Functional Implementations of the Jacobi Eigensolver

A. P. W. BÖHM[1] AND R. E. HIROMOTO[2]

[1]*Computer Science Department, Colorado State University, Fort Collins, CO 90523; e-mail: bohm@cs.colostate.edu*
[2]*Division of Mathematics, Computer Science and Statistics, The University of Texas at San Antonio, San Antonio, TX*

ABSTRACT

In this article we describe the systematic development of two implementations of the Jacobi eigensolver and give their initial performance results for the MIT/Motorola Monsoon dataflow machine. Our study is carried out using MINT, the MIT Monsoon simulator. The functional semantics with respect to array updates, which cause excessive array copying, have led us to an implementation of a parallel "group rotations" algorithm first described by Sameh. Our version of this algorithm requires $O(n^3)$ operations, whereas Sameh's original version requires $O(n^4)$ operations. The convergence of the group algorithm is briefly treated. We discuss the issues involved in rewriting the algorithm in Sisal, a strict, purely functional language without explicit I-structures. © 1996 John Wiley & Sons, Inc.

1 INTRODUCTION

A fundamental strength of functional languages is their ability to express the implementation of parallel algorithms closely following their mathematical formulation in a machine-independent fashion. This combined with their ability to express parallelism at the function, loop, and instruction levels provides strong arguments for the use of functional languages and development of functional algorithms for parallel computing. As functional languages provide a gateway to novel multithreaded machine architectures [6], they are of interest to computational scientists and designers of numerical algorithms for these machines. Id [11] and Sisal [9] are languages with such potential.

Received April 1995
Revised May 1995

© 1996 John Wiley & Sons, Inc.
Scientific Programming, Vol. 5, pp. 111–120 (1996)
CCC 1058-9244/96/020111-10

An ultimate goal of parallel programming language design should be the efficient mapping of programs onto parallel hardware. Functional languages have yet to provide general evidence that they can achieve sufficient levels of performance. Some recent results of optimizing compilation for Sisal are beginning to demonstrate this capability [3].

In this article, we present the design and analysis of a numerical algorithm, the Jacobi eigensolver, initially written in the functional dataflow language Id [1, 8, 10]. The programming constructs are functional, but we are using explicit I-structures in implementing array computations, exploiting the extra expressive power that I-structures bring to a declarative language. I-structures are data structures with built-in element-level synchronization, implemented with tag bits. Each element of an I-structure can be written only once. Element reads before writes are deferred until the write occurs. We have not used Id features that allow nondeterminism in parallel programs. In particular, mutable arrays (called M-structures in Id) that allow parallel updating of array elements, causing time dependence and hence nondeterminism, are not used. We discuss the issues involved in rewriting the algorithm in Sisal, a strict functional language without explicit I-structures.

Algorithms for eigensolvers represent an important class of numerical software typically found in standard Fortran system libraries. The Jacobi algorithm exhibits an interesting matrix calculation where the ordered update of each matrix element is governed by a sequence of previously computed updates. From the description of this algorithm given below, the computational use and organization of the data are initially seen to be a challenging task for implementation in a functional language.

We show that a functional implementation taken directly from the specifications of the numerical algorithm is marred by an intolerable amount of work caused by useless data copying required to maintain functional semantics. A second implementation, which avoids useless copying by performing more real work in one step, is of the same order of total work complexity as the original sequential algorithm and provides a higher degree of parallelism. This turns out to be an improved version of an algorithm that was designed for the ILLAC-IV [13].

2 THE JACOBI EIGENSOLVER

Given a symmetric $\mathbf{N} \times \mathbf{N}$ matrix $\mathbf{A}$, the eigenvalue problem is the determination of eigenvectors $\mathbf{x}$ and eigenvalues λ defined by

$$\mathbf{Ax} = \lambda\mathbf{x}. \tag{1}$$

Any standard reference on numerical methods [12] will provide several methods for determining the solution to this problem. One such method, known as the Jacobi algorithm, uses two-dimensional rotations applied successively to each off-diagonal element of the matrix $\mathbf{A}$. When the rotations are done systematically, $\mathbf{A}$ converges to a diagonal matrix, thereby producing both the eigenvectors and the corresponding eigenvalues. The "plane" or Jacobi rotation is described by an "orthogonal" transformation matrix $\mathbf{R}_{pq}$, in which all diagonal elements are unity except for the two elements c located at $\mathbf{R}_{pp}$ and $\mathbf{R}_{qq}$, and all off-diagonal elements are zero except for s and $-s$ located at $\mathbf{R}_{pq}$ and $\mathbf{R}_{qp}$, respectively. The rotation is defined by the values c (cosine) and s (sine) with respect to a free angular parameter ϕ. A rotation is performed by the matrix product

$$\mathbf{A}' = \mathbf{R^T}_{pq}\mathbf{AR}_{pq} \tag{2}$$

This rotation can be shown to preserve the eigenvalues of $\mathbf{A}$ and to allow for a simple recovery of the eigenvectors. A Jacobi rotation (p, q) changes only the p and q rows and columns of $\mathbf{A}$. Solving Equation 2 we get:

$$a'_{rp} = ca_{rp} - sa_{rq} \tag{3}$$

$$a'_{rq} = ca_{rq} + sa_{rp} \tag{4}$$

$$a'_{pp} = c^2a_{pp} + s^2a_{qq} - 2sca_{pq} \tag{5}$$

$$a'_{qq} = s^2a_{pp} + c^2a_{qq} + 2sca_{pq} \tag{6}$$

$$a'_{pq} = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \tag{7}$$

where $r \neq p$, $r \neq q$. The Jacobi method defines the choice of the free angular parameter ϕ such that $a'_{pq} = 0$. Given this choice of ϕ, the corresponding values of a'_{rp}, a'_{rq}, a'_{pp}, and a'_{qq} can be evaluated. It is important to notice that elements zeroed under this method are likely to be unzeroed as a result of a subsequent transformation applied to a different off-diagonal element. Fortunately, it can be shown that by systematic application of the Jacobi method, the off-diagonal elements will converge to zero. Section 5 discusses convergence issues. We now replace Equations 3–6 with

$$a'_{rp} = a_{rp} - s(a_{rq} + \tau a_{rp}) \tag{8}$$

$$a'_{rq} = a_{rq} + s(a_{rp} - \tau a_{rq}) \tag{9}$$

$$a'_{pp} = a_{pp} - ta_{pq} \tag{10}$$

$$a'_{pp} = a_{pp} + ta_{pq} \tag{11}$$

where $t = tan\phi$ and $\tau \left(= tan\frac{\phi}{2}\right)$ is defined by

$$\tau \equiv \frac{s}{1 + c}$$

Using the property that the matrix $\mathbf{A}$ is symmetric, the pattern of element updates as induced by the similarity transformation $\mathbf{R^T}_{3,5}\mathbf{AR}_{3,5}$ is depicted in Figure 1. These updated elements are denoted by a' with the $(3, 5)$ element zeroed by the appropriate choice of ϕ. When elements are zeroed in a strict

$$\begin{bmatrix}
\cdot & \cdot & a' & \cdot & a' & \cdot & \cdot & \cdot \\
 & \cdot & a' & \cdot & a' & \cdot & \cdot & \cdot \\
 & & a' & a' & 0 & a' & a' & a' \\
 & & & \cdot & a' & \cdot & \cdot & \cdot \\
 & & & & a' & a' & a' & a' \\
 & & & & & \cdot & \cdot & \cdot \\
 & & & & & & \cdot & \cdot \\
 & & & & & & & \cdot
\end{bmatrix}$$

FIGURE 1 Elemental updates induced by $\mathbf{R}^{\mathbf{T}}_{3,5}\mathbf{A}\mathbf{R}_{3,5}$.

cyclic order, the convergence of this method is quadratic for nondegenerate eigenvalues (i.e., eigenvalues that are not identical). Because the matrix A is symmetric, one *sweep* of the Jacobi method is applied to $n(n-1)/2$ distinct off-diagonal elements. Furthermore, each rotation requires $O(n)$ operations, so that the total computational complexity is of order n^3 for each sweep.

3 IMPLEMENTATIONS

3.1 A Row Major Order Implementation

In the following implementations of the Jacobi algorithm, A stands for the input matrix, D for the diagonal elements that will be converted into eigenvalues by a number of rotations, and V stands for the matrix that will be converted from an identity matrix into the matrix of eigenvectors.

A sequential implementation of Jacobi's algorithm performs *sweeps* of rotations around points in the upper triangle in row major order, until the sum of the absolute values of the upper triangle of the matrix is sufficiently small. In the following sketch of the main program, some of the details concerned with not rotating around a point that is relatively small are left out:

```
{ while
  abs_sum_up_triangle A > epsilon do
   next A, next V, next D =
   { for p <- 1 to (N-1) do
       next A, next V, next D =
       { for q <- (p+1) to N do
           next A, next V, next D =
                              Rotate A V D p q
         finally A, V, D};
     finally A, V, D}
  finally A, V, D}
```

The *while* and *for* loops in the above code have the standard meaning. A *nextified value*, such as *next A*, receives a new value for A in every loop iteration. The *finally* construct yields the last value produced in a loop. The function *Rotate* does the actual work. *Rotate* computes the values for $s = sin\phi$, $t = sin\phi/cos\phi$, and $\tau = s/(1+c)$, as defined in the previous section, and with these values it creates next versions of A, V, and D. In the following sketch of function *Rotate*, only the creation of the next value of A using Equations 8 and 9 is considered (Equations 10 and 11 are used in the computation of *next* D), and again complications considering small values are left out:

```
def Rotate A p q = { % compute s and tau
 def e8 p1 p2 = p1 - s*(p2+tau*p1);
 def e9 p1 p2 = p2 + s*(p1-tau*p2)
in {matrix ((1,N), (1,N)) of
| [i,j] = A[i,j] || i <- 1 to N; j <- 1
          to p-1
| [i,p] = e8 A[i,p] A[i,q] || i <- 1 to p-1
| [i,p] = A[i,p] || i <- p to N
| [i,j] = A[i,j] || i <- 1 to p-1; j <- p+1
          to q-1
| [p,j] = e8 A[p,j] A[j,q] || j <- p+1
          to q-1
| [i,j] = A[i,j] || i <- p+1 to N; j <- p+1
          to q-1
| [i,q] = e9 A[i,p] A[i,q] || i <- 1 to p-1
| [p,q] = 0.0
| [i,q] = e9 A[p,i] A[i,q] || i <- p+1
          to q-1
| [i,q] = A[i,q] || i <- q to N
| [i,j] = A[i,j] || i <- 1 to p-1; j <- q+1
          to N
| [p,j] = e8 A[p,j] A[q,j] || j <- q+1 to N
| [i,j] = A[i,j] || i <- p+1 to q-1;
          j <- q+1 to N
| [q,j] = e9 A[p,j] A[q,j] || j <- q+1
          to N
| [i,j] = A[i,j] || i <- q+1 to N; j <- q+1
          to N
}};
```

The above Id code uses an *array comprehension* in which the dimensionality and bounds of the array are first defined, followed by a number of *region* definitions. A region definition of the form

$$|[target] = expression \parallel generators$$

is equivalent to the loop

for generators do array[*target*] = *expression.*

A generator of the form

$$i \leftarrow L_i \text{ to } U_i; j \leftarrow L_j \text{ to } U_j$$

indicates a nested loop.

This first implementation closely follows the mathematics of the Jacobi transformation and allows for the natural exploitation of parallelism. The problem is that this algorithm is too inefficient. To update $O(n)$ elements in A, *Rotate* performs $O(n^2)$ work, most of which is just copying. This makes a sweep [involving $O(n^2)$ rotations] an $O(n^4)$ operation, which is one order of magnitude too high. A nonfunctional solution to this copying problem would be to use updatable (mutable) structures (M-structures in Id). This forces the programmer to leave the functional model of computation and rely on time-dependent and nondeterministic constructs. It is our view that these constructs should be used as much as possible at the compiler level, and as little as possible at the source code level. Examples of this approach are the *build-in-place* and *update-in-place* optimizations performed by the Sisal compiler [3].

3.2 Sameh's Parallel Group Rotations

A more parallel and at the same time more space efficient implementation of the Jacobi algorithm allows several rotations to be performed concurrently. A *group of rotations* $(p_1, q_1) \ldots (p_k, q_k)$ is *valid* if each point (p_i, q_i) occupies its own row and column in the upper triangle of A. Clearly there cannot be more than $\lfloor n/2 \rfloor$ rotation points in a group. In a parallel rotation based on all rotation points of such a group, each element in the resulting matrix is affected by at most two rotation points. A set of groups *partitions* the upper triangle of a matrix if all points in the upper triangle are included in exactly one group. In [13] Sameh defines a minimal number of $2n - 1$ groups of maximal size $\lfloor n/2 \rfloor$. These groups are essentially antidiagonals which wrap around the matrix boundaries. Sameh's group definition can be directly translated into the following function *MakePQs*:

```
def MakePQs n =
{ m = floor( float (n+1)/2.0 );
  PQs = 2D_I_array ((1,2*m-1),(1,n))   in
  { for k <- 1 to 2*m-1 do
      if k <= (m-1) then
       { for q <- (m-k+1) to (n-k) do
         p = if (     ((m-k+1) <= q)
                and  (q <= (2*m-2*k)) )
                then ((2*m-2*k+1)-q)
                else if (   ((2*m-2*k) < q)
                        and (q<=(2*m-k-1)) )
                      then ((4*m-2*k)-q)
                      else n;
            (i,j) = if p < q then (p,q)
                     else (q,p);
            PQs[k,i] = (i,j);  PQs[k,j] = (i,j)
       }
      else
       { for q <- (4*m-n-k) to (3*m-k-1) do
          p = if ( q < (2*m-k+1) )
               then n
               else if ( ((2*m-k+1) <= q)
                     and (q<=(4*m-2*k-1)) )
                    then ((4*m-2*k)-q)
                    else ((6*m-2*k-1)-q);
            (i,j) = if p < q then (p,q)
                     else (q,p);
            PQs[k,i] = (i,j); PQs[k,j] = (i,j)
         };
      {for i <- n to 2*m-1 do PQs[k,2*m-k] =
      (0,0)}
      finally PQs
  }
};
```

The following are Sameh's groups for n = 5 and n = 6:

$$\mathbf{n}=5 \quad \begin{bmatrix} . & 2 & 4 & 1 & 3 \\ & . & 1 & 3 & 5 \\ & & . & 5 & 2 \\ & & & . & 4 \\ & & & & . \end{bmatrix}$$

$$\mathbf{n}=6 \quad \begin{bmatrix} . & 2 & 4 & 1 & 3 & 5 \\ & . & 1 & 3 & 5 & 4 \\ & & . & 5 & 2 & 3 \\ & & & . & 4 & 2 \\ & & & & . & 1 \\ & & & & & . \end{bmatrix}$$

Observe that for odd n, in the example $n = 5$, Sameh's group numbers start at $\lfloor (n - 1)/2 \rfloor$ and increment (modulo n) with $\lfloor (n - 1)/2 \rfloor$ in both row and column directions, and that for even n a last column is added with groups n down to 1. We have a proof for this observation in general [2], which is beyond the scope of this article. Thus *MakePQs* can be simplified, especially if we separate the cases for odd and even n:

```
def MakeOddPQs n =
  { m = div (n-1) 2;
    PQs = 2D_I_array ((1,n), (1,n))
    in {for p <- 1 to n do
         {for q <- p+1 to n do
            h = (mod (m*p + m*q - 2*m) n );
            k = if (h == 0) then n else h;
            PQs[k,p] = (p,q);  PQs[k,q] =
            (p,q)};
          PQs[p,n+1-p] =(0,0)
          finally PQs}
  };

def MakeEvenPQs n =
  { m = div (n-1) 2; dn = n-1;
    PQs = 2D_I_array ((1,dn), (1,n))
    in {for p <- 1 to dn do
         {for q <- p+1 to dn do
            h = (mod (m*p + m*q - 2*m) dn);
            k = if (h == 0) then dn else h;
            PQs[k,p] = (p,q);  PQs[k,q] =
            (p,q)};
          PQs[p,n]=(n-p,n); PQs[p,n-p] =
          (n-p,n)
          finally PQs}
  };
```

We have left the definition of *PQs* in *MakeOddPQs* and *MakeEvenPQs* in the form of a side-effecting loop, as an array comprehension would force us to compute k twice, for index expression $[k, p]$ and for $[k, q]$.

3.3 Implementing the Group Rotations

Sameh uses the groups defined in the previous section to create an orthogonal transformation Q_k for each group, consisting of sines and cosines of the various ϕs that occupy disjoint elements of the transformation matrix, and then performs the transformation using a matrix product given in Equation 2. As there are $2n - 1$ groups, this method requires $O(n)$ matrix multiplications, which renders the complexity of one sweep to be $O(n^4)$.

We now present a new, demand-driven, implementation of the parallel group rotations algorithm that requires only $O(n^3)$ operations. Instead of forming a transformation matrix and performing a matrix product, we register with each element in the transformed matrix A' the two corresponding rotations that affect it and perform those rotations in row major order, guaranteeing that for the two elements modified by the same rotations, the rotation orders are the same.

We will derive this algorithm in two transformation steps from the row major implementation. In the first transformation step we turn the row major implementation function *Rotate* into the demand-driven form *RotDemand* in which the effect of one rotation point (p, q) is computed for each element of the result array.

```
def RotDemand A p q =
{% compute s and tau
 def e8 p1 p2 = p1 - s*(p2+tau*p1);
 def e9 f1 f2 = p2 + s*(p1-tau*p2) in
 {matrix ((1,N), (1,N)) of
 |[i,j] =
  if (j == p)
  then e8 A[i,p] A[i,q]
  else if (j == q)
       then if (i < p)
            then e9 A[i,p] A[i,q]
            else e9 A[p,i] A[i,q]
       else if (i == p)
            then if (j < q)
                then e8 A[p,j] A[j,q]
                else e8 A[p,j] A[q,j]
            else if (i == q)
                then e9 A[p,j] A[q,j]
                else A[i,j]
  || i <- 1 to N-1; j <- i+1 to N
  }
};
```

In the second step of the transformation, we allow each element of a result matrix to be affected by two rotation points as defined by Sameh's groups. For this we define a table *PQs* where row PQs_k defines the k^{th} group rotation, such that $PQs[k, i]$ and $PQs[k,j]$ contain the points affecting $A'[i,j]$. A tuple (0,0) in $PQs[k,i]$ signifies that there is no rotation in row or column i in group k. The assignments to array elements of *PQs* in the functions *MakePQs*, and also in *MakeOddPSs* and *MakeEvenPQs* accomplish the creation of *PQs*, which is constant throughout the computation, and is created once. The *PQs* arrays for n = 5 and n = 6 are:

$$\mathbf{n} = 5 \begin{bmatrix} (14) & (23) & (23) & (14) & (00) \\ (12) & (12) & (35) & (00) & (35) \\ (15) & (24) & (00) & (24) & (15) \\ (13) & (00) & (13) & (45) & (45) \\ (00) & (25) & (34) & (34) & (25) \end{bmatrix}$$

$$\mathbf{n} = 6 \begin{bmatrix} (14) & (23) & (23) & (14) & (56) & (56) \\ (12) & (12) & (35) & (46) & (35) & (46) \\ (15) & (24) & (36) & (24) & (15) & (36) \\ (13) & (26) & (13) & (45) & (45) & (26) \\ (16) & (25) & (34) & (34) & (25) & (16) \end{bmatrix}$$

A parallel group rotation now involves the computation of the s, t, and τ values associated with all points in the group, and one array comprehension defining A'. The function *GroupRot* presents this process, again with complications concerning small elements of A left out. Notice the strong similarity between *rot1* and *RotDemand*. The difference is that in *rot1* we need to create arrays of τ and s values. This takes $O(n)$ operations. Each element of A' is computed with a constant number of operations, so creating A' takes $O(n^2)$ operations. Consequently, a sweep takes $O(n^3)$. Furthermore, all n^2 elements of A' can be computed in parallel, and no needless copying is performed.

```
def GroupRot A V D PQs k N = {
 Ts, Taus, Ss = MakeTsTausSs A D PQs k N in
 { matrix((1,N),(1,N)) of
  |[i,j] =
   {p1,q1,p2,q2 =
    {p1,q1 = PQs[k,i]; p2,q2 = PQs[k,j]
    in if (p1<p2) then p1,q1,p2,q2
                  else p2,q2,p1,q1}
   in if (p1 == 0)
      then rot1 A Taus Ss i j p2 q2
      else if ((p1 == i) and (q1 == j))
           then 0.0
           else rot2 A Taus Ss i j p1 q1
           p2 q2}
   ||i <- 1 to N-1; j <- i+1 to N
 }
};
```

```
def rot1 A Taus Ss i j p q =
{ def e8 p1 p2 = p1 - Ss[p]*
                 (p2+Taus[p]*p1);
  def e9 f1 f2 = p2 + Ss[p]*
                 (p1-Taus[p]*p2) in
 if (Ss[p] == 0.0)
 then A[i,j]
 else if (j == p)
      then e8 A[i,p] A[i,q]
      else if (j == q)
           then if (i < p)
                then e9 A[i,p] A[i,q]
                else e9 A[p,i] A[i,q]
           else if (i == p)
                then if (j < q)
                     then e8 A[p,j] A[j,q]
                     else e8 A[p,j] A[q,j]
           else if (i == q)
                then e9 A[p,j] A[q,j]
                else A[i,j]
};
```

```
def rot2 A Taus Ss i j p1 q1 p q =
{ def re8 r1 c1 r2 c2 =
    {g = rot1 A Taus Ss r1 c1 p1 q1;
     h = rot1 A Taus Ss r2 c2 p1 q1
     in g-Ss[p]*(h+g*Taus[p])};
  def re9 r1 c1 r1 c2 =
    {g = rot1 A Taus Ss r1 c1 p1 q1;
    h = rot1 A Taus Ss r2 c2 p1 q1
     in h+Ss[p]*(g-h*Taus[p])}  in
  if (Ss[p] == 0.0)
  then rot1 A Taus Ss i j p1 q1
  else if (j == p)
       then re8 i p i q;
       else if (j == q)
            then if (i < p)
                 then re9 i p i q
                 else re9 p i i q
            else if (i == p)
                 then if (j < q)
                      then re8 p j j p
                      else re8 p j q j
                 else re9 p j q j
};
```

4 PRELIMINARY MONSOON PERFORMANCE

This section provides some preliminary performance results. A more complete study of the performance of the algorithms, written in Id and Sisal and running on parallel platforms, still needs to be done. The run-time behavior of Jacobi algorithms is highly data dependent. Also, the convergence rate is dependent on the order in which the rotations take place. Section 5 discusses convergence issues further. The order of the rotations differs in the two implementations, and thus the number of rotations and sweeps needed to converge may differ. This is exemplified by Table 1, which contains simulation results of both Jacobi implementations, run for a matrix A with 1.0 on the diagonal and A[i,j] = i+j off the diagonal. "Instr" stands for the number of instructions executed, "Rots" for the number of rotations performed, and "Sweeps" for the number of sweeps performed. The diagonal elements in this particular type of input matrix are smaller than the off-diagonal elements, which give rise to relatively slow conver-

Table 1. Monsoon Performance of the Algorithms

	Row Major Order		Group Rotations	
n	Instr * 1000	Rots	Instr * 1000	Sweeps (Rots)
4	133	21	84	4(24)
6	527	64	313	4(60)
8	773	72	866	5(140)
10	1769	132	1928	5(225)
12	4295	261	3367	5(330)
14	7144	359	6492	6(546)
16	11248	476	9791	6(720)

gence. Notice that the row major order algorithm performs many fewer rotations than the group rotation algorithm, but that the group rotation algorithm still executes fewer instructions most of the time (except in cases n=8 and n=10). Clearly, a more thorough study of the convergence characteristics of the two algorithms is needed; however, that is beyond the scope of this article. The next section provides an outline of the convergence proof for these algorithms.

5 NUMERICAL CONVERGENCE

The convergence of the Jacobi method is dependent on the ordering of the rotations applied to each of the off-diagonal elements of the matrix. With $(n(n-1))/2)!$ ways of choosing the updating order for the Jacobi method, it is important that a particular class of rotation orderings can be guaranteed to converge. One such ordering is the *cyclic row major ordering*, in which the first rotation in the sweep is (1,2). Forsythe and Henrici [4] have proved the convergence of the cyclic row major ordering algorithm via the following theorem:

THEOREM: Let a sequence of Jacobi transformations be applied to a symmetric matrix A. Further, let the angle ϕ_k be restricted as follows:

$$\phi_k \in [a,b] \text{ and } -\frac{\pi}{2} < a < b < \frac{\pi}{2}$$

If the off-diagonal elements are annihilated using a cyclic row major ordering, then this Jacobi method converges.

Building on this theorem, Shroff and Schreiver [14] introduce the notions of "cyclic wavefront" orderings, and "weak equivalent ordering" to prove that a modulus ordering of the form

$$\begin{bmatrix} . & 1 & 2 & 3 & 4 \\ & . & 3 & 4 & 5 \\ & & . & 5 & 1 \\ & & & . & 2 \\ & & & & . \end{bmatrix}$$

converges. In the modulus ordering, rotations are sequenced according to their group number $I(p,q)$, where $I(p,q) = ((p + q - 3) \bmod n) + 1$ for odd n, and $I(p,q) = ((p + q - 3) \bmod n - 1) + 1$ for even n. Inside a group, the rotations are sequenced according to row major ordering. Shroff and Schreiver show that, if an ordering X can be transformed into the modulus ordering by a permutation of the array indices, then X gives rise to a converging Jacobi algorithm. As an example, for N = 6, the permutation of the array indices (123456) → (135246) transforms Sameh's ordering, when taking symmetry into account, into modulus ordering:

$$\begin{bmatrix} . & 2 & 4 & 1 & 3 & 5 \\ & . & 1 & 3 & 5 & 4 \\ & & . & 5 & 2 & 3 \\ & & & . & 4 & 2 \\ & & & & . & 1 \\ & & & & & . \end{bmatrix}$$

with index permutations:

(12)	→	(13)
(13)	→	(15)
...	...	...
(35)	→	(54) → (45)
...	...	...
(45)	→	(24)
(46)	→	(26)

transforms into:

$$\begin{bmatrix} . & 1 & 2 & 3 & 4 & 5 \\ & . & 3 & 4 & 5 & 1 \\ & & . & 5 & 1 & 2 \\ & & & . & 2 & 3 \\ & & & & . & 4 \\ & & & & & . \end{bmatrix}$$

6 A STRICT FUNCTIONAL IMPLEMENTATION OF GROUP ROTATIONS

In Id's array comprehensions, the target of a certain array element expression can be explicitly specified in a region definition:

| [*target*] = *expression* || *generators*

Moreover, Id's I-structures allow array-element assignments $A[i] = v$ to be scattered over the program. Thus, in Id there is no relation between the process defining an array element and the placement of that element. As Id data structures are nonstrict, it is not even necessary that all elements of an array be defined. It is in general not decidable whether an Id array is uniquely defined, i.e., all elements are defined at most once, or completely defined, i.e., all elements are defined exactly once. For array comprehensions with linear expressions defining the array dimensions, array-element targets, and bounds of the generators, uniqueness and completeness can be checked [5].

In contrast, the strict functional programming language Sisal allows array creation in a loop only monolithically and in an implicitly defined order: loop body $(i_1, \ldots, i_n)$ of an n-deep nested loop defines element $(i_1, \ldots, i_n)$ of an n-dimensional array. For example

```
for i in l1,u1 cross j in l2,u2
 returns array of f(i,j)
end for
```

creates a two-dimensional array with bounds $((l1,u1), (l2,u2))$, and $f(i,j)$ at position (i,j). We call this the *strict loop order* property of Sisal, which statically guarantees that arrays are uniquely and completely defined. This property allows efficient array allocation and creation at the cost of some loss in expressiveness. As an example, translating the Id functions *Grouprot*, *rot1*, and *rot2* into Sisal is straightforward, whereas translating the Id function *MakePQs* into Sisal is not, as it relies on a nonmonolithic computation of one I-structure (in this case scattered over three loops), and side-effecting assignments with computation of the target index.

There are four approaches in Sisal to create arrays such as *PQs*, where there is no direct relation between the order of creation and the place in the array.

1. Devise a new implementation of the algorithm that adheres to strict loop ordering. This often requires a more thorough understanding of the algorithm being implemented, and often helps the programmer to come to a more elegant solution of the problem. It turns out that it is possible to change *MakePQs* so that it adheres to the strict loop order.
2. Create the array in a sequential loop by subsequent array updates of the form:

```
PQs := old PQs[k,i: tuple] [k,j: tuple]
```

relying on the update in place optimization of the Sisal compiler to create only one array and perform destructive updates. In this form the creation of the array is completely sequential.

3. The second approach can sometimes be made more parallel when rows (or more generally, subarrays) can be created in parallel and concatenated together, exploiting the build-in-place optimization. This can be done for *MakePQs*, as the elements are placed in the correct rows, but not in the strict loop order inside a row.
4. Create an intermediate array of (value,index) tuples in any order, followed by a forall loop that places the values in the array by searching in the intermediate array. This very general approach exploits all available parallelism, but is costly in instruction count and space usage as a search is involved and an intermediate array is built. In our case the intermediate array would have the correct row order but not the correct element order per row. Also, it would not be necessary to store the row index *i* with the element (p,q), as an element (p,q) of the intermediate array must be placed in both positions $(k,p$ and $k,q)$ of the final array PQs.

The design of the Sisal function *MakePQs* is based on the following observations.

I. If n is odd, then
 1. The branches assigning $p = n$ are never taken, as $2m - k - 1 = n - k$ and $4m - n - k = 2m - k + 1$.
 2. All other expressions in different conditional branches assigning a value to p are equal *modulo n*. The different expressions in the

various conditional branches guarantee that the value assigned to p lies between 1 and n.

3. p and q are interchangeable: $p = 2m - 2k + 1 - q$ implies that $q = 2m - 2k + 1 - p$. This coincides with our intuition: A rotation point (p,q) touches rows and columns p and q, so rows and columns play interchangeable roles. This allows us to perform the two assignments in one loop body in *MakePQs*. However, this is not necessary. We can also perform one assignment per loop body and run the loop for all points in *PQs*, which is what we need to do in a strict loop order implementation of *MakePQs*.

II. If n is even, the assignments to p are the same as for $n - 1$ except that an extra assignment $p = n$ occurs for $q = n - k$ $(4m - n - k = n - k)$. This implies that *PQs* for even n is equal to *PQs* for $n - 1$, except that it gets $(n - k,n)$ in an extra n-th column and in position $(k,n - k)$, which has (0,0) for $n - 1$.

III. All loop bodies are independent, so they can be run in any order, hence also in strict loop order.

The Sisal implementation of *MakePQs* follows.

```
 type pair = record[x,y: integer];
type table = array[array[pair]]

function compq(k,q,n: integer returns pair)
 let pp := mod (n - 2*k - q + 2, n);
     p := if pp = 0 then n else pp end if;
 in if q < p then record pair[x:q; y:p]
             else record pair[x:p; y:q]
    end if
 end let
end function

function MakePQs(n: integer returns table)
 if n = 2 * (n/2) % even
 then for k in 1, n-1 cross q in 1,n
      returns array of
       if (q = n-k) | (q = n)
       then record pair [x: n-k; y:n]
       else compq(k,q,n-1)
       end if
      end for
 else for k in 1,n cross q in 1,n
      returns array of
       if (q = n-k+1)
       then record pair [x: 0; y:0]
       else compq(k,q,n)
       end if
      end for
 end if
end function
```

The code is clearly simpler than the original *MakePQs*, and much more efficient than an implementation that builds an intermediate array with unordered rows and reorders these in a second sweep.

7 CONCLUSIONS AND FUTURE WORK

In this article we have discussed the design of a functional parallel implementation of the Jacobi eigensolver. We have demonstrated that the Jacobi method, although initially encumbered by the functional semantics of nondestructive array updates, is efficiently expressible in the functional paradigm. Without relaxing the fundamental single-assignment semantics of functional languages, an efficient implementation was possible by resorting to parallelism at the algorithmic level, so that copying of array elements could be eliminated. Nonstrictness was explicitly used in only one function of the original Id program, and this could be avoided by reordering the creation of the array elements.

In future work we will compare the actual parallel performance of the code against the performance expected from analyzing its algorithmic specification. This approach was first used as a metric for comparing the computational complexity of sequential and parallel Fortran implementations [7]. We will compare the row order and group rotation algorithms with respect to their instruction efficiency and parallelism in both Id and Sisal on stock and experimental hardware. We will study the effect on efficiency and parallelism of explicit updates in place using mutable arrays in Id for the row order algorithm and compare this to the automatic update in place optimizations available in Sisal. We will also study the effect of storing vs. recomputing elements of the *PQs* table.

ACKNOWLEDGMENTS

John Feo (LLNL) pointed out the four approaches to strict loop order array creation, and wrote the strict loop order Sisal version of the *MakePQs* function. This work is supported by a grant from Motorola Inc. and in part by NSF grant MIP-9113268.

REFERENCES

[1] Arvind, R. A. Ianucci, *Instruction Set Definition for a Tagged Token Dataflow Machine.* LCS, MIT, 1983.

[2] A. P. W. Böhm and R. E. Hiromoto, "A functional implementation of the Jacobi eigensolver," TR 93-106, Colorado State University, Fort Collins, CO, Tech. Rep. CSU TR 93-106, 1993.

[3] D. Cann, "Retire Fortran? A debate rekindled," *CACM*, vol. 35, pp. 81–89, Aug. 1992.

[4] G. E. Forsythe and P. Henrici, "The cyclic Jacobi method for the principal values of a complex matrix, *Trans. Am. Math. Soc.*, vol. 94, pp. 1–23, 1960.

[5] D. A. Garza and A. P. W. Böhm, "Uniqueness and completeness analysis of array comprehensions," *Proceedings of the First International Static Analysis Symposium*, SAS94, Namur, Belgium, Springer LNCS 864, 1994, pp. 193–207.

[6] J. R. Gurd, A. P. W. Böhm, and Y. M. Teo, "Performance issues in dataflow machines," *Future Generation Computer Systems*, vol. 3, pp. 285–297, 1987.

[7] J. J. Lambiotte, Jr., and R. Voigt, "The solution of tridiagonal linear systems on the CDC STAR-100 computer," *ACM Trans. Math. Software*, vol. 1, pp. 308–329, 1975.

[8] Motorola, Inc., *Id World User's Manual.* Motorola, MCRC, Cambridge, MA, 1992.

[9] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas, *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual,* Version 1.2., Manual M-146, Rev. 1, Livermore, CA: Lawrence Livermore National Laboratory, March 1985.

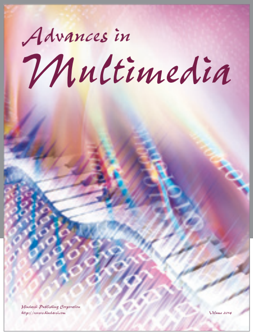
[10] D. R. Morais, ID World: An environment for the development of dataflow programs written in ID, Tech. Rep. MIT LCS TR-365, May 1986.

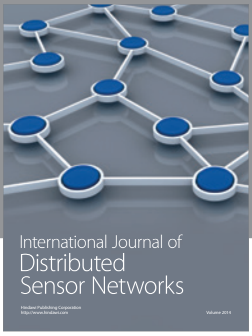
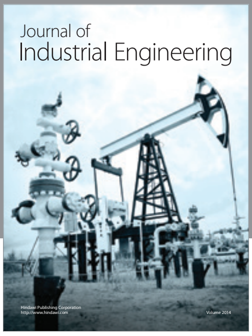
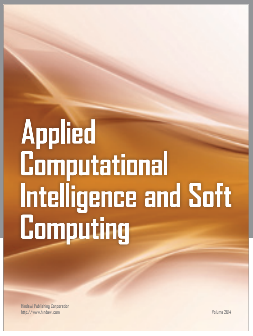
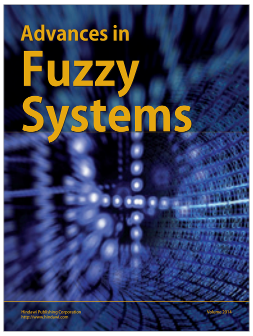
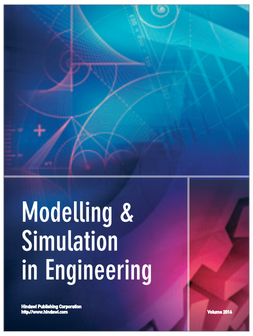
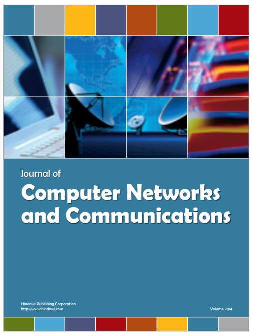
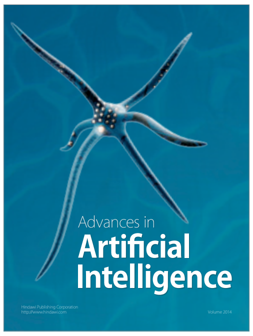
[11] R. S. Nikhil, *Id* (Version 90.0) *Reference Manual.* TR CSG Memo 284-1, MIT LCS, 1990.

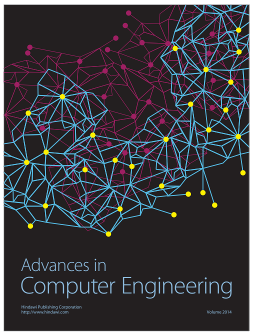
[12] W. H. Press, et al., *Numerical Recipes: The Art of Scientific Computing.* Cambridge, MA: Cambridge University Press, 1986.

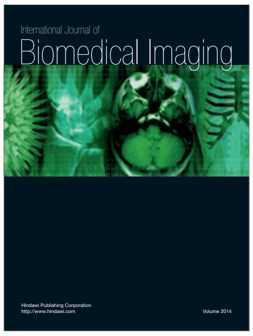
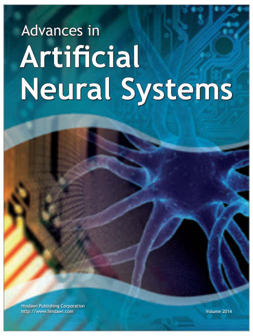
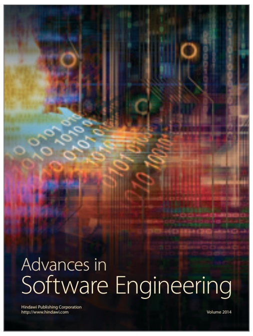
[13] A. H. Sameh, "On Jacobi-like algorithms for a parallel computer, *Math. Comput.*, vol. 25, pp. 579–590, 1971.

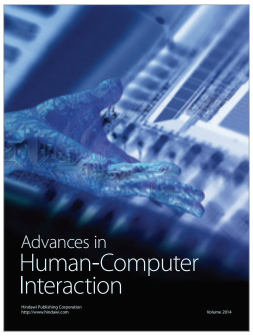
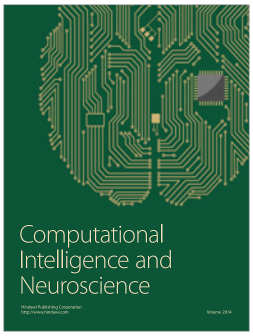
[14] G. Shroff and R. Schreiver, "On the convergence of the cyclic Jacobi method for parallel block orderings, *SIAM J. Matrix Anal. Appl.*, vol. 10, pp. 326–346, July 1989.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems

Modelling & Simulation in Engineering

Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence


Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering