatrix} \quad C = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad D = \begin{bmatrix} a & b & c \\ & e & f \\ g & h & \end{bmatrix}$$

A programmer might write `(over A B)` intending to compute matrix `C`, but would actually get matrix `D`. (The correct statement would be (α`over A B`).)

Luckily, the problem is not that abstractions are a bad idea for parallel languages, but that the xapping is not a flexible enough abstraction. One could imagine a language like CM Lisp that allowed one to avoid the masking problem as follows. Suppose one could α-apply a function to a xapping such that the results of the function were conditional on the *index* of its argument. A function of the form "if index > i, then ... else ..." could help to solve the problem we saw above. In addition, one might provide tools for creating more complex data structures with user-defined access methods and locality properties. Such ideas are explored by the next language we consider.

3.3 Paralation Lisp

The Paralation Model [8] is intended to be a machine-independent model of programming, but we consider it here only in terms of the Connection Machine. The abstraction mechanism provided by Paralation Lisp [8, 9] is called the *paralation*, a contraction of "parallel" and "relation." A paralation consists of a number of *sites*, which are numbered sequentially from zero, and a number of *fields*. The paralation's *length* (number of sites) is fixed when the paralation is created, but fields can be created dynamically. A field has one value (of any type) at each site. Every paralation has an index field, which contains the site-id at each site; this field is created and returned by the **make-paralation** function, which takes a length as its argument. Additional fields can be created with the **elwise** function, which takes a list of fields of a paralation, performs an element-wise computation on those fields, and returns the result as a new field in that paralation. The lines

```
(setq my-par (make-paralation 5))
(setq times2 (elwise (my-par) (* my-par 2)))
(setq sum (elwise (times2 my-par) (+ times2 my-par)))
```

produce the following paralation:

my-par	times2	sum
0	0	0
1	2	3
2	4	6
3	6	9
4	8	12

Aggregation is performed with the **vref** function, which applies a combining function to the elements of a paralation field and returns the result. Communication is accomplished by the function **<-** ("move") which takes up to four arguments: a source field, a combining function, a default field, and a *mapping*. A mapping is a collection of directed arrows from the sites of the source field's paralation to the sites of a destination paralation. There is no restriction on the number of arrows originating from or arriving at a given site. The move function creates a new field in the destination paralation specified by the mapping, and then sends the data in the source field to the sites of the new field according to the pattern of the mapping arrows. If more than one arrow emanates from a given site, the data from the source field is sent to all the corresponding destination sites. If more than one arrow arrives at a given site, the data is combined according to the specified combining function, which can be arbitrary. If no arrow arrives at a given site, the data at that site in the default field is used. In general, a move operation requires $O(\lg n)$ time. Note that `<-` is more general than β communication of CM Lisp because a data item can be sent to more than one place.

A mapping is most commonly created with the `match` function, which takes a source and a destination field, and produces an arrow from a source site p to a destination site q if and only if the source field at p contains the same value as the source field at q. This can be quite useful for symbolic computation (e.g., dictionary lookup). Producing a mapping with `match` involves sorting and requires $O(\lg n)$ time. Special-purpose mappings may also be created with the functions `choose` and `collapse`, which create new paralations when used with `<-`.

The Paralation Lisp *shape* facility permits the optimization of commonly-used communications patterns and the definition of natural coordinate systems for accessing paralations. The `make-shaped-paralation` function takes a list of mappings (from a field to itself) and returns a new paralation of the appropriate length. For example, a ring-shaped paralation might have mappings to rotate a field one position in the clockwise and counter-clockwise directions. These mappings can later be used by supplying an index to the `shape-map` function. In addition to mappings, access functions may be associated with a paralation. The standard Common Lisp function `elt` is used to access paralation fields by site-id. If he desires, the programmer can define an `fref` function for a paralation to perform access in different coordinate system, and may supply a print function to print out the fields of a paralation differently. In this way, a grid-shaped paralation may be accessed using row and column indices and may be printed in a form reminiscent of its shape. While shape mappings are supplied when a paralation is created, the access functions are added to existing paralations with `define-shape-access`, which may also be invoked on an unshaped paralation. Implementations of Paralation Lisp may provide libraries of functions for creating paralations of various shapes.

Paralation Lisp has a notion of *locality.* Field entries for the same site in the same paralation are considered to be *near* each other. In addition, the sites connected by mappings supplied for a shaped paralation are considered to be near each other. The compiler may lay out the paralation on the machine so that the arrows of the mappings are "short." Data in different paralations are considered to be *far.*

3.3.1 Discussion: Why too much power can hurt

A Paralation Lisp program for LU decomposition is shown in Figure 4. The input matrix is assumed to be a grid-shaped paralation. The program is relatively easy to read and understand, largely because of the ability to reference the matrix using a natural coordinate system. Even though the program makes no use of predefined grid-shaped mappings for communication, it benefits from the locality information supplied by those mappings.

A close look at the example program reveals only one use of `<-` and no use of `match`, even though the algorithm does quite a lot of communication. The reason is efficiency. Rather than use these expensive operators, `fref`s are used to accomplish communication. Although this (hack) seems against the grain of the paralation philosophy, it makes the program run significantly faster. Consider the following alternative version of the swap routine.

```
(defun alt-swap (i row col mat)
(defun bigger (a b)
       (if (> (abs (second b)) (abs (second a))) b a))
(defun max-row (i row col mat)
       (let ((col-i (elwise (row col) (logand (>= row i) (= col i))))
             (pairs (<- (elwise (row mat) (list row mat)) :by (choose col-i))))
                (vref pairs :with #'bigger :else 0)))
(defun swap (i row col mat)
       (let ((k (max-row i row col mat)))
           (elwise (row col) (cond ((logand (= row i) (>= col i)) (fref mat k col)
                                   ((logand (= row k) (>= col i)) (fref mat i col))
                                   (t (fref mat row col)))))))
(defun norm (i row col mat)
       (let ((t (/ -1.0 (fref mat i i))))
                (elwise (row col mat) (if (logand (= col i) (> row i)) (* mat t) mat))))
(defun update (i row col mat)
       (elwise (row col mat) (if (logand (> col i) (> row i))
             (+ mat (* (fref mat row i) (fref mat i col))) mat)))
(defun lu-decomp (values)
       (let* ((mat (elwise (values) values))
             (n (sqrt (length mat)))
             (self (site-names mat))
             (row (elwise (self) (first self)))
             (col (elwise (self) (second self))))
            (do ((i 0 (1+ i))) (= i (sqrt (length mat))) mat
             (setq mat (update i row col (norm i row col (swap i row col mat)))))))
```

Figure 4: LU Decomposition Program in Paralation Lisp. The functions swap and alt-swap are interchangable.

```
(let* ( (k (max-row i row col mat))
        (newrow (elwise (row col) (cond ((logand (= row i) (>= col i)) k)
                                        ((logand (= row k) (>= col i)) i)
                                        (t row))))
        (origin (elwise (row col) (list row col)))
        (dest (elwise (newrow col) (list newrow col))))
            (<- mat :by (match dest origin))))
```

The routine is longer than the one in the example because it needs to set up the key fields for the match. To compute the relative speeds of the two functions, assume that our matrix exactly fits on the Connection Machine, with one element per processor. The match function, which produces a canonicalized mapping by means of two sorting steps, takes roughly 60 milliseconds (the time for two sorts on the Connection Machine [13]). In general, the `<-` operation also performs a sort, but we will assume that the compiler is smart enough not to sort if a combining function is not supplied. It takes 260 to 820 microseconds for all processors to send one message to some other processor. The `frefs` in the example program would probably be implemented as a *get* which translates roughly to two *send* operations. Since there are two sets of `frefs` in `swap` that actually cause communication[4],

[4]Since mat is not an `elwise` variable, `fref` must be used in the default case of the conditional, even though no communication occurs.

Language	*Lisp	CM Lisp	Paralation Lisp
Parallel Variables	pvar	xapping, xector, xet	paralation field
creation	*defvar, *let, !!	iota, α	make-paralation, elwise
structure	*fixed* linear or grid	nested xappings	user-defined shape
access	pref, pref-grid	xref	elt, fref
coordinates	fixed index	xapping domain	user coordinates
Processor Selection	*all , *when , etc.	implicit	implicit
Parallel Computation	!!	α, $\bullet$	elwise
Communication	pref!!, pref-grid!!	$(\beta f\ d\ x)$	`<-` with mapping
Aggregation	scan!!, scan-grid!!, *max, *min, etc.	$(\beta f\ x)$	`<-` with mapping, vref, choose
combining function	fixed set	arbitrary	arbitrary

Figure 5: Language Summary Table

the entire communication takes about 2 milliseconds. Therefore, the `alt-swap` routine would run *thirty* times more slowly than `swap`!

So, efficient Paralation Lisp programs are characterized by (1) working on entire data structures (as opposed to passing portions of them as new paralation to more general routines) and (2) using `fref` for communication (as opposed to the general communication primitives provided by the language). Programmers are forced into this ugly style for two related reasons. The first is that `match` and `<-` are too powerful, and therefore too expensive for many common communication patterns. The second is that creation of new paralations for working on portions of data is more expensive than doing the computation in place. The communication for creating new paralations and then placing the computed results back into the original data structure are high, especially since the language regards the new paralations as "unrelated" to (and therefore *far* from) the original data structure.

3.4 Comparisons

All three languages we have explored provide a parallel data structure and provide operators for element-wise computation, global communication, and aggregation. (See Figure 5.)

The explicit processor selection of *Lisp makes programs efficient. However, explicit processor selection together with the lack of an abstraction mechanism make it difficult to write modular programs, because portions of data cannot be conveniently passed as new structures to procedures. These difficulties are somewhat relieved by Connection Machine Lisp, which uses the xapping abstraction to permit implicit processor selection. Although it is possible to pass portions of data as new structures, it is quite awkward to specify the right portion to use. This is because multi-dimensional data structures can only be created by nested xappings. In addition, creating new data structures make one a bit nervous, because the language makes no guarantees about the locality relationships between different data structures. In Paralation Lisp, the multi-dimensional access problems are solved by the *shape* facility. And the notion of *locality* permits the compiler to efficiently distribute a data structure on the Connection Machine processors. A desire to have a minimal number of primitives forces the communication operators of Paralation Lisp to be too general and expensive for many situations. Also, efficiency is sacrificed for modularity, since creating new data paralations as arguments to procedures and then replacing the results in the original structure is expensive.

At this point, a scientist/programmer would justifiably choose *Lisp out of an unwillingness to sacrifice speed for modularity and abstraction. The decision would not be not easy, though, because it is desirable to write modular programs that use libraries of general-purpose routines. In the next two sections, we propose enhancements to Paralation Lisp that permit one to write programs that are both efficient and modular.

4 Fast Mapping Creation

We have seen that the basic communication mechanism in Paralation Lisp involves creating a mapping by means of a `match` on two fields. Computing a mapping this way is expensive, requiring $O(\lg n)$ time, where n is the length of the larger paralation. This cost is particularly disconcerting when compared with the time actually required to move the useful data, which is a constant when the network is uncongested.

To make matters worse, in many computations, the overhead caused by the expensive match is not neccessary because the destination site for each data value can be computed locally at the source. In such cases, it is possible to get around this problem by using `setf` and `fref` instead of using `<-` and `match`. (Recall the two versions of the Paralation Lisp swap routine.) But this solution does not admit a combining function. A solution more in the spirit of the Paralation Model would be to provide an additional function for creating a mapping, in which the destination sites are computed locally. We propose the function

```
(fast-map dest-field source-field),
```

where `dest-field` is a field in the destination paralation d, and `source-field` is a field in the source paralation s such that each field element contains a site-id in the destination paralation. This function produces a mapping (in constant time) with the obvious properties: When the mapping is used in conjunction with `<-`, it causes the data from the given field in s to be copied to the sites in d according to the site ids in the `source-field` at the time the mapping is created.

The combining function and default fields of `<-` retain the same meanings as when `<-` is applied to a mapping created with `match`. The usual procedure for using `fast-map` would be to create the `source-field` with `elwise`. To facilitate fast mapping creation for shaped paralations, the shape designer might supply a `site-ref` function, which takes a shaped paralation and access coordinates as arguments and computes the corresponding site-id. Note that `fref` could be trivially implemented on top of `site-ref`.

5 Paralation Views

We have seen that performing an aggregate operation on only a portion of a paralation field is rather difficult and expensive, especially when side-effects are desired. One cannot pass a portion of a paralation field as the argument of a procedure without creating a new paralation of smaller size, moving the data to that smaller paralation, and then moving the data back to the original paralation after the procedure returns.

Of course, one *is* allowed to nest paralations, but this nesting has a fixed structure, and often one likes to view the data in different ways at different times. For example, one might arrange a matrix as a nested paralation, whose elements each contain a paralation with the elements of that row. This makes operating on a particular row relatively easy. But working with a particular column becomes very difficult! We saw a similar problem with Connection Machine Lisp.

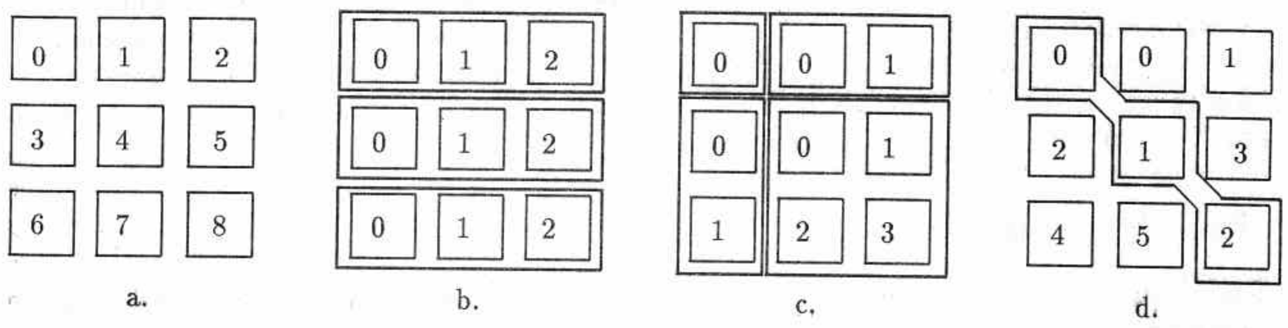

Figure 6: Three views of a matrix.

In this section, we propose *Paralation Views*, an enhancement to Paralation Lisp that allows a given data set to be viewed in multiple ways.

5.1 What is a View?

We define a *view* to be a partition of the sites of a paralation, called its *parent*, into a set of *classes*. We represent a view as a nested paralation, whose elements are the classes of the partition. Each class paralation has as its sites the corresponding sites of the parent paralation (renumbered from zero). In the existing Paralation Lisp language, one can partition the data of a paralation into several ordinary paralations using `<-` with `choose`. However, there are two important distinctions between views and ordinary paralations.

- Semantic distinction: *A class shares portions of the parent's fields with the parent paralation.*
- Locality distinction: *The sites of a class are located on the same physical processor as the corresponding sites of the parent paralation.*

These properties of a view permit many common operations to be easily performed *in place*, without the communication overhead that would result if ordinary paralations were used. Notice that the second property may result in the sites of a shaped class being physically farther apart than one might like because of the physical arrangement of the parent paralation. Section 7.2 discusses one method for alleviating this problem.

Figure 6 shows three possible views of a 3×3 matrix. The sites in the classes of each view would be located on the same processor as the corresponding matrix elements of the parent paralation, and would share the corresponding field entries at those sites. Note that the site numbers of class elements are renumbered from zero, and that classes may have a different shape from the parent.

5.2 Creating Views

In this section, we explain the methods by which views are created, introducing syntax as necessary. Views may be created in three different ways, *project*, *split*, and *extract*. Several views of the same paralation may exist simultaneously. In addition, since views and classes are both paralations, one could conceivably create veiws of them, as well.

5.2.1 Project

For shaped paralations, one may *project* on a coordinate (or set of coordinates). The classes of the view contain elements whose value(s) for that coordinate (or set of coordinates) are equal. For example, the line

```
(setq rows (project mat '(t nil) 'ring))
```

projects on the first coordinate of a grid-shaped paralation `mat` to get a view `rows` whose classes are the rows of the grid (as in Figure 6b). The second argument is a list of booleans indicating on which coordinates to project. The third argument is the desired shape type for the resulting classes. For the moment, one may think of the shape type as simply defining a set of mappings and access methods (print function, `fref`, etc.) for the resulting classes. Later, we will see how the shape type of a view might be used as locality information by the compiler. To refer to the third row of *mat*, one would write `(elt rows 2)`. When multiple coordinates are projected, `fref` is used to access particular classes of the view.

Projection is useful for working on slices of multidimensional structures. For example, one might like to filter and normalize each frame of a digitized movie (projecting on z) and then apply a temporal filter to each pixel (projecting on x and y).

5.2.2 Split

A second way to create a view of a paralation is to *split*[5] it according to subranges of the access coordinates. For example, consider

```
(setq mygrids (split ten-mat '(3 7) '(2 4 6) 'rectangle))
```

where `ten-mat` is a 10 by 10 grid-shaped paralation. The resulting view, `mygrids`, partitions `mat` into 12 classes according to three subranges of row coordinates (0..2, 3..6, 7..9) and four subranges of column coordinates (0..1, 2..3, 4..5, 6..9). To access a particular class, one uses `fref` with indices corresponding to the subrange in each dimension. For example, `(fref mygrids 2 2)` refers to the class of `mat` whose row coordinates are in the range 7..9 and whose column coordinates are in the range 4..5. The view in Figure 6c could be produced by `(split mat '(1) '(1) 'rectangle)`.

Splitting is particularly useful for divide and conquer algorithms. Some parallel divide and conquer algorithms for image processing are described in [12].

5.2.3 Extract:

The most arbitrary (and most expensive) way to create a view is to *extract* it by specifying a field of non-negative integers, and assigning all sites having the same value in that field to the same class. For example, suppose that `values` is a field of integers and `prime` is a primality testing function, and consider the following statements.

```
(setq decider (elwise (values)
                  (cond ((< values 0) 7) ((prime values) 2) (t 6))))
(setq foo (extract decider 'unshaped))
(setq positive-primes (elt foo 2))
```

[5]Not to be confused with the split operation of [3].

The unshaped[6] paralation `positive-primes` contains all sites whose `values` field contains a non-negative integer satisfying the `prime` tester. Notice that the values used to access the different classes of `foo` are the values from the `decider` field. Of course, later changes to that field do not affect class membership in `foo`. Also, note that the length of an extracted view is one greater than the maximum value in the field supplied to the `extract` function, even though some of the classes may be empty. Here, the length of `foo` is 8, and `(elt foo 5)` is a paralation of length zero. The view in Figure 6d could be produced by

```
(let (decider (elwise (site-names)
      (cond ((= (first site-names) (second site-names)) 0) (t 1))))
  (extract decider 'unshaped)
```

As an example application, consider a map of climate data represented as a paralation, where fields contain information such as altitude, average temperature, and average annual rainfall. One might use ranges of some variables to slice up the map into regions, and then apply aggregation functions to each slice in parallel.

5.3 Operating on Views

We have said that views share fields with the parent. Given a class of a view, one can access the corresponding portion of some field of the view's parent using the function **take**, which we now define with an example. Suppose that the view paralation `my-view` has `my-data` as one of the fields of its parent. Then the line

```
(elwise (take my-data (elt my-view 3)) (analyze my-data))
```

causes the `analyze` function to be applied element-wise to the data elements corresponding to the fourth class of my-view. One might also write

```
(elwise (take my-data (elt my-view 3)) (setq my-data (analyze my-data)))
```

to update the values of my-data in place. Another useful way to use views is to operate on each of the classes in parallel. For example,

```
(elwise (my-view) (vref (take my-data my-view) :with #'max))
```

returns a paralation containing the maximum data item in each view. Note that operations on the elements of a class are performed within the coordinate system of that class, and not in the coordinate system of the parent. This is one reason that views are so expressive, as we will see in the next section.

6 LU Decomposition Revisited

Figure 7 contains an LU decomposition program written using Paralation Views and `fast-map`. Unlike the programs we have seen thus far, this LU decomposition program is recursive. It was easy to write this way because the view facility permits a greater degree of modularity than we have seen in any of the other languages. Subroutines can work within their own coordinate systems, yet it is not awkward to call them on selected portions of data. It is not necessary to replace the data

[6] One would expect that classes created by `extract` would be unshaped. However, like `project` and `split`, we permit a shape to be specified. In all three cases, it is the responsibility of the programmer to ensure that the dimensions of the resulting classes conform to the specified shapes.

```
(defun bigger (a b)
       (if (> (abs (second b)) (abs (second a))) b a))
(defun max-entry (vector)
       (vref (elwise ((idx (index vector)) vector)
             (list index value)) :with #'bigger :else 0)))
(defun swap (field1 field2)
       (let ((temp field1))
            (setq field1 (<- field2 (fast-map field1 (index field2))))
            (setq field2 (<- temp (fast-map field2 (index field1))))))
(defun update (mat left top)
       (let* ((self (site-names mat))
              (row (elwise (self) (first self)))
              (col (elwise (self) (second self))))
             (setq mat (elwise (row col mat) (+ mat (* (elt left row) (elt top col)))))))
(defun lu-decomp (mat)
       (if (= (length mat) 1) mat
             (let* ((rows (project mat '(t nil) 'unshaped))
                    (cols (project mat '(nil t) 'unshaped))
                    (quad (split mat '(1 1) 'grid))
                    (max-pair (max-entry (elt cols 0))))
              (swap (take mat (elt rows 0)) (take mat (elt rows (cadr max-pair))))
              (elwise (vector (take mat (fref quad 1 0)))
                    (setq vector (* v (/ -1.0 (car max-pair)))))
              (update (take mat (fref quad 1 1)) (take mat (fref quad 1 0)) (take mat (fref quad 0 1)))
              (lu-decomp (take mat (fref quad 1 1))))))
```

Figure 7: LU Decomposition Program using Paralation Views.

after procedure calls because operations can be performed in place. Note the generality of `swap` routine, which simply takes two paralation fields and swaps their values. Also, the `max` routine simply takes a vector, without extra indices. It is easy to see how a library of useful operations on vectors, matrices, etc. could be exploited using views.

The frequent appearance of `take` in programs is somewhat annoying. One might imagine a shorthand `elwise` notation in which the first element-wise variable set the context for the rest of the elwise. Recalling the example of the previous section, the two lines

```
(elwise (take my-data (elt my-view 3)) (setq my-data (analyze my-data)))
(elwise ((elt my-view 3) my-data) (setq my-data (analyze my-data)))
```

would be equivalent. Using the standard elwise naming shorthand, one could also assign a variable name to the class, and refer to its indices element-wise in the computation.

Of course, all of this expressive power is only useful if views can be implemented efficiently.

7 Implementation

In this section, we suggest a possible implementation strategy for Paralation Views. We first describe data structures for supporting the required view operations. Then we discuss a way in which adding standard views to the shapes library can provide additional locality information to the compiler.

7.1 Data Structures

An important consideration in designing an implementation for Paralation Views is that a single view creation may result in a large number of new paralations. For example, projecting on one dimension of an $n \times n$ matrix would create $n+1$ new paralations. A naive approach might create $n+1$ new data structures (serially) on the front-end to keep track of these paralations, but this could cause problems with the asymptotic complexity of algorithms. An otherwise $O(\lg n)$ algorithm could take $O(n \lg n)$ time. A similar problem would occur in a divide and conquer algorithm using splitting to create paralations.

For the above reasons, we would like to keep view creation as cheap as possible, even if this means storing more information on the processing elements. As it turns out, storing more information on the processors also makes implementation of element-wise operations on classes relatively simple and speeds communication within classes. This is especially important when many classes are being operated on in parallel (see Section 8). The front-end must store enough information about a view to select particular classes or particular elements of classes, and it must be able to identify the fields of the parent of a view. In addition, the processors must store enough information so that class sites can efficiently locate other sites within their class (for sending or getting data).

The current implementation of Paralation Lisp uses *segments* [3] to represent nested paralations. This successfully minimizes the information stored on the front-end and is useful for scan operations. However, segments would only apply to views if each class of a view consisted of contiguous sites of the parent. Since this is not the case, we need a more general strategy. Suppose that for each view, the frond-end stores a single record containing the name of the parent and a representation of the information used to create the view. For projection, the list of projected coordinates (e.g., `(t nil)`) would be kept. For splitting, the list of subranges would be kept. And for extraction, the maximum value in the decider field would be kept. In addition, the front-end would keep pointers to three new fields of information to be used locally by the processors, as follows:

- Indicator field: This identifies the class to which the given site belongs. For projections, this is a list of that site's values for the coordinates being projected on. For splits, this is the corresponding subrange of the coordinates. And for extractions, this is a copy of the decider field.

- Site field: This is the site number within the view. For projections and splits, this can be easily calculated from the parent site number and the information in the indicator field, given that shape designers supply certain functions described below. For extract, filling this field would require an enumeration operation for each class. We do not expect extraction to be inexpensive if the number of classes is large.

- Length field: This is the length of the class of which the site is a member. For projections and splits, this can be computed directly from the indicator field. For extraction, this would have to be broadcast from the front-end.

The paralation in Figure 8 shows the indicator, site, and length fields for the views in Figure 6.

To support the necessary calculations for filling in these fields when the parent paralation is shaped, we require that shape access methods include three additional functions. The first function takes a site-id and range of user coordinates and produces a new site-id as if the sites in that range were numbered from zero. (Note that this can be used for projections as well as splits.) The second function is the inverse of this. The third function simply takes a range of user coordinates and returns the number of sites in that range. For reasonable shapes, these functions would be quite simple and efficient.

site	site-names	ind-b	site-b	len-b	ind-c	site-c	len-c	ind-d	site-d	len-d
0	(0 0)	0	0	3	(0 0) (0 0)	0	1	0	0	3
1	(0 1)	0	1	3	(1 2) (0 0)	0	2	1	0	6
2	(0 2)	0	2	3	(1 2) (0 0)	1	2	1	1	6
3	(1 0)	1	0	3	(0 0) (1 2)	0	2	1	2	6
4	(1 1)	1	1	3	(1 2) (1 2)	0	4	0	1	3
5	(1 2)	1	2	3	(1 2) (1 2)	1	4	1	3	6
6	(2 0)	2	0	3	(0 0) (1 2)	1	2	1	4	6
7	(2 1)	2	1	3	(1 2) (1 2)	2	4	1	5	6
8	(2 2)	2	2	3	(1 2) (1 2)	3	4	0	2	3

Figure 8: Fields for three views of a matrix.

Given the above fields and related functions, it is easy to see how processor selection, logarithmic scans, and interprocessor communication could be handled efficiently for projected views and split views. Interprocessor communication for extracted views, however, would be expensive even if the pattern of communication within the classes is known. This is because there are no "ranges" that can be supplied to the above functions to quickly find other elements of a class in terms of the parent's coordinate system. One would have to use the `match` and `<- move` operators, or (worse) an implementation of `fref` using associative lookup.

7.2 The Shape Library and View Locality Information

From both a user-interface point of view and an efficiency point of view, it would be useful to associate commonly used views with shapes in the same way that commonly used mappings are currently associated with shapes. We would modify the function `make-shaped-paralation` to take two arguments, a list of mappings and a list of views. The views could be accessed using a function `shape-view` in the same way that mappings are now accessed with `shape-map`.

If the compiler knows that a certain shape will be viewed in certain ways, it can make use of the locality information provided by the mappings associated with the classes of the views. For example, if the compiler knows that a matrix is going to be viewed as a collection of rings using a projection on x, it can lay out the matrix on the Connection Machine hypercube so that the first element of each column is near the last element of each column.

As a side note, we suggest that the declaration of access methods and locality information should be better integrated (perhaps using a single function instead of two) so that user-coordinates could be used to define mappings and views in a more convenient way. The current language permits access methods to be added only after the the paralation (and its corresponding mappings) have already been created.

8 Achieving Nested Parallelism

Paralation Views provides an ideal setting in which to perform parallel procedure calls on different portions of a data set. For example, we saw earlier that one can easily express parallel aggregation on all the rows of a matrix. Views are true partitions of the parent. The sites of classes do not overlap, so scans and element-wise computations on classes can proceed in parallel. But not all imaginable operations on one data set admit parallel execution on many data sets. For example, consider a procedure that, given a vector, computes the maximum value v_{max} in that vector by a scan computation and then broadcasts an instruction to divide all the elements of the vector by v_{max}. An **elwise** call of this procedure on multiple vectors would necessarily result in serialization of the second step, because the front-end would have to broadcast a *different* v_{max} to the elements of each vector.

A procedure can be executed in parallel on multiple data sets only if all immediate data contained in instructions broadcast by the front-end is constant over the parallel invocations. This leads us to a programming paradigm we call *compute-aggregate-flood* (CAF), as opposed to compute-aggregate-broadcast (CAB) of [7]. The basic idea of this paradigm is that the results of an aggregation step (scan computation) are not broadcast to the processors by the front end, but are flooded back to the processors using another scan in the reverse direction. In this way, the front-end is only responsible for issuing data-independent instructions. The programmer could write procedures using this paradigm, or possibly an optimizing compiler could convert a CAB procedure to a CAF procedure if it is recognized that the procedure is invoked in parallel. Of course, one has to be careful not to blindly convert programs in this way. When the number of parallel invocations is small, it may be more efficient to do a constant-time broadcast serially than do a logarithmic flooding operation in parallel. However, it might be possible to optimize the flooding phase using low-level instructions to the Connection Machine routers.

Most aggregation operations on the Connection Machine are accomplished with logarithmic scans, which can easily take place on multiple data sets in parallel. However, certain operations can be optimized using the "wired-OR" capability, which does not lend itself to parallel execution. Even so, there are some cases in which the "wired-OR" could be used effectively in parallel procedure calls. For example, a common use of the wired-OR is to detect termination of an iterative procedure. It could still be used to detect termination on parallel invocations of that procedure, provided that either

1. extra iterations of the algorithm are acceptable and the conditions for signalling termination are stable, or

2. processors terminate independently.

If either of these conditions hold, one simply iterates until termination is detected accross all invocations.

9 Conclusion

We have examined three languages in terms of their ability to express efficient scientific programs for the Connection Machine. We concluded that the explicit processor selection of *Lisp produces efficient but monolithic programs, while the abstraction mechanisms of Connection Machine Lisp and Paralation Lisp provide implicit processor selection and increased modularity at the expense of efficiency. The inefficiencies result from an inability to apply functions to portions of data structures without awkwardly moving the data to new structures and then moving the results back afterwards. The Paralation Lisp shape facility provides a flexible abstraction mechanism that appears to have potential for improving the efficiency of programs.

We presented Paralation Views, an enhancement to the shape facility of Paralation Lisp that permits efficient in-place computations on portions of data structures to be expressed in a straightforward manner with a small set of operators. We saw that the view abstraction integrates well with the existing Paralation Lisp language and provides excellent support for modularity and nested parallelism.

A possible approach to implementing Paralation Views was suggested, but many of the details still need to be worked out. For example, we need to study ways to efficiently handle creating views of classes, both singly and in parallel. This is particularly important when the *split* operation is used to create views in a recursive program. Other work might include integrating the access method definitions with locality and view descriptions so that mappings and views could be more easily created in terms of user access coordinates.

The Paralation Model is intended to be machine-independent. It would be interesting to see if Paralation Views is a practical idea for other architectures.

Acknowledgements

I thank my area examination committee, Barbara Liskov, Arvind, and Rishiyur Nikhil, for suggesting that I study these languages. I also thank Sally Goldman for her comments on an earlier draft, and Ken Steele for a discussion about LU decomposition algorithms.

References

[1] Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974, Chapter 6.

[2] Arvind and Ekanadham, K. Future Scientific Programming on Parallel Machines. M.I.T. Laboratory for Computer Science, Computation Structures Group Memo 272, March 1987 (revised February 1988).

[3] Blelloch, G. and Little, J. Parallel Solutions to Geometric Problems on the Scan Model of Computation. M.I.T. Artificial Intelligence Laboratory Memo 952, February, 1988.

[4] Cormen, T., Leiserson, C., and Rivest, R. *Algorithms for Parallel Computers*, Chapter 32. In progress.

[5] Hillis, W.D., *The Connection Machine*, ACM Distinguished Dissertation, MIT Press, 1985.

[6] Karp, A. Programming for Parallelism. IEEE Computer, Vol. 20, No. 5, pp.43–57 (May, 1987).

[7] Nelson, P., and Snyder, L. Programming Paradigms for Nonshared Memory Parallel Computers. From *The Characteristics of Parallel Algorithms*, Jamieson, Gannon, and Douglas, eds. MIT Press, 1987.

[8] Sabot, G. An Architecture-Independent Model for Parallel Programming. Ph.D. Thesis, Harvard University, Division of Applied Sciences TR–06–88, February 1988.

[9] Sabot, G. Paralation Lisp Reference Manual. Thinking Machines Corporation Technical Report PL87–11, May 5, 1988.

[10] Steele, G. *Common Lisp, The Language.* Digital Press, 1984.

[11] Steele, G., Hillis, W. Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. In *Proc. of 1986 ACM Conference on Lisp and Functional Programming*, pp. 279–297

[12] Stout, Q. Properties of Divide–and–Conquer Algorithms for Image Processing. In *Proc. of 1985 IEEE Workshop of Computer Architecture for Pattern Analysis and Image Database Management*, pp. 203–209.

[13] Thinking Machines Corporation. Connection Machine Model CM-2 Technical Summary. Thinking Machines Technical Report HA87–4, April 1987.

[14] Thinking Machines Corporation. *Lisp Reference Manual, Version 4.0, 1987.