Analysis of Non-Strict Functional Implementations of the Dongarra-Sorensen Eigensolver

S. Sur and W. Bohm

Technical Report CS-93-133

December 15, 1993

Analysis of Non-Strict Functional Implementations of the Dongarra-Sorensen Eigensolver

S. Sur and W. Böhm [*]

Department of Computer Science
Colorado State University
Ft. Collins, CO 80523

December 14, 1993

Abstract

We study the producer-consumer parallelism of Eigensolvers composed of a tridiagonalization function, a tridiagonal solver, and a matrix multiplication, written in the non-strict functional programming language Id. We verify the claim that non-strict functional languages allow the natural exploitation of this type of parallelism, in the framework of realistic numerical codes. We compare the standard top-down Dongarra-Sorensen solver with a new, bottom-up version. We show that this bottom-up implementation is much more space efficient than the top-down version. Also, we compare both versions of the Dongarra-Sorensen solver with the more traditional QL algorithm, and verify that the Dongarra-Sorensen solver is much more efficient, even when run in a serial mode. We show that in a non-strict functional execution model, the Dongarra-Sorensen algorithm can run completely in parallel with the Householder function. Moreover, this can be achieved without any change in the code components. We also indicate how the critical path of the complete Eigensolver can be improved.

Address for Correspondence:

A. P. W. Böhm
Department of Computer Science
Colorado State University
Ft. Collins, CO 80523
Tel: (303) 491-7595
Fax: (303) 491-6639
Email: bohm@CS.ColoState.Edu

[*]This work is supported in part by NSF Grant MIP-9113268, Motorola Grant YCM002, and a Motorola Monsoon donation from ARPA

1 Introduction

In our work we study the effectiveness of non-strict functional programming languages in expressing the parallelism of complex numerical algorithms in a machine independent style. A numerical application is often composed of a number of algorithms. In this paper, for example, we study an Eigensolver composed of a tridiagonalization function, a tridiagonal solver, and a matrix multiplication. We verify the claim that non-strict functional languages allow the natural exploitation of fine-grain parallelism of modular programs [7]. Elements of a non-strict data structure can be used before the whole structure is defined. Combined with the data-driven execution of functional modules, this provides for maximal exploitation of parallelism without the need for explicit specification of it.

In this paper we compare the standard top-down Dongarra-Sorensen solver with a new, bottom-up version. We show that this bottom-up implementation is much more space efficient than the top-down version. Also, we compare both versions of the Dongarra-Sorensen solver with the more traditional QL algorithm, and verify that the Dongarra-Sorensen solver is much more efficient, even when run in a serial mode.

Our algorithms are written in Id [8] and run on the Motorola Monsoon machine [6]. To obtain parallelism profiles, we run our programs on a Monsoon Interpreter. To obtain information about the space usage of our programs, we determine the largest problem size that can run on a one node (one processor module and one storage module) Monsoon machine.

Dongarra and Sorensen mention the possibility of exploiting producer-consumer parallelism between Householder and their algorithm [4] and mention that "an efficient implementation of this scheme is difficult". We will show that in a non-strict functional execution environment, the Dongarra-Sorensen algorithm can run completely in parallel with the Householder function. Moreover, this has been achieved without any change in the code components.

2 The Dongarra-Sorensen Eigensolver

Let A be a symmetric matrix, for which we want to find the eigenvectors and eigenvalues. The Householder transformation function takes A and produces a tridiagonal matrix represented by the diagonal d and upper diagonal e, and an orthogonal transformation matrix Q. The QL factorization function transforms d and e into a vector containing the eigenvalues and a matrix Q' of eigenvectors of the tridiagonal system. The eigenvalues of A are equal to the eigenvalues of the tridiagonal system, whereas the eigenvectors of A are obtained by multiplying Q and Q'. The Dongarra-Sorensen algorithm performs the same operation as QL, but in a divide and conquer fashion. For further details concerning eigensolvers we refer to [5].

In this section we introduce the existing theory regarding Dongarra-Sorensen algorithm for solving the eigenvalue problem of a tridiagonal matrix in some more detail, because we will introduce a bottom up version of the algorithm later.

The Dongarra-Sorensen algorithm is a *divide and conquer* approach [4, 2] for computing the eigenvalues and eigenvectors of a *symmetric tridiagonal* matrix. Let T be a symmetric tridiagonal matrix:

$$\mathbf{T} = \begin{bmatrix} a_1 & b_1 & 0 & \dots & & . & 0 \\ b_1 & a_2 & b_2 & \dots & & & \\ & & . & & & & . \\ & & . & & & & . \\ & & . & & & & . \\ . & & & & & & \\ . & & & & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & . & & & 0 & b_{n-1} & a_n \end{bmatrix} \tag{1}$$

The Dongarra-Sorensen algorithm computes the Schur decomposition

$$Q^T T Q = \Lambda = diag(\lambda_1, \cdots, \lambda_n), \; Q^T Q = I$$

by gluing together the Schur decompositions of two half sized tridiagonal problems derived from the original matrix T. To obtain these half-sized problems we use *partitioning by rank-one tearing* discussed below. Each of these reductions can in turn be specified by a pair of quarter sized Schur decompositions and so on.

2.1 Partitioning by rank-one tearing:

One can easily check that any symmetric tridiagonal matrix T can be reduced to the following form:

$$\mathbf{T} = \begin{pmatrix} T_1 & \beta e_k e_1^T \\ \beta e_1 e_k^T & T_2 \end{pmatrix} = \begin{pmatrix} \hat{T}_1 & 0 \\ 0 & \hat{T}_2 \end{pmatrix} + \theta^{-1}\beta \begin{pmatrix} e_k \\ \theta e_1 \end{pmatrix} \begin{pmatrix} e_k^T & \theta e_1^T \end{pmatrix} \tag{2}$$

where $1 \leq k \leq n$ and e_j represents the j-th unit vector of appropriate dimension and $\beta = b_k$. $\hat{T}_1$ is identical to top $k \times k$ tridiagonal sub-matrix of T except that the last diagonal element $\tilde{a}_k$ is modified so that $\tilde{a}_k = a_k - \rho$, where $\rho = \beta/\theta$. Similarly, $\hat{T}_2$ is the bottom $(n-k)\times(n-k)$ tridiagonal submatrix of T, with only the first diagonal element modified. This modified element $\tilde{a}_{k+1}$ is given by $\tilde{a}_{k+1} = a_{k+1} - \rho\theta^2$. The factor θ is incorporated to avoid certain numerical difficulties associated with cancellation of diagonal terms [4].

2.2 Divide and conquer step:

Now we have two smaller tridiagonal eigenvalue problems to solve. We can first find the Schur decompositions of $\hat{T}_1$ and $\hat{T}_2$ so that:

$$\hat{T}_1 = Q_1 D_1 Q_1^T, \;\; \hat{T}_2 = Q_2 D_2 Q_2^T$$

which gives,

$$\mathbf{T} = \begin{pmatrix} Q_1 D_1 Q_1^T & 0 \\ 0 & Q_2 D_2 Q_2^T \end{pmatrix} + \theta^{-1}\beta \begin{pmatrix} e_k \\ \theta e_1 \end{pmatrix} \begin{pmatrix} e_k^T & \theta e_1^T \end{pmatrix} \tag{3}$$

Therefore,

$$T = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \left(\begin{pmatrix} D_1 & 0 \\ 0 & D_2 \end{pmatrix} + \theta^{-1}\beta \begin{pmatrix} q_1 \\ \theta q_2 \end{pmatrix} \begin{pmatrix} q_1^T & \theta q_2^T \end{pmatrix} \right) \begin{pmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{pmatrix} \tag{4}$$

where $q_1 = Q_1^T e_k$ (the last row of matrix Q_1) and $q_2 = Q_2^T e_1$ (the 1st row of matrix Q_2). The problem right now is reduced to computing the eigensystem of the interior matrix in the previous equation, which is discussed in the following section.

2.3 The updating problem:

The problem that is left to solve is that of computing the eigensystem of a matrix of the form

$$\hat{Q}\hat{D}\hat{Q}^T = D + \rho z z^T \tag{5}$$

where D is a real $n \times n$ diagonal matrix, ρ is non-zero scalar and z is real vector of order n. In our case

$$D = \begin{pmatrix} D_1 & 0 \\ 0 & D_2 \end{pmatrix}, \quad z = \begin{pmatrix} q_1 \\ \theta q_2 \end{pmatrix}, \quad and \;\; \rho = \frac{\beta}{\theta}$$

In this study we implement this eigensolver for the case where all the eigenvalues are distinct and so we can write $D = diag(\delta_1, \delta_2, \cdots, \delta_n)$, where $\delta_i \neq \delta_j$ for $i \neq j$. Moreover, we can sort the δ's and sort z accordingly such that $\delta_i < \delta_j$, for $i < j$. We also assume that no component ζ_i of vector z is zero. The eigensolver can be modified to solve problems with equal eigenvalues and zero components of z by incorporating certain deflation techniques into the algorithm [4]. We will, however, not deal with these deflation techniques. Under above assumptions, the eigenpair λ (the eigenvalue) and q (the corresponding eigenvector) satisfying

$$(D + \rho z z^T)q = \lambda q$$

can be obtained from satisfying the following equations [4]:

$$1 + \rho z^T (D - \lambda I)^{-1} z = 0 \tag{6}$$

and q is obtained from

$$q = (D - \lambda I)^{-1} z \tag{7}$$

If equation (6) is written in terms of the components ζ_i of z, then λ must be a root of the equation

$$f(\lambda) = 1 + \rho \sum_{j=1}^{n} \frac{\zeta_j^2}{\delta_j - \lambda} = 0 \tag{8}$$

Equation (8) is referred to as the *secular equation.* A Newton's method to solve this will not converge [2] and the general bisection method would be too slow. However, this equation has the delightful property of having a distinct root between every pair of consecutive diagonal elements (δ_i, δ_{i+1}). This property is used by Dongarra and Sorensen [4] to come up with a fast root-finder described below.

2.4 The root-finder of the secular equation:

Without loss of generality, one can assume that the coefficient ρ of the secular equation is positive. If it is not, a change of variable can be used where ρ can be replaced by $-\rho$. To achieve this without changing the secular equation δ_i needs to be replaced by $-\delta_{n-i+1}$ and ζ_i needs to be replaced by ζ_{n-i+1} for all i. Given that we wish to find the i-th root $\hat{\delta_i}$ of the function f in equation (8), the function can be rewritten as

$$f(\lambda) = 1 + \psi(\lambda) + \phi(\lambda) \tag{9}$$

where

$$\psi(\lambda) = \rho \sum_{j=1}^{i} \frac{\zeta_j^2}{\delta_j - \lambda}$$

and

$$\phi(\lambda) = \rho \sum_{j=i+1}^{n} \frac{\zeta_j^2}{\delta_j - \lambda}$$

This root lies in the open interval (δ_i, δ_{i+1}) and for λ in this interval all of the terms of ψ are negative and all of the terms of ϕ are positive. This situation is very suitable for an iterative method for solving the equation

$$-\psi(\lambda) = 1 + \phi(\lambda)$$

One can start with an initial guess λ_0 close to δ_i in the appropriate interval so that $\lambda_0 < \lambda$ [2], and then construct simple rational interpolants of the form

$$\frac{p}{q-\lambda}, \; r + \frac{s}{\delta - \lambda}$$

where δ is fixed at δ_{i+1} (the i+1th diagonal element of D) and the parameters p, q, r, s are defined by the interpolation conditions

$$\frac{p}{q-\lambda_0} = \psi(\lambda_0), \;\; r + \frac{s}{\delta-\lambda_0} = \phi(\lambda_0), \;\; \frac{p}{(q-\lambda_0)^2} = \psi^{'}(\lambda_0), \;\; \frac{s}{(\delta-\lambda_0)^2} = \phi^{'}(\lambda_0)$$

The new approximate λ_1 to the root $\hat{\delta_i}$ is then found by solving

$$\frac{-p}{q-\lambda} = 1 + r + \frac{s}{\delta - \lambda} \tag{10}$$

A sequence of iterates is thus derived following the same principle and the process is stopped when the value of the secular function at the current iterate goes below a certain threshold. Bunch et. al. [2] showed that this iteration converges quadratically from one side of the root and does not need any safeguarding.

3 A bottom-up approach

The theory described in the previous section is particularly suitable for a top-down implementation, where each problem is recursively reduced to two smaller size problems, until the trivial case (problem size 1) is reached. As we will see in the analysis and results section this implementation is very inefficient in terms of space. In this section we develop the theory for an alternate version of the algorithm, which starts from the bottom instead of the top, glues the solutions of smaller problems together at every iteration, and finally arrives at the solution. It turns out that this approach is extremely efficient in terms of space and does not lose any bit of time efficiecy. Our bottom-up approach provides more insight in the workings of the Dongarra-Sorensen algorithm too. Also, it is useful for other language implementations, as *while* loops are usually more efficient than the recursive calls. Moreover this form of implementation is more suitable for languages not supporting recursive calls, e.g. FORTRAN.

In order to perform the Dongarra-Sorensen algorithm bottom-up, we need to compute the effects of all the rank-one tearings, so that we solve the same size-1 problems and combine them in the same way as the top-down algorithm does. The important observation on which our bottom-up approach is based, is that the mathematical theory behind Dongarra-Sorensen algorithm is independent of the position of the rank-one tearings. Therefore it is irrelevant where the matrix is partitioned as long the final result size-1 matrices are derived from $n-1$ tearings, one at each position. Thus, iteratively tearing at the the top, done across all the elements, gives the same bottom case as the recursive half and half tearing. So we start with a tridiagonal matrix given in (1) and partition it by a rank one tearing about the first off-diagonal element b_1. We get,

$$\mathbf{T} = \begin{bmatrix} a_1 - b_1/\theta & 0 & 0 & \dots & & . & 0 \\ 0 & a_2 - b_1\theta & b_2 & \dots & & & \\ & & . & & & & . \\ & & . & & & & . \\ & & . & & & & . \\ . & & & & & & \\ . & & & & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & . & & & 0 & b_{n-1} & a_n \end{bmatrix} + \frac{b_1}{\theta} \begin{bmatrix} 1 & \theta & 0 & \dots & & . & 0 \\ \theta & \theta^2 & 0 & \dots & & & \\ & & . & & & & . \\ & & . & & & & . \\ & & . & & & & . \\ . & & & & & & \\ . & & & & 0 & 0 & 0 \\ 0 & . & & & 0 & 0 & 0 \end{bmatrix}$$

The second matrix of the above equation can be rewritten as $\frac{b_1}{\theta}\begin{pmatrix} e_1^1 \\ \theta e_1^{n-1} \end{pmatrix}\begin{pmatrix} e_1^1 \\ \theta e_1^{n-1} \end{pmatrix}^T$. Here e_j^k represents the j-th unit vector of length k. Tearing the matrix again with respect to element b_2 gives $\mathbf{T}$ =

$$\begin{bmatrix} a_1 - \frac{b_1}{\theta} & 0 & 0 & \dots & . & 0 \\ 0 & a_2 - b_1\theta - \frac{b_2}{\theta} & 0 & \dots & & \\ 0 & 0 & a_3 - b_2\theta & & & . \\ & & . & & & . \\ & & . & & & . \\ . & & & & & \\ 0 & . & & & b_{n-1} & a_n \end{bmatrix} + \frac{b_1}{\theta}\begin{pmatrix} e_1^1 \\ \theta e_1^{n-1} \end{pmatrix}\begin{pmatrix} e_1^1 \\ \theta e_1^{n-1} \end{pmatrix}^T + \frac{b_2}{\theta}\begin{pmatrix} e_2^2 \\ \theta e_1^{n-2} \end{pmatrix}\begin{pmatrix} e_2^2 \\ \theta e_1^{n-2} \end{pmatrix}^T$$

Repeating this tearing process $n-1$ times, i.e. for each off-diagonal element, the tridiagonal matrix T can be torn down to a diagonal matrix and summation of matrices of the form ρzz^T, which is given by:

$$\mathbf{T} = \begin{bmatrix} a_1 - \frac{b_1}{\theta} & 0 & 0 & \dots & . & 0 \\ 0 & a_2 - b_1\theta - \frac{b_2}{\theta} & 0 & \dots & & \\ & & . & & & . \\ & & . & & & . \\ . & & & & & \\ 0 & . & & & a_{n-1} - b_{n-2}\theta - \frac{b_{n-1}}{\theta} & 0 \\ 0 & . & & & 0 & a_n - b_{n-1}\theta \end{bmatrix} + \sum_{i=1}^{n-1} \frac{b_i}{\theta} \begin{pmatrix} e_i^i \\ \theta e_1^{n-i} \end{pmatrix} \begin{pmatrix} e_i^i \\ \theta e_1^{n-i} \end{pmatrix}^T \tag{11}$$

The diagonal matrix in equation (11) is the starting point of our bottom-up approach, and as mentioned before is the same bottom case of the top-down approach. Every element of the diagonal matrix can be considered as the eigensolution (Schur decomposition) QDQ^T of a single element matrix, where D is the element itself and Q is the 1×1 matrix 1. Now, two of these solutions of size one problems can be updated to get a solution of a size two problem. The first combination step can be viewed as:

$$\mathbf{T} = \begin{bmatrix} \delta_1 + \frac{b_1}{\theta}\begin{pmatrix} 1 & \theta \\ \theta & \theta^2 \end{pmatrix} & 0 & \dots & 0 \\ 0 & \delta_3 + \frac{b_3}{\theta}\begin{pmatrix} 1 & \theta \\ \theta & \theta^2 \end{pmatrix} & \dots & 0 \\ & & . & . \\ . & . & . & . \end{bmatrix} + \sum_{i=even} \frac{b_i}{\theta} \begin{pmatrix} e_i^i \\ \theta e_1^{n-i} \end{pmatrix} \begin{pmatrix} e_i^i \\ \theta e_1^{n-i} \end{pmatrix}^T \tag{12}$$

Where δ_i is the 2×2 matrix given by:

$$\begin{pmatrix} a_i - b_{i-1}\theta - \frac{b_i}{\theta} & 0 \\ 0 & a_{i+1} - b_i\theta - \frac{b_{i+1}}{\theta} \end{pmatrix}$$

assuming $b_0 = 0$ and $b_n = 0$. Here the matrix $\delta_i + \frac{b_i}{\theta}\begin{pmatrix} 1 & \theta \\ \theta & \theta^2 \end{pmatrix}$ is in $D + \rho zz^T$ form and can be Schur decomposed to $Q_i D_i Q_i^T$ form by solving the updating problem given in equation 5. Hence after the first step of updating the tridiagonal matrix is:

$$\mathbf{T} = \begin{bmatrix} Q_1 D_1 Q_1^T & 0 & \dots & 0 \\ 0 & Q_3 D_3 Q_3^T & \dots & 0 \\ & & . & . \\ . & . & . & . \end{bmatrix} + \sum_{i=even} \frac{b_i}{\theta} \begin{pmatrix} e_i^i \\ \theta e_1^{n-i} \end{pmatrix} \begin{pmatrix} e_i^i \\ \theta e_1^{n-i} \end{pmatrix}^T \tag{13}$$

We now continue the same process, and in the next step include the term associated with b_2, b_6 etc. from the summation to inside of the updated matrix in the following form:

$$\frac{b_i}{\theta} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & \theta & 0 \\ 0 & \theta & \theta^2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The same updating is then done as given by equation 4 and Schur decomposition of 4x4 matrices are obtained. The process is continued until all the terms of the summation outside the matrix are exhausted and the final result obviously is the eigensolution of the tridiagonal problem. A top-down recursive approach also does updating of the same elements in the recombination step (after the problem is broken down to the bottom-most level).

4 Implementation Issues

4.1 Top-down implementation

Here we describe the functional implementation of the original top-down version of the Dongarra-Sorensen algorithm. A function $ds(d, e, n, \theta)$ takes d, the diagonal vector, e, the off-diagonal vector, n, the dimension and θ, the stability factor θ discussed in the section 2, as arguments. It returns ev, the vector of eigenvalues and Q, the corresponding matrix of eigenvectors as the output. If the size is 1 the only element in the diagonal vector is returned as the eigenvalue and the identity matrix of dimension 1×1 is returned as Q. Otherwise the problem is broken into two parts, half the size of the original. Uneven halves originate if the problem size is not even, which need not be treated specially because the position of the tearing does not affect the final result. Two half size vectors are generated from each of the vectors d and e. The last element of the first half of d and the first element of the second half of d are modified, as they represent the last diagonal element of $\hat{T}_1$ and first diagonal element of $\hat{T}_2$, respectively (described in section 2.1). Now, the two half eigenproblems are solved by recursively calling ds for each half. The rest of the work is to glue these two solutions together, i.e. the recombination step in a normal recursive function. Let us call the vector of eigenvalues and matrix of eigenvectors of the first half v_1 and Q_1 respectively. Call the corresponding ones for the second half v_2 and Q_2. Combining v_1 and v_2 gives the term D of equation (5). To obtain z of the same equation, we combine the last row of Q_1 and θ times the first row Q_2. The next step is to solve the updating problem described in section 2, which requires the elements in D to be sorted. Consequently, to keep the same correspondence between D and z in $D + \rho zz^T$ eigenproblem, z also needs to be rearranged, according to D's sorted indices. To solve the $D + \rho zz^T$ eigenproblem the secular equation can now be formed and its roots will give us the eigenvalues. A function $rf(D, z, \rho)$ computes n roots by choosing a starting point λ_0 infinitesimally right of the left endpoint within each interval of components of D and iterating within the interval as follows:

$$\lambda_{i+1} = \lambda_i + 2b/(a + \sqrt{a^2 - 4b}),$$

where

$$a = (\Delta(1 + \phi_i) + \psi_i^2/\psi_i^{'})/c + \psi + i/\psi_i^{'},$$

$$b = (\Delta w \psi_i)/(\psi_i^{'} c),$$

$$c = 1 + \phi_i - \Delta \phi_i^{'},$$

$$w = 1 + \phi_i + \psi_i,$$

and

$$\Delta = \delta_{i+1} - \lambda_i$$

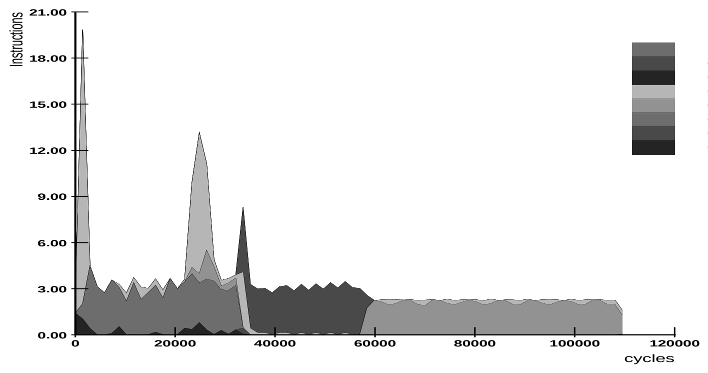

Figure 1: Idealized Parallelism Profile for Top-Down Dongarra-Sorensen

where ϕ and ψ have the same meaning as in section 2. The roots of the secular equation thus obtained are the eigenvalues of the $D + \rho zz^T$ problem, which are also the eigenvalues of the symmetric tridiagonal problem we started with. The corresponding eigenvectors are obtained using equation (7) of section 2. These eigenvectors are normalized and the final matrix form of the eigenvectors of the tridiagonal problem is obtained by multiplying thss matrix of eigenvectors (of $D + \rho zz^T$ problem) with the zero padded Q matrix of equation (4).

4.2 Iterative (bottom-up) Dongarra-Sorensen algorithm

This implementation first creates the bottom case by filling in a vector *curv* (the current value of eigenvalues) with $a_i - b_i\theta - b_{i-1}/\theta$ (refer to the preceding section), assuming $b_0 = 0 = b_n$. The corresponding matrix of eigenvectors is the identity matrix at this point of time. In a *while* loop we start solving the updating problem (see section 2), and start with gluing the solutions of size 1 to solutions of size 2. Every two consecutive elements of the vector *curv* now contains eigenvalues of the size-2 problem (obtained from usual formulation and solution of the secular equation). The matrix of eigenvectors *curQ* contains 2x2 submatrices along the diagonal (and 0's elsewhere), that represents the two eigenvectors of problem size-2. In the next step two of these size-2 solutions are glued together to form solutions of size 4, and the process is continued, increasing step size by a factor of 2 in every successive iteration, until the whole problem is solved. This method requires that the problem size be a power of two. But this limitation can be overcome if the problem size n is broken into sum of powers of two (the binary representation of n), and each part is solved using the method described above, and then gluing the solutions together in the usual way (by solving the updating problem).

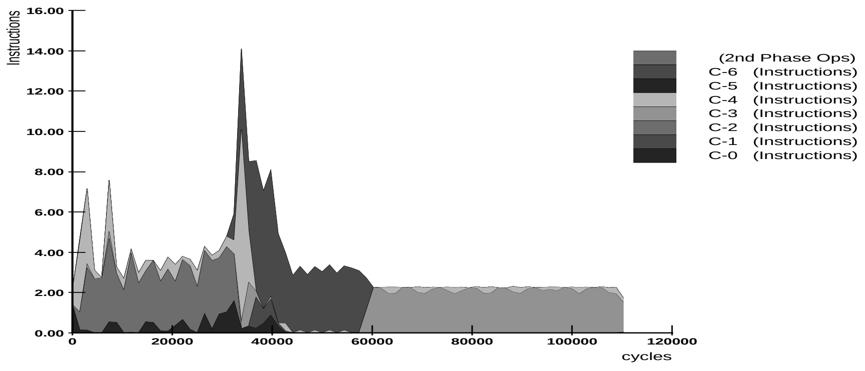

Figure 2: Idealized Parallelism Profile for Bottom-Up Dongarra-Sorensen

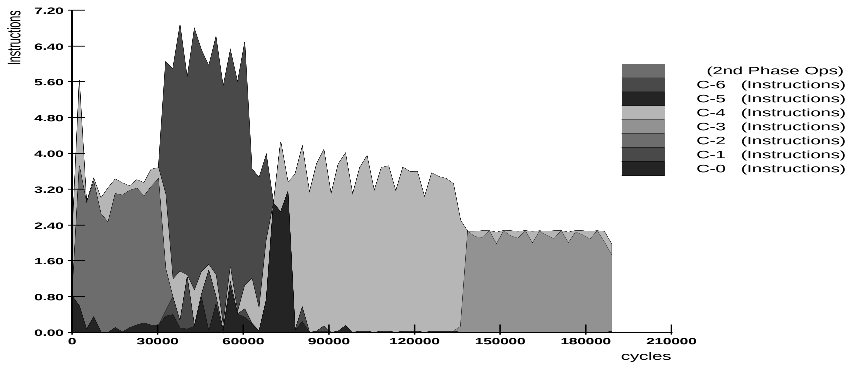

Figure 3: Idealized Parallelism Profile for QL

Function cost	HH + Top-down DS	HH + Bottom-up DS	HH +QL
HH eigenvalue computation	93,320	89,622	91,754
HH eigenvector computation	77,905	99,699	167,002
Matrix multiply	114,553	112,781	109,756
Tridiagonal solver (HH or QL)	78,560	53,775	256,391
Total instructions	374,405	379,010	679,928
Critical path length	109,640	110,450	189,340
Max size on monsoon 1PE1IS	55×55	166×166	101×101

Table 1: Quantitative Characteristics of the Eigensolvers

5 Time and Space Analysis

Figures 1, 2 and 3 display the idealized parallelism profiles of the three implementations of the eigensolver for an input matrix of size 16×16, where the (i,j)-th element equals (i+j). The algorithms behave similarly for other symmetric input matrices we have tried. The horizontal *cycles* axis represents the critical path measured in the number of execution cycles under the idealized assumption that an instruction will get executed as soon as its inputs are available. The vertical *instructions* axis represents the number of instructions that can execute in parallel at a certain cycle. The simulator allows us to mark the instructions of certain function with a certain "color". Colors 0 and 1 display the instructions executed by the run-time system. In this way we can study the overlapping of function execution. In all figures, color 2 represents the part of the Householder function that produces the diagonals d and e, and color 6 represents the part of the Householder function that produces the orthogonal transformation matrix Q. Color 4 represents the particular tridiagonal solver, color 3 represents the matrix multiplication. The peak at the beginning of the execution in figure 1 represents the parallel unrolling of the divide and conquer tree of the recursive *ds* function. The second peak in figure 1 and the peak in figure 2 represent the execution of the root-finder.

Table 1 gives for each implementation information regarding the number of instructions per function, the total number of instructions executed, the total critical path length, and the maximum size problem that can be run on a one node Monsoon machine [6], which has a 4 Megaword data memory. From the figures and table we draw the following conclusions.

- Both top-down and bottom-up Dongarra-Sorensen implementations (color 4) exploit producer-consumer parallelism, and run in parallel with the part of Householder that creates the diagonals d and e. The Dongarra-Sorensen tridiagonal solvers end virtually at the same time as the part of Householder that creates the diagonals d and e, which indicates that the non-strict implementation of arrays in Id is highly effective for this application.
- The part of the Householder function that produces Q (color 6) contributes to the critical path. Therefore, in order to shorten the critical path of the program, attention should be paid to this part of the algorithm. Improving the parallelism of Householder can be easily achieved by by increasing the K-Bounds of the loops [3].

- The matrix multiplication (color 3) can only start after the matrix Q has been produced by the Householder function, and contributes largely to the critical path length. Again, a more parallel version of matrix multiply would shorten the overall critical path length, and can also be realized by increasing K-bounds of loops.

- Table 1 indicates that the bottom-up version of Dongarra-Sorensen is much more space efficient than the top-down version. The largest problem size solvable on a one node Monsoon machine is 9 times larger for the bottom-up than that for the top-down algorithm. The bottom-up algorithm runs out of heap memory (data structure space), whereas the top-down algorithm runs out of frame memory (run-time stack space). This is because the recursive algorithm is executed in an eager, breadth first order, which uses excessive amounts of frame store. This relates directly to the *throttling problem* [9] of controlling the amount of program parallelism such that the machine resources are utilized, but not swamped.

- For the 16×16 problem, QL (color 4) has a 75% longer critical path length than the two Dongarra-Sorensen algorithms, and executes 80% more instructions. This verifies the claim in [4] that even in sequential mode, the Dongarra-Sorensen algorithm outperforms QL. Also note that QL cannot do much work until all of d and e have been produced, which is an inherent characteristic of the algorithm. This makes producer consumer parallelism hard to achieve for QL.

6 Conclusions

We have studied symmetric Eigensolvers, consisting of the Householder tridiagonalization, a tridiagonal solver, and a matrix multiplication in a non-strict functional environment. For the tridiagonal solver we have studied implementations of the Dongarra-Sorensen algorithm and compared them with the more traditional QL algorithm. We have introduced a new, bottom-up, implementation of the Dongarra-Sorensen algorithm, and have shown that it is considerably more space efficient than the top-down version without incurring time inefficiency.

The Dongarra-Sorensen functions run completely in parallel with the Householder function, which verifies the claim that non-strict functional languages allow the modular design of applications, exploiting producer-consumer parallelism to the fullest.

We have observed that the critical path of the programs can be improved by making Householder and Matrix Multiply more parallel, and indicated how this can be achieved.

The QL algorithm needs all values of its input vectors in order to start executing and therefore does not benefit as much from the non-strictness of the data structures. QL has a longer critical pathlength and executes a considerably larger number of instructions, which verifies the claim that for sufficiently large problems, Dongarra-Sorensen is a better algorithm, even in a sequential execution environment.

References

[1] Böhm, A. P. and R. E. Hiromoto, "The Dataflow Time and Space Complexity of FFTs", **J. Par and Dist Comp**, Vol. 18, pp. 301-313, 1993.

[2] Bunch, J. R., C.P. Nielsen and D. C. Sorensen, "Rank-one Modification of the Symmetric Eigenproblem", **Numerische Mathematik**, 31, pp. 31-48, 1978. Reading, MA, 1972.

[3] Culler, D. E., "Managing parallelism and resources in scientific dataflow programs", PhD Thesis, MIT, June 1989.

[4] Dongarra, J. J. and D. C. Sorensen, "A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem", **SIAM J. Sci and Stat Comp**, Vol. 8, pp. S139-154, March 1987.

[5] Golub, G. H. and C. F. Van Loan, "Matrix Computations", The Johns Hopkins University Press, 2nd edition, 1989.

[6] Hicks, James, D. Chiou, B. S. Ang and Arvind, "Performance studies of Id on the Monsoon dataflow system," **Journal of Parallel and Distributed Computing** no. 18, pp 273-300, 1993.

[7] Hughes, J. "Why Functional Programming Matters", **The Computer Journal**, April 1989.

[8] Nikhil, R. S. , "Id Reference Manual, version 90.1", Computation structures group memo 284-2, MIT Laboratory for Computer Science, Cambridge, MA, Sept 1990.

[9] Ruggiero, C. A. and J. Sargeant, "Control of Parallelism in the Manchester Dataflow Computer", Lecture Notes in Computer Science no. 274, pp 1-15, 1987.