A Functional Implementation of the Jacobi Eigen-Solver *

A.P.W. Böhm
Computer Science Department
Colorado State University

R.E. Hiromoto
Computer Research Group
Los Alamos National Laboratory

June 15, 1994

Abstract

In this paper, we describe the systematic development of two implementations of the Jacobi eigen-solver and give their performance results for the MIT/Motorola Monsoon dataflow machine. Our study is carried out using MINT, the MIT Monsoon simulator. The design of these implementations follows from the mathematics of the Jacobi method, and not from a translation of an existing sequential code. The functional semantics with respect to array updates, which cause excessive array copying, has lead us to a new 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 implementations are programmed in the language Id, and although Id has non-functional features, we have restricted the development of our eigen-solvers to the functional sub-set of the language.

1 Introduction

A fundamental strength of functional languages is their power to concisely express the implementation of algorithms in general, and numerical algorithms in particular, closely following their mathematical formulation. This combined with their ability to implicitly and machine-independently express parallelism at the function, loop, and instruction levels provides strong arguments for the use of functional languages and development of functional algorithms for high performance computing. As functional languages provide a machine independent and implicitly

*This work is supported by a grant from Motorola Inc. and in part by NSF grant MIP-9113268, and under the auspices of the U.S. Department of Energy under contract # W-7405-ENG-36.

parallel gateway to novel parallel machine architectures, such as multithreaded or hybrid von Neumann/dataflow, it is of vital interest to computational scientists and designers of numerical algorithms for these machines that functional languages provide expressive power and efficient implementation. Id[9] is a language with the potential to provide all of this.

It is our opinion that there is a need for careful study of the effectiveness of the functional language paradigm in expressing numerical algorithms, as the suitability of a programming language extends beyond mere elegance. As an example, it would be clumsy to have to express numerical algorithms in a functional language that lacks loop constructs and array datatypes, given that many of these algorithms are based on linear algebra. The ultimate goal is an efficient mapping of the problem specification from the language, through the compiler, onto the parallel hardware. As imperative languages are capable of providing efficient computational performance, although at the cost of programming inelegance and machine dependence, they still represent the principle programming languages for high performance computing. Functional languages have yet to provide sufficient evidence that they can achieve the same levels of performance. Some recent results of an applicative language have demonstrated this capability[4].

In this paper, we present the design and complexity analysis of a numerical algorithm, the Jacobi eigen-solver, written in the functional dataflow language Id[9]. The programming constructs are functional, but we are using explicit I-structures in implementing array computations. We have used none of the non-functional programming features provide by the Id language such as mutable-arrays.

Algorithms for eigen-solvers 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 sequences of previously computed updates. From the description of this algorithm given below, the computational use and organization of the data is initially seen to be a challenging task for implementation in a functional language. As in previous work[2], we will use the notion of *abstracted complexity* , first used as a metric for comparing the computational complexity between sequential and parallel Fortran implementations[6]. This metric will provide us with a quantitative measure of equivalence between functional and non-functional implementations.

We begin with a direct functional implementation taken from the specifications of the numerical algorithm. We show, based on our notion of *abstracted complexity* , that this direct approach is marred by an intolerable amount work caused by data copying required to maintain functional semantics. A second implementation is designed that avoids this problem and is of the same order of total work complexity as the original sequential algorithm but provides a high degree of parallelism.

The Jacobi algorithm selected is one of several methods used in solving the eigenvalues of a symmetric matrix. Surprisingly, our parallel Jacobi algorithm is an improved version of an algorithm that was designed for the ILLAC-IV [11]. The algorithm can be efficiently expressed in Id in a highly elegant fashion.

2 Description and Complexity of Jacobi Eigenvalue Solver

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

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

Any standard reference on numerical methods [10] will provide a number of methods available for determining the solution to this problem. One such method is known as the Jacobi algorithm uses two-dimensional rotations applied successively to each off-diagonal element of the matrix $\mathbf{A}$ which, when the rotations are done systematically, converges to a diagonal matrix, thereby producing both the eigenvectors and the corresponding eigenvalues. The "plane" or Jacobi rotation $\mathbf{R}_{pq}$ is described by an orthogonal transformation matrix of the form

$$\mathbf{R}_{pq} = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & . & & . & & \\ & & . & 1 & . & & \\ & & . & & . & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{bmatrix}$$

In $\mathbf{R}_{pq}$ 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 numbers c (cosine) and s (sine) with respect to the angle ϕ. A rotation is performed by the matrix product

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

can be shown to preserve the eigenvalues of $\mathbf{A}$ and allow for a simple recovery of the eigenvectors. A Jacobi rotation (p, q) changes only the p and q rows and columns of $\mathbf{A}$ as shown Fig. 1.

$$\mathbf{A}' = \begin{bmatrix} & \dots & {a'}_{1p} & \dots & {a'}_{1q} & \dots & \\ \cdot & & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot & & \cdot \\ {a'}_{p1} & \dots & {a'}_{pp} & \dots & {a'}_{pq} & \dots & {a'}_{pn} \\ \cdot & & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot & & \cdot \\ {a'}_{q1} & \dots & {a'}_{qp} & \dots & {a'}_{qq} & \dots & {a'}_{qn} \\ \cdot & & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot & & \cdot \\ & \dots & {a'}_{np} & \dots & {a'}_{nq} & \dots & \end{bmatrix}$$

Figure 1. Elemental updates induced by ${\mathbf{R}^{\mathbf{T}}}_{p,q}\mathbf{A}\mathbf{R}_{p,q}$.

Solving for Eqn. 2, we get the following set of equations:

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

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

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

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

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

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 tranformations applied to a different off-diagonal element. Fortunately, it can be shown that the systematic application of the Jacobi method to the off-diagonal elements will converge to zero. Let $t = \frac{s}{c}$, and

$$c = \frac{1}{\sqrt{t^2 + 1}},$$

$$s = tc.$$

We now replace the Eqns. 3-7 with

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

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

$$a'_{pp} = a_{pp} - t a_{pq}, \tag{10}$$

$$a'_{qq} = a_{pp} + t a_{pq}, \tag{11}$$

$$a'_{pq} = 0, \tag{12}$$

where $\tau(= tan\frac{\phi}{2})$ 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 $\mathbf{R^T}_{3,5}\mathbf{A}\mathbf{R}_{3,5}$ is depicted in Fig. 2a. These updated elements are denoted by $a^{(1)}$ with the $(3,5)$ element zeroed by the choice of ϕ. In Fig. 2b, we again depict the results of following the $\mathbf{R^T}_{3,5}$ similarity transformation with the rotations defined by $\mathbf{R^T}_{5,7}\mathbf{A}\mathbf{R}_{5,7}$. Under the later rotations, the correpsonding element updates are denoted by $(a^{(2)})$. There are two important features to note. First, the zeroed element $(3,5)$ is now rescaled to the value $a^{(0)}$ which may be different from zero. Second, the application of each similarity transformation affect the change of only $n-1$ elements.

$$
\begin{bmatrix}
. & . & a^{(1)} & . & a^{(1)} & . & . & . \\
 & . & a^{(1)} & . & a^{(1)} & . & . & . \\
 & & a^{(1)} & a^{(1)} & 0 & a^{(1)} & a^{(1)} & a^{(1)} \\
 & & & . & a^{(1)} & . & . & . \\
 & & & & a^{(1)} & a^{(1)} & a^{(1)} & a^{(1)} \\
 & & & & & . & . & . \\
 & & & & & & . & . \\
 & & & & & & & .
\end{bmatrix}
$$

Figure 2a. Elemental updates induced by $\mathbf{R^T}_{3,5}\mathbf{A}\mathbf{R}_{3,5}$.

$$
\begin{bmatrix}
. & . & . & . & a^{(2)} & . & a^{(2)} & . \\
 & . & . & . & a^{(2)} & . & a^{(2)} & . \\
 & & . & . & a^{(0)} & . & a^{(2)} & . \\
 & & & . & a^{(2)} & . & a^{(2)} & . \\
 & & & & a^{(2)} & a^{(2)} & 0 & a^{(2)} \\
 & & & & & . & a^{(2)} & . \\
 & & & & & & a^{(2)} & a^{(2)} \\
 & & & & & & & .
\end{bmatrix}
$$

Figure 2b. Elemental updates induced by $\mathbf{R^T}_{5,7}\mathbf{A}\mathbf{R}_{5,7}$.

When elements are zeroed in a strict order using Eqns. 8-12, we talk of a *cyclic Jacobi method.* It can be shown that the convergence of this method is generally quadratic for nondegenerate eigenvalues (i.e. eigenvalues that are not identical). Because the matrix $\mathbf{A}$ 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, sothat 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_upper_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 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 a next versions of A, V, and D. In the following sketch of function *Rotate*, only the creation of the next value of A is shown, and again complications considering small values are left out:

```
def Rotate A V D p q = {

% compute t,s and tau    . . .

newA = { matrix ((1,N),(1,N)) of
 |[i,j] = A[i,j]  || i <- 1 to N ; j <- 1 to p-1
 |[j,p] = {g=A[j,p]; h=A[j,q] in g-s*(h+g*tau) }
                  || j <- 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] = {g=A[p,j]; h=A[j,q] in g-s*(h+g*tau) }
                  || j <- p+1 to q-1
```

```
 |[i,j] = A[i,j]  || i <- p+1 to N ; j <- p+1 to q-1
 |[j,q] = {g=A[j,p]; h=A[j,q] in h+s*(g-h*tau) }
                  || j <- 1 to p-1
 |[p,q] = 0.0
 |[j,q] = {g=A[p,j]; h=A[j,q] in h+s*(g-h*tau) }
                  || j <- 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] = {g=A[p,j]; h=A[q,j] in g-s*(h+g*tau) }
                  || j <- q+1 to N
 |[i,j] = A[i,j]  || i <- p+1 to q-1 ; j <- q+1 to N
 |[q,j] = {g=A[p,j]; h=A[q,j] in h+s*(g-h*tau) }
                  || j <- q+1 to N
 |[i,j] = A[i,j]  || i <- q+1 to N ; j <- q+1 to N
 };
in  newA,newV,newD
};
```

Although this first implementation follows the mathematics of the Jacobi transformation closely and allows for natural exploitation of parallelism, and therefore demonstrates the power of the functional approach, the problem is that of this algorithm is too inefficient. For example, 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, and is therefore not acceptable. A non-functional approach would be to use updatable (mutable) structures (M-structures in Id). However, this would complicate the code considerably and cause loss of parallelism. Also, the implementation would loose its elegance.

3.2 An Implementation based on Sameh's parallel group rotations

A more parallel and at the same time more space efficient implementation of the Jacobi algorithm allows a number of 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 then $\lfloor N/2 \rfloor$ points in a group. In a parallel rotation based an such a group, each point in the matrix A is influenced by at most two points. A set of groups *partitions* and *covers* the upper triangle of A iff all points in the upper triangle of A are included in exactly one group. In [11] Sameh defines a minimal number of $2n-1$ groups k of maximal size $\lfloor N/2 \rfloor$. These groups are essentially anti-diagonals which wrap around the matrix boundaries. Sameh's group definition can be translated into the following loop:

```
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:

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

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

Sameh uses these groups to create an orthogonal transformation Q_k for each group, consisting of sin-s and cos-s, of the various ϕs, which all occupy disjoint elements of the transformation matrix, and then performs the transformation using a matrix product given in eqn. 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 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 for each element in the transformed matrix A' which two rotations affect it and perform the two rotations in a well-defined order, guaranteeing that for two elements affected by the same rotations, the rotation orders are the same. 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 rotation k. The array-element assignments in the function *MakePQs* accomplish the creation of PQs, which is constant throughout the computation, and needs to be created only once. A parallel group rotation now involves the computation of the s, t and τ values associated to all points in the group, and one array comprehension defining A'. The following function *GroupRot* sketches this process for the creation of the next value of A, again with complications concerning small elements of A left out. The next values of V and D are computed similarly.

```
def GroupRot A V D PQs k N  = {
 Ts, Taus, Ss = MakeTsTausSs A D PQs k N
 in % next A
 { 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 =
if (Ss[p] == 0.0)
then A[i,j]
else if (j == p)
     then {g=A[i,p]; h=A[i,q]
```

```
           in g-Ss[p]*(h+g*Taus[p]) }
     else if (j == q)
          then
           if (i < p)
           then {g=A[i,p]; h=A[i,q]
                 in h+Ss[p]*(g-h*Taus[p]) }
           else {g=A[p,i]; h=A[i,q]
                 in h+Ss[p]*(g-h*Taus[p]) }
          else
           if (i == p)
           then
             if (j < q)
             then {g=A[p,j]; h=A[j,q]
                   in g-Ss[p]*(h+g*Taus[p]) }
             else {g=A[p,j]; h=A[q,j]
                  in g-Ss[p]*(h+g*Taus[p]) }
           else
             if (i == q)
             then {g=A[p,j]; h=A[q,j]
                   in h+Ss[p]*(g-h*Taus[p]) }
             else A[i,j];


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

```
            else  {g = rot1 A Taus Ss p j p1 q1;
                   h = rot1 A Taus Ss q j p1 q1
                   in h+Ss[p]*(g-h*Taus[p]) };
```

The creation of the s, t and τ values involves $O(n)$ operations. As for each element of A' only a constant amount of operations is needed, the creation of A' takes $O(n^2)$ operations. Consequently a sweep now takes $O(n^3)$. Furthermore, all n^2 elements of A' can be computed in parallel.

3.3 Simplifying MakePQs

Observe that in the example for 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. In the case of n=6, the last column consists of the group numbers n down to 1. A general proof for this is provided in the appendix. This implies that our rather complex function *MakePQs* can be simplified and made more efficient, especially if we separate the cases for odd and even n:

```
def EvenPQs n = {
 m = floor( float n/2.0 );  mm = n-1;
 PQs = 2D_I_array ((1,mm),(1,n))
 in
 { for p <- 1 to mm do
   { for q <- p+1 to mm do
     h = (mod (q*(m-1)-p*m+1) mm);
     k = if (h == 0) then mm 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
 }
};


def OddPQs n = {
 m = floor( float (n+1)/2.0 );
 PQs = 2D_I_array ((1,n),(1,n))
 in
 { for p <- 1 to n do
   { for q <- p+1 to n do
     h = (mod (q*(m-1)-p*m+1) 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
 }
```

	Row Major Order			Group Rotations		
n	Instr * 1000	Cycles *1000	Rots	Instr * 1000	Cycles *1000	Sweeps (Rots)
4	133	8	21	84	12	4(24)
6	527	34	64	313	42	4(60)
8	773	59	72	866	130	5(140)
10	1,769	140	132	1,928	244	5(225)
12	4,295	350	261	3,367	460	5(330)
14	7,144	630	359	6,492	900	6(546)
16	11,248	1000	476	9,791	1350	6(720)

Table 1: Monsoon performance of the algorithms

```
};
```

We have left the definition of PQs in $OddPQs$ and $EvenPQs$ 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 $[k,q]$.

4 Monsoon Performance

The run time behaviour of Jacobi algorithms is highly input dependent. Most importantly, the convergence rate is 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. Table 1 contains the 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, *Cycles* stands for the number of Monsoon machine cycles the algorithms take and relates to the critical path length of the algorithms, *Rots* stands for the number of rotations performed, and *Sweeps* for the number of sweeps performed. As the diagonal elements are smaller than the off diagonal elements, this type of input gives rise to relatively slow convergence. Notice that the row-major order algorithm performs much less rotations than the group rotation algorithm, but that the group rotation algorithm still executes less instructions (except in cases n=8 and n=10). The number of Monsoon machine cycles is always less for the row major order algorithm than for the group rotations algorithm. This is an unexpected result, given the abstract complexity of the algorithms, and has to do with the way Id loops are compiled. (The default "K-bound" – the number of loops bodies that execute in parallel – is one.) By explicitly changing the loop bounds, the group rotations algorithm can be made more parallel.

5 Numercial Convergence

Given that there are $(n(n-1))/2)!$ ways of choosing the updating order for the Jacobi method, one particular ordering is important since it can be proved to converge. This particular ordering is a cyclic by row ordering, where the rotations are chosen according to the following rule. The first rotation in the sweep is $(1, 2)$. A rotation (p, q) is followed by

$$\begin{array}{cccc} (p, q+1) & if & p < n-1, & q < n, \\ (p+1, p+2) & if & p < n-1, & q = n, \\ (1, 2) & if & p = n-1, & q = n. \end{array}$$

Under this cyclic row major ordering, we have the following theorem by Forsythe and Henrici [3].

THEOREM 1. Let a sequence of Jacobi transformations be applied to a symmetric matrix A. Further, let the angle ϕ_k be restricted as follows:

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

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

It was this cyclic row major ordering that when initially implemented substantially increased the computational complexity, because of the functional array semantics of Id. This undesireable feature finally lead to the parallel cyclic ordering described above.

However, with the new ordering it is important that convergence is still guaranteed. Fortunately, Shroff and Schreiver [12] have introduced the notion of "cyclic wavefront" orderings where in a cyclic ordering of pairs

$$\{(i, j), 1 \leq j < j \leq n\}$$

the rotation index $I(i, j)$ at the which the pair (i, j) occurs. If

$$I(i, j-1) < I(i, j) < I(i+1, j)$$

for all $1 \leq i < j \leq n$, then the ordering is called a cyclic wave front ordering.

The important point is the Theorem that follows

THEOREM 2. A cyclic Jacobi ordering is equivalent to the cyclic by rows ordering if and only if it is a cyclic wavefront ordering.

Moreover, under *shift-equivalent orderings* and transpositions of commuting rotations a class of *weakly wavefront orderings* is shown to be equivalent to the cyclic wavefront ordering and thus also convergent. Finally a permutation equivalent ordering is also show to be weakly equivalent to this ordering.

$$\mathbf{N} \; = \; 6 \begin{bmatrix} . & 2 & 4 & 1 & 3 & 5 \\ & . & 1 & 3 & 5 & 4 \\ & & . & 5 & 1 & 3 \\ & & & . & 2 & 2 \\ & & & & . & 1 \\ & & & & & . \end{bmatrix} \tag{13}$$

We now consider the following permutation given by

$$(123456) \longrightarrow (135246)$$

$$\begin{array}{lllll} 1strow: & element & (12) & \longrightarrow & (13) \\ & element & (13) & \longrightarrow & (15) \\ & element & (14) & \longrightarrow & (12) \\ & element & (15) & \longrightarrow & (14) \\ & element & (16) & \longrightarrow & (16) \\ \\ 2strow: & element & (23) & \longrightarrow & (35) \\ & element & (24) & \longrightarrow & (32) \\ & element & (25) & \longrightarrow & (34) \\ & element & (26) & \longrightarrow & (36) \\ \\ 3strow: & element & (34) & \longrightarrow & (52) \\ & element & (35) & \longrightarrow & (54) \\ & element & (36) & \longrightarrow & (56) \\ \\ 4strow: & element & (45) & \longrightarrow & (24) \\ & element & (46) & \longrightarrow & (26) \end{array}$$

Under these permutations, the following cyclic ordering results:

$$\begin{bmatrix} . & 1 & 2 & 3 & 4 & 5 \\ & . & 3 & 4 & 5 & 1 \\ & & . & 5 & 1 & 2 \\ & & & . & 2 & 3 \\ & & & & . & 4 \\ & & & & & . \end{bmatrix} \tag{14}$$

Therefore (1) is a P(ermutation)-wavefront ordering that is equivalent to (2). (2) converges and under the permutation equivalence so does (1) [12].

6 Conclusion

In this paper we have discussed the design of an efficient, parallel implementation of the Jacobi eigen-solver algorithm.

It is often claimed that without some form of destructive updates, the computational efficiency of some algorithms can be increased. In this paper, we have demonstrated that the Jacobi method, though initially encumbered by the functional semantics of non-destructive array updates, is efficiently expressible in the functional paradigm. Without relaxing this fundamental restriction, an efficient implementation was still possible by resorting to parallelism at the algorithmic level, so that unnecessary copying of array elements could be totally eliminated.

Given their implicit nature, it is not always clear how to assess the complexity of functional algorithms. We have used the notion of *abstracted or computational complexity* to guide the design of our functional numerical algorithms

7 Appendix

THEOREM

For N odd, the congruence $k \equiv [m - p + (q - p - 1)(m - 1)]\ mod(2m - 1)$ is equivalent to the Sameh parallel rotation groups (see function MakePQs in the text) which defines the minimal number of groups that are consequently of maximal size, covering all points (p, q) of the upper triangle of A.

PROOF

In the range $k = 1, 2, \ldots, m - 1$, the following holds

$$m - k + 1 \leq q \leq 2m - 2k,$$

$$p = 2m - 2k + 1 - q,$$

and

$$2m - 2k + 1 \leq q \leq 2m - k - 1,$$

$$p = 4m - 2k - q,$$

whereas in the range $k = m, m + 1, \ldots, 2m - 1$, we have

$$2m - 2k + 1 \leq q \leq 2m - k - 1,$$
$$p = 4m - 2k - q,$$

and

$$4m - 2k \geq p \geq 3m - k - 1,$$
$$p = (6m - 2k - 1) - q.$$

Consider a symmetric matrix of dimension $N \ \ times\ N$. We will consider the case for N odd, $N = 2m - 1$. The elements of the matrix (p,q) are ordered so that $p \leq q$.

Case a.

$k = 1, 2, \ldots, m - 1,$

Case a1.

$$m - k + 1 \leq q \leq 2m - 2k,$$
$$p - q = 2m - 2k + 1. \tag{15}$$

Solving for k in Eqn. 15, gives

$$k = m - \frac{p + q - 1}{2}. \tag{16}$$

Eqn. 15 restricts the index pair (p, q) to $p + q$, odd. Setting k to the value in Eqn. 15, the following difference is formed:

$$(m - \frac{p + q - 1}{2}) - (m - p + (q - p - 1)(m - 1))$$

and with a little manipulation, it is easy to show that

$$(m - \frac{p + q - 1}{2}) - (m - p + (q - p - 1)(m - 1))$$
$$= \frac{p - q + 1}{2}(2m - 1).$$

From Eqn. 15, $p + q$ is odd and so $p - q + 1$ is even and divisible by 2. Thus

$$m - \frac{p + q - 1}{2}$$
$$\equiv m - p + (q - p - 1)(m - 1) \; mod(2m - 1). \tag{17}$$

Case a2.

$$2m - 2k + 1 \leq q \leq 2m - k - 1,$$
$$p - q = 4m - 2k, \tag{18}$$

Solving for k in Eqn. 18, gives

$$k = 2m - \frac{p + q}{2}. \tag{19}$$

and restricts the index pair (p, q) to $p + q$, even.

Using Eqn. 19, the difference between k and $(m - p + (q - p - 1)(m - 1))$ gives

$$(2m - \frac{p + q}{2}) - (m - p + (q - p - 1)(m - 1))$$
$$= \frac{p - q + 2}{2}(2m - 1). \tag{20}$$

From Eqn. 19, $p + q$ is even, then so is $p - q + 2$. Thus

$$m - \frac{p + q}{2}$$
$$\equiv m - p + (q - p - 1)(m - 1) \, mod(2m - 1). \tag{21}$$

Case b.

$k = m, m + 1, \ldots, 2m - 1,$

Case b1.

See Case a2 above.

Case b2.

$$4m - 2k \geq p \geq 3m - k - 1, \tag{22}$$
$$p = (6m - 2k - 1) - q.$$

So $p + q$ is even.

$$k = 3m - \frac{p + q + 1}{2}. \tag{23}$$

Since $p + q + 1$ is even, k is integral. Forming the difference

$$\begin{aligned} & k - (m - p + (q - p - 1)(m - 1)) \\ & = \frac{p - q + 3}{2}(2m - 1). \end{aligned} \tag{24}$$

Since $p + q$ is odd so is $p - q + 3$ and thus

$$k \equiv m - p + (q - p - 1)(m - 1)\, mod(2m - 1). \tag{25}$$

QED

References

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

[2] A.P.W. Böhm, R. E. Hiromoto, *The Dataflow Time and Space Complexity of FFTs*, Submitted to the Journal of Parallel and Distributed Computing, a special Datalow issue, guest edited by Gao et al.

[3] G.E. Forsythe and P. Henrici, *The Cyclic Jacobi Method for the Principal Values of a Complex Matrix*, Trans. Amer. Math. Soc., 94 (1960), pp. 1-23.

[4] David Cann, *Retire Fortran? A Debate Rekindled*, Proceedings of Supercomputing '91, IEEE Computer Society Press, 1991, pp. 264-272.

[5] J.R. Gurd, A.P.W. Böhm and Y.M. Teo, *Performance Issues in Dataflow Machines*, Future Generation Computer Systems, 3, 1987, pp. 285-297.

[6] J.J. Lambiotte, Jr., R. Voigt, *The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer*, ACM Transaction on Mathematical Software, 1, pp. 308-329, 1975.

[7] *Id World User's Manual*, Motorola, Inc., 1992.

[8] D. R. Morais, *ID World: An Environment for the Development of Dataflow Programs Written in ID*, MIT LCS TR-365, may 1986.

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

[10] W.H. Press et al., Numerical Recipes, the art of Scientific programming, Cambridge University Press.

[11] A. H. Sameh, *On Jacobi-like Algorithm for a Parallel Computer*, Math. Comput., 25 (1971), pp. 579-590.

[12] G. Shroff and R. Schreiver, *On the Convergence of the Cyclic Jocobi Method for Parallel Block Orderings*, SIAM J. Matrix Anal. Appl., Vol. 10, No. 3, pp. 326-346, July 1989.