CONF-910771--1

ARGONNE NATIONAL LABORATORY

9700 South Cass Avenue

Argonne, Illinois 60439-4801

CONF-910771--1

DE91 006044

The Lanczos Algorithm for the Generalized Symmetric Eigenproblem on Shared-Memory Architectures*

Mark T. Jones and Merrell L. Patrick[†]

Preprint MCS-P182-0990

September 1990

ABSTRACT

The generalized eigenvalue problem, $Kx = \lambda Mx$, is of significant practical importance, for example, in structural engineering where it arises as the vibration and buckling problems. The paper describes the implementation of a solver based on the Lanczos algorithm, LANZ, on two shared-memory architectures, the CRAY Y-MP and Encore Multimax. Issues arising from implementing linear algebra operations on a multivector processor are examined. Portability between a multivector processor and a simple multiprocessor is discussed. A model is developed and used to predict the performance of LANZ on shared-memory architectures. Performance results from some practical problems are given and analyzed.

*This research was supported by the National Aeronautics and Space Administration under NASA contract nos. NAS1-18107 and NAS1-18605 while the authors were in residence at ICASE. Additional support was provided by NASA grant no. NAG-1-466. The first author also received support from the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The authors also received time on a CRAY Y-MP at the North Carolina Supercomputing Center.

[†] Permanent address: Computer Science Department, Duke University, Durham, NC 27706.

1. Introduction. The generalized symmetric eigenvalue problem, $Kx = \lambda Mx$, is of significant practical importance, for example, in structural engineering where it arises as the vibration and buckling problems. In the problems of interest, a few of the eigenpairs closest to some point, σ, in the eigenspectrum are sought. The matrices, K and M, are usually sparse or have a narrow bandwidth. New software, LANZ, based on the Lanczos algorithm has been developed for solving these problems and has been reported on in [5] [3] [7].

Because eigenvalue problems arising in structural engineering are often very large, it is natural to attempt to use parallel computers to solve them. In Section 2 the parallel LANZ algorithm and its implementation on shared-memory architectures with a small to moderate number of processors is described. A model for predicting the performance of LANZ on shared-memory architectures is given in Section 3. In Section 4 results from the implementations are given and analyzed.

2. Algorithm and Implementation. To speed convergence to desired eigenvalues, a shift-and-invert Lanczos algorithm similar to that described in [12] is used. On sequential and vector machines, this algorithm has been observed to be superior to the subspace iteration method that is popular in engineering [13] [5]. To maintain the desired semi-orthogonality among the Lanczos vectors, a version of partial reorthogonalization [19] is used. Extended local orthogonality among the Lanczos vectors is also enforced [11] [17]. If eigenvectors are found before executing the Lanczos algorithm, an improved version of external selective orthogonalization [2] suggested in [3] is used to avoid recomputing these eigenvectors. Although the discussions in this paper assume that M is positive semi-definite, the computations remain essentially the same when M is indefinite.

The Force, a Fortran-based language for parallel programming [9], was used to implement LANZ for two reasons: (1) it is available on several shared-memory architectures, thus allowing at least a superficial level of portability, and (2) it has been shown to be a language suitable for implementing parallel numerical linear algebra algorithms [8].

The parallel LANZ algorithm is presented in Figure 1. Its various computational components and their parallel implementations are discussed in the following subsections. Explicit global synchronization points in the algorithm are denoted by the term "SYNCHRONIZE." Other synchronization points are required by particular operations, for example inner products, and are not explicitly denoted in the algorithm. To avoid extra synchronization, each processor is responsible for computing a fixed subset of each vector computation. For example, if at step 21 processor i computes the first m elements of q_{j+1}, then at step 22, processor i would compute the contribution of the first m elements to the inner product, thus avoiding a synchronization between steps 21 and 22. In these discussions p represents the number of processors, n represents the

```
0) $q_0 = p_0 = 0$
1) Choose an initial vector, $guess$
2) $p_1 = M guess$
3) Orthogonalize
4) SYNCHRONIZE
5) $p_1 = M guess$
6) SYNCHRONIZE
7) $q_1 = (K - \sigma M)^{-1} p_1$
8) (factorization occurs here)
9) SYNCHRONIZE
10) $p_1 = M q_1$
11) $\beta_1 = (p_1^T q_1)^{\frac{1}{2}}$
12) Orthogonalize
13) $q_1 = q_1 / \beta_1$
14) $p_1 = p_1 / \beta_1$
15) For $j = 1, \ldots$
16)    $(K - \sigma M) q_{j+1} = p_j$
17)    (only matrix solution here)
18)    SYNCHRONIZE
19)    rnorm=$\| q_{j+1} \|$
20)    (if external orthogonalization)
21)    $\gamma = p_{j-1}^T q_{j+1}$
22)    $q_{j+1} = q_{j+1} - \gamma q_{j-1}$
23)    $\hat{\alpha} = p_j^T q_{j+1}$
24)    $q_{j+1} = q_{j+1} - \hat{\alpha} q_j$
25)    $\gamma = p_{j-1}^T q_{j+1}$
26)    $q_{j+1} = q_{j+1} - \gamma q_{j-1}$
27)    $\alpha_j = p_j^T q_{j+1}$
28)    $q_{j+1} = q_{j+1} - \alpha_j q_j$
29)    SYNCHRONIZE
30)    $p_{j+1} = M q_{j+1}$
31)    $\alpha_j = \alpha_j + \hat{\alpha}$
32)    $\beta_{j+1} = (p_{j+1}^T q_{j+1})^{\frac{1}{2}}$
33)    Calculate eigenvalues of $T_j$
34)    Count the converged eigenvalues
35)    Orthogonalize
36)    $q_{j+1} = q_{j+1} / \beta_{j+1}$
37)    (requires use of critical sections)
38)    $p_{j+1} = r_{j+1} / \beta_{j+1}$
39) End of Loop
40) compute ritz vectors
```

FIG. 1. *Parallel shift-and-invert Lanczos algorithm*

order of the matrices, b represents the block size in a block algorithm, and j represents the current Lanczos step.

2.1. Factorization. Factorization takes place only once during the algorithm, at step 7. Because the matrices, K and M, are sparse (or have been reordered to have a narrow bandwidth), the parallel implementation of direct factorization and solution methods must be carefully considered. In this paper, only the case in which the matrices have been reordered to a narrow bandwidth, β, will be considered. However, the limitations on parallelism in factorization and forward/backward matrix solution that are imposed by a narrow bandwidth are similar to those imposed by sparse matrices.

Two situations may exist when factoring $(K - \sigma M)$: (1) $(K - \sigma M)$ is known to be positive definite, and therefore it is desirable to use either Cholesky factorization or LDL^T decomposition, or (2) $(K - \sigma M)$ may be indefinite, and therefore a factorization algorithm with pivoting is necessary. In the first case, a block factorization and solution subroutine described in [18] has been parallelized for use in LANZ. In the second case, a block algorithm for banded matrices based on Bunch-Kaufman factorization is used [4] [6].

LANZ was initially written for vector architectures, and therefore careful attention has been paid to achieving good vectorization. With small-to-moderate vector lengths, it is desirable to perform *saxpy* operations[1] as opposed to inner products,

[1] The *saxpy* operation is defined as $w = ax + y$, where w, y, and x are vectors and a is a scalar.

as well as to compute more than one *saxpy* operation at a time.[2] On multivector processors, however, good vectorization is often at odds with parallelization. In the factorization algorithms, this conflict between vectorization and parallelization occurs in the computation of the pivot column(s): the pivot columns(s), vectors of length β, must be split into vectors of length β/p for each processor to compute. On the CRAY Y-MP the benefit of parallel computation of the pivot column is outweighed by the resulting inefficient short vector operations and the cost of the added synchronization; therefore, this computation is not parallelized. However, on a simple multiprocessor such as the Multimax, this conflict does not occur, and the computation of the pivot column is parallelized. The dominant part of the calculation is the updating of the uneliminated nonzeroes by using the pivot columns: the updating is implemented by distributing $\frac{\beta-b}{p}$ extended *saxpy*'s to each of the processors to compute. The extended *saxpy*'s parallelize well because there is sufficient work for each processor, and the vector lengths are unaffected by parallelization.

2.2. Matrix Solution. Forward and backward matrix solution is required at steps 7 and 16. The conflict between vectorization and parallelism is much worse in these operations. This discussion will be limited to the forward and backward solution algorithms that take place after a Bunch-Kaufman factorization in which the block sizes vary and are selected according to numerical criteria rather than the number of processors.[3] The following discussion will assume that the lower triangular factor, L, resulting from the Bunch-Kaufman algorithm has been stored by row.[4] Because of the order in which pivots are performed, a *saxpy*-based algorithm for the forward solution must be used, and an inner product algorithm for the backward solution must be used.

The time-consuming portion of the block forward solution algorithm is the b β-length *saxpy* operations that can be combined into a single extended *saxpy* operation. The only practical way to parallelize this operation is to split the vector into p shorter vectors. This approach, of course, significantly reduces the efficiency of the vector operations.

The time-consuming portion of the block backward solution algorithm is the computation of b β-length inner products. Two types of parallelism are available here: (1) two or more processors can cooperate to compute a single inner product, and (2) individual inner products can be computed independently. Even though both methods are used, the algorithm is still inefficient because inner products are not as fast

[2] Performing more than one *saxpy* at a time, called an *extended saxpy* in this paper, is defined as $w = y + \sum_{i=1}^{j} a_i x_i$ and is often implemented via loop unrolling [1]. This type of operation reduces the ratio of memory references to computations.

[3] The situation is slightly better for the positive definite case in which the block sizes can be selected based on the number of processors rather than according to numerical criteria.

[4] If it were stored by column, the same limitations would apply, but the discussion for the forward solution would be applicable to backward solution and vice versa.

as *saxpy*'s, the parallel computation of a short inner product is adversely affected by synchronization delays, and the block size may not be evenly divisible by p, and therefore a load imbalance may result.

The considerations regarding efficiency of vector operations are not a concern when implementing this algorithm on the Encore, and therefore better parallel speedup from the forward and backward solution algorithms can be expected than on the CRAY Y-MP. The ratio of computation to synchronization, however, is still much worse than for factorization, and good speedup cannot be expected.

2.3. Sparse Matrix Multiplication. Multiplication of the matrix M by a vector is required in steps 2, 5, 10, and 29 of the algorithm, as well as in orthogonalization and Ritz vector computation. Again, what is appropriate for a multivector processor may not be appropriate for a multiprocessor. On both machines, better performance can be obtained if symmetry is not exploited and if both halves of the matrix are stored. If these steps are not taken, a significant price in parallel performance is paid as a results of the cost of added synchronization and/or the use of inefficient operations. On the Encore Multimax an appropriate method to parallelize sparse matrix-vector multiplication is the straightforward inner product-based algorithm in which each processor computes a subset of the elements in the result vector. However, on the CRAY Y-MP the most efficient operation is a *saxpy* operation; therefore, a *saxpy*-based algorithm is used in which each processor computes a partial result for every element in the result vector, and then these partial results are combined at the end of the computation. If the number of processors is small, then this column-based algorithm is faster than the row-based algorithm because it uses the *saxpy* operation exclusively. Both methods will result in good speedup because little or no synchronization is required, plenty of work is available to divide up amongst the processors, and vectors lengths are unaffected by parallelization.

2.4. Solving $T_j s = \theta s$. At every step of the algorithm, the eigenvalues of T_j and their error bounds are computed so that the algorithm can be stopped when the desired eigenpairs have converged. Because the eigenvalues of T_{j-1}, θ_i, interlace those of T_j[5], and error bounds, bj_i, are known for the eigenvalues of T_{j-1}, an eigensolver that uses this information will be much more efficient than one that does not. A serial algorithm that finds the outermost eigenvalues of T_j is given in [16]. A parallel algorithm that uses all available information from the previous Lanczos step is shown in Figure 2. The first loop is used to find intervals that contain the eigenvalues of T_j. The second loop is used to compute each eigenvalue and its error bounds. Both loops in this algorithm can be partitioned among the processors and, therefore, can achieve a speedup of approximately j, where j is the size of the tridiagonal system. However,

[5] Cauchy's interlace theorem; see [15].

```
bounded[$i$] = 0 , for $i = 1, j$
Par Do $i = 1, j-1$
    if (($2^*\beta_{ji} < \theta_i - \theta_{i-1}$) and ($2^*\beta_{ji} < \theta_{i+1} - \theta_i$)) then
      probe = $\theta_i + \beta_{ji}$
      less = numless(probe)
      if (less = $i$) then
        bounded[$i$] = $i$
      else /* $i$ and $i+1$ are the only values numless will return; if
        it returns something else, a grave error has occurred */
        bounded[$i+1$] = $i$
      endif
    endif
enddo
Barrier
End Barrier
Par Do $i = 1, j$
    if (bounded[$i$] = 0) then
      leftbound = $\theta_{i-1}$
      rightbound = $\theta_i$
      newtonroot(leftbound,rightbound,new$\theta_i$,new$\beta_{ji}$)
    else if (bound[$i$] = $i$) then
      leftbound = $\theta_i - \beta_{ji}$
      rightbound = $\theta_i$
      newtonroot(leftbound,rightbound,new$\theta_i$,new$\beta_{ji}$)
    else if (bound[$i$] = $i-1$) then
      leftbound = $\theta_{i-1}$
      rightbound = $\theta_{i-1} + \beta_{ji-1}$
      newtonroot(leftbound,rightbound,new$\theta_i$,new$\beta_{ji}$)
    endif
enddo
```

numless determines the number of eigenvalues less than probe
newtonroot finds the eigenvalue (and its error bound) between leftbound and rightbound

FIG. 2. *Parallel tridiagonal eigensolver*

because the time required to find each eigenvalue often differs, it is unlikely that a speedup of j would be achieved.

2.5. Ritz Vector Computation. The assumption is made that n_e Ritz vectors are computed at the end of n_s steps. For each Ritz vector that must be calculated, a j-length eigenvector of T_j must be calculated by inverse iteration. This computation is very inexpensive because T_j is tridiagonal; the computation is not parallelized. The major computations used to compute a Ritz vector in LANZ are (1) $y_i = Q_j s_i$, which is a full matrix multiplication, and (2) the normalization of y_i to ensure that $y_i^T M y_i = 1$. The full matrix multiplication can be partitioned in a fashion similar to sparse matrix multiplication, with similar results expected. The normalization requires a sparse matrix multiplication, a vector inner product, and a vector division. Because the dominant computations, matrix multiplications, parallelize well, good speedup can be expected.

2.6. Orthogonalization. The algorithm given in Figure 1 already includes the extended local orthogonalization algorithm. At steps 4, 6, and 34, external selective orthogonalization will take place, *if* semi-orthogonality against eigenvectors computed in a previous Lanczos run must be maintained. The algorithm for external selective orthgonalization is given in [3] and is not given here. A j-length three-term recurrence is updated every step to check whether any eigenvectors must be orthogonalized against. If necessary, q_j and q_{j+1} are orthogonalized against some of the eigenvectors.

The partial reorthogonalization algorithm maintains semi-orthogonality among the Lanczos vectors and is described in [17]. A j-length three-term recurrence is updated, in parallel, every step to check whether q_j and q_{j+1} must be orthogonalized against *all* of the previous Lanczos vectors. This reorthogonalization usually occurs approximately once every three steps.

If q_j and q_{j+1} have been modified by reorthogonalization, then p_j and p_{j+1} are recomputed, and q_{j+1} is orthogonalized against q_j. These operations require two sparse matrix multiplications to recompute the p's, an inner product and a *saxpy* for the orthogonalization, and an inner product to recompute β_{j+1}.

Excellent speedup should be obtained during orthogonalization because the computations involve long vector operations and very little synchronization.

2.7. Other Computations. The rest of the computations in the Lanczos algorithm (e.g. steps 11 and 13) are vector operations of length n. These vectors are long enough to partition among a moderate number of processors without significantly reducing vector efficiency.

3. Performance Model. A parameterized parallel execution model of the computations in LANZ has been constructed to allow the prediction of performance on parallel computers of varying characteristics, as well as for problems of varying size and type. This model is based on the parallel version of LANZ outlined in the previous section. One application of the model is the comparison of the actual parallel performance of LANZ against the performance predicted by the model. This ensures that parallel implementations of LANZ are performing as expected. Two other applications of the model are prediction of performance on different architectures, and prediction of the effect of changes to LANZ without the actual implementation of the changes. Given the parameters listed in Figure 3 and the submodels given below, the cost of execution on p processors can be estimated using the following model:

$$(1) T(p) = t_f(p) + (n_s + 1)(t_{fs}(p) + t_{bs}(p)) + (n_s + 3)t_m(p) + t_{ec}(p) + t_r(p) + t_o(p).$$

Because of its complexity, the model is split into the following submodels:

$$(2) t_f(p) \quad = \quad \frac{n}{b}\{2f_s(p) + (\frac{b(b-1)c_m}{2} + \sum_{j=1}^{b}[\frac{j(j-1)}{2}(c_m + c_a)] +$$

Parameter	Description
n	order of the eigensystem
p	number of processors
β	average bandwidth of the linear system
α	average number of non-zeros per row of the linear system
b	average block size during the factorization
n_s	number of Lanczos steps
n_e	number of eigenvalues
$f_s(i)$	cost of synchronization given i processors
$c_{sx}(i,j)$	cost of i-length j size extended *saxpy* operation
$c_{ip}(i,j)$	cost of j simultaneous i-length vector inner products
$c_{vm}(i)$	cost of i-length vector multiplication
$c_{sax}(\alpha)$	cost of sparse *saxpy* operation with α non-zcroes
$c_{sip}(\alpha)$	cost of sparse inner-product operation with α non-zeroes
c_m	cost of single multiplication
c_a	cost of single addition

FIG. 3. *Parameters for the model*

$$\sum_{i=1}^{b} c_{sx}((\beta-b), i-1))\frac{1}{p}[(b(\beta-b)c_m) + ((\beta-b)c_{sx}(\frac{1}{2}(\beta-b), b))]\}$$

or

$$= \frac{n}{b}[f_s(p) + (\frac{b(b-1)c_m}{2} + \sum_{j=1}^{b}\frac{j(j-1)}{2}(c_m+c_a) +$$

$$\frac{1}{p}(\sum_{i=1}^{b} c_{sx}(\frac{(\beta-b)}{p}, i-1) + (b(\beta-b)c_m) + ((\beta-b)c_{sx}(\frac{1}{2}(\beta-b), b)))]$$

$$(3)\ t_{fs}(p) = \frac{n}{b}[f_s(p) + \frac{b(b-1)}{2}(c_m+c_a) + c_{sx}(\frac{\beta-b}{p}, b)]$$

$$(4)\ t_{bs}(p) = \frac{n}{b}[f_s(p) + \frac{b(b-1)}{2}(c_m+c_a) + b(p-1)c_a + c_{ip}(\frac{\beta-b}{p}, b)]$$

$$(5)\ t_m(p) = \frac{2n}{p}c_{sax}(\alpha) + f_s(p) + c_{sx}(\frac{n}{p}, p-1)$$

or

$$= \frac{2n}{p}c_{sip}(\alpha)$$

$$(6)\ t_{ec}(p) = n_e(n_s c_{sx}(\frac{n}{p},1) + t_m(p) + c_{ip}(\frac{n}{p},1) + c_{vm}(\frac{n}{p}) + 2f_s(p))$$

$$(7)\ t_r(p) = (n_s+3)[2t_m(p) + 2c_{ip}(\frac{n}{p},1) + c_{sx}(\frac{n}{p},1)n_s(c_{ip}(\frac{n}{p},1) + c_{sx}(\frac{n}{p},1))]$$

$$(8)\ t_o(p) = (6n_s+1)c_{ip}(\frac{n}{p},1) + (2n_s+2)c_{vm}(\frac{n}{p}) + 4n_s c_{sx}(\frac{n}{p},1) + (2n_s+2)f_s(p)$$

Two versions of the cost of a single factorization, $t_f(p)$, are given Equation 2: (1) when the updating of the pivot columns is not parallelized, e.g., on the CRAY Y-MP, and (2) when the updating of the pivot columns is parallelized, e.g., on the Encore Multimax. The models for triangular matrix solution, $t_{fs}(p)$ and $t_{bs}(p)$, are given in

Equations 3 and 4.

Two models for the cost of the multiplication of a sparse matrix times a vector, $t_m(p)$, are given in Equation 5: (1) the *saxpy*-based algorithm, and (2) the inner product-based algorithm. The model for the cost of computing a Ritz vector, $t_{ec}(p)$, is given in Equation 6. The assumption is made that n_e Ritz vectors are computed at the end of n_s steps. The term $t_r(p)$ accounts for the cost of partial reorthogonalization. Because the cost of external selective orthogonalization depends on the number, often 0, of previously converged eigenvectors, it will not be included in the model.

The term $t_o(p)$ accounts for the length n vector operations and synchronizations given in Fig ıre 1. The cost of the rest of the operations in the algorithm is small compared to those costs modeled and is therefore left out.

4. Performance Results and Analysis.

One application for this model is to ensure that the parallel implementation of LANZ is performing as expected. For a comparison of the model against the implementation, LANZ was run on a medium size eigenproblem from structural engineering[6] where the ten lowest eigenpairs were found in 22 steps on a four processor CRAY Y-MP. A smaller problem[7] was run on a twenty processor Encore Multimax in which the ten lowest eigenpairs were found in 22 steps. An examination of the speedup curves in Figure 4 reveals that the speedups from the implementations are very close to those predicted by the model.

It is clear from the speedup curves in Figure 4 that a speedup plateau occurs. The main cause of this plateau is the poor speedup realized in the forward and backward matrix solution algorithms. This problem can best be observed by plotting the percentage of execution time taken by forward and backward matrix solution algorithms as the number of processors increases. These percentages are shown in Figure 5 for Lanczos runs taking 10, 25, and 50 steps on the Encore Multimax using the same problem described in the previous section. The problem caused by the matrix solution algorithms is exacerbated as the number of Lanczos steps increases, because each Lanczos step requires another forward/backward matrix solution, taking more and more time as compared to factorization, which speeds up well.

This plateau occurs later on the Encore than the CRAY because the Encore does not have to contend with the conflict between vectorization and parallelization: vector lengths decreasing as the number of processors increases. However, both implementations suffer from the poor ratio of computation to synchronization in the forward and backward matrix solution algorithms.

[6] Finding the vibration modes and mode shapes of the finite element model of a circular cylindrical shell [20]. In this problem $n = 12054$ and the average semi-bandwidth is 394.

[7] Finding the five lowest buckling modes and mode shapes of the finite element model of an I-stiffened panel. In this problem $n = 4474$ and the average semi-bandwidth was 207.

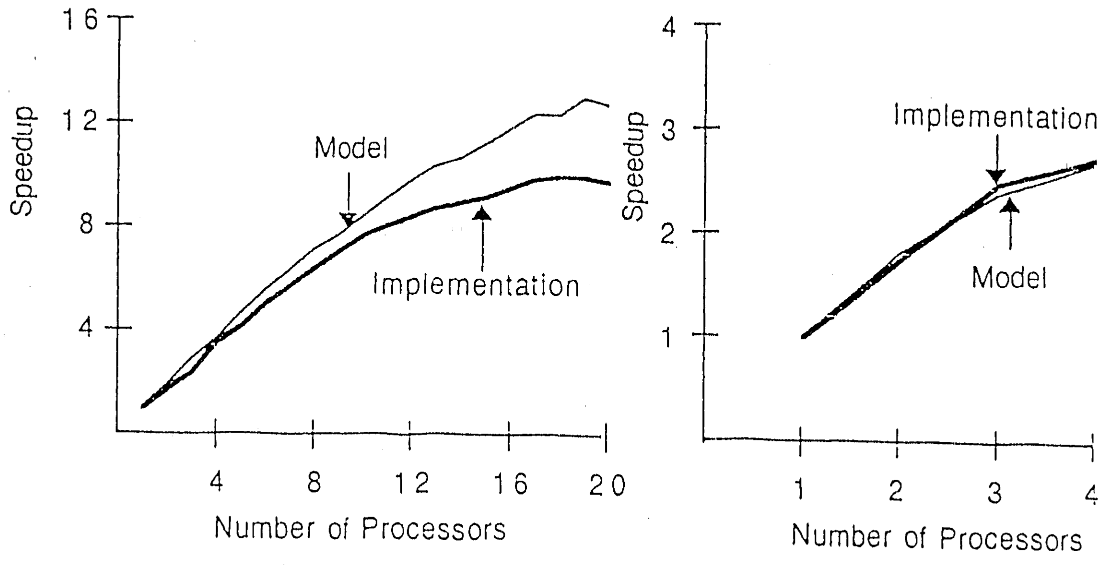

FIG. 4. *Speedup curves*

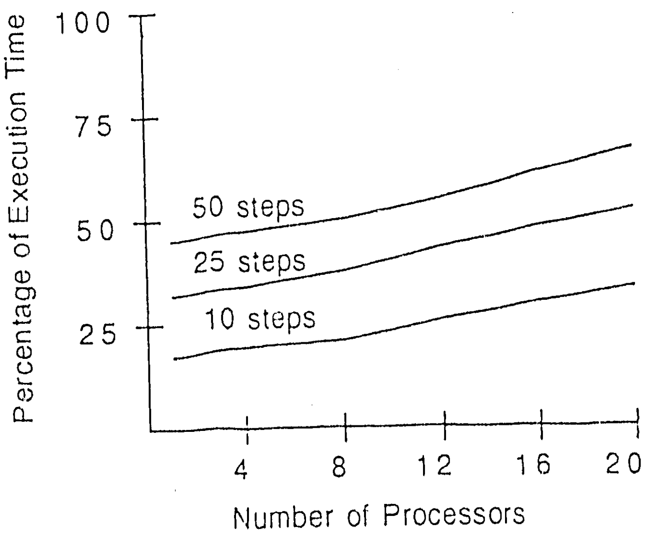

FIG. 5. *Matrix solution as a percentage of execution time*

If problems with larger bandwidths were used, better speedup from these algorithms could, of course, be expected. It has been the authors' experience, however, that if the bandwidth arising from a structural engineering problem is large, then most likely many zeroes exist inside the band, and therefore sparse methods are best used.

5. Concluding Remarks. A parallel Lanczos algorithm for finding a few of the eigenpairs around a point in the eigenspectrum was described. Differences in the implementation of the algorithm on a multivector processor and on a multiprocessor were described. The algorithm was shown to perform reasonably well on a moderate number of processors. The algorithm was analyzed by using an execution-time model, and the performance bottleneck which prevents efficient utilization of a large number of processors was identified.

Several possible modifications to the LANZ algorithm can be used to improve its parallel performance. The use of dynamic shifting [2] to improve parallelism by reducing the number of forward and backward matrix solutions was investigated in [3] and was found to be successful when the eigenvalue distribution was difficult. The use of groups of processors executing the LANZ algorithm independently at different shifts was investigated in [3] and was found to be successful when many eigenpairs are being sought. Block Lanczos holds some promise because it allows several forward and backward matrix solutions to occur simultaneously [2]. The improvement in performance resulting from block Lanczos will depend on how many Lanczos steps are eliminated and what block size can be effectively used. Unfortunately, s-step Lanczos methods [10] will not alleviate the bottleneck imposed by forward and back solutions and, therefore, will not have a significant effect on performance.

Another avenue for improving parallel performance is the use of iterative methods, such as SYMMLQ [14], to solve $(K - \sigma M)x = y$ rather than direct methods. However, it has been the authors' experience that $(K - \sigma M)$ is often poorly conditioned and, therefore, is difficult to solve by iterative methods.

REFERENCES

[1] J. J. Dongarra and A. R. Hinds, *Unrolling Loops in FORTRAN*, Software: Practice and Experience, 9 (1979), pp. 219–226.

[2] R. G. Grimes, J. G. Lewis, and H. D. Simon, *The Implementation of a Block Lanczos Algorithm with Reorthogonalization Methods*, ETA-TR-91, Boeing Computer Services, Seattle, Washington, May 1988.

[3] M. T. Jones, *The Use of Lanczos' Method to Solve the Generalized Eigenproblem*, PhD thesis, Department of Computer Science, Duke University, 1990.

[4] M. T. Jones and M. L. Patrick, *Bunch-Kaufman Factorization for Real Symmetric Indefinite Banded Matrices*, Technical Report 89-37, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Va., 1989.

[5] ——, *The Use of Lanczos's Method to Solve the Large Generalized Symmetric Definite Eigenvalue Problem*, Technical Report 89-67, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Va., 1989.

[6] ———, *Factoring Symmetric Indefinite Matrices on High-Performance Architectures*, Technical Report 90-8, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Va., 1990.

[7] ———, *LANZ: Software for Solving the Large Sparse Symmetric Generalized Eigenproblem*, Preprint MCS-P158-0690, MCS Division, Argonne National Laboratory, Argonne, Il., 1990. Also available as ICASE Interim Report no. 12.

[8] M. T. JONES, M. L. PATRICK, AND R. G. VOIGT, *Language Comparison for Scientific Computing on MIMD Architectures*, Report No. 89-6, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Va., 1989.

[9] H. JORDAN, *The Force*, Computer Systems Design Group, University of Colorado, 1987.

[10] S. KIM AND A. CHRONOPOULOS, *A Class of Lanczos-Like Algorithms Implemented on Parallel Computers*, Computer Science Department 89-49, University of Minnesota, July 1989.

[11] J. G. LEWIS, *Algorithms for Sparse Matrix Eigenvalue Problems*, PhD thesis, Department of Computer Science, Stanford University, 1977.

[12] B. NOUR-OMID, B. N. PARLETT, T. ERICSSON, AND P. S. JENSEN, *How to Implement the Spectral Transformation*, Mathematics of Computation, 48 (1987), pp. 663–673.

[13] B. NOUR-OMID, B. N. PARLETT, AND R. L. TAYLOR, *Lanczos Versus Subspace Iteration For Solution of Eigenvalue Problems*, International Journal for Numerical Methods in Engineering, 19 (1983), pp. 859–871.

[14] C. C. PAIGE AND M. A. SAUNDERS, *Solution of Sparse Indefinite Systems of Linear Equations*, SIAM Journal of Numerical Analysis, 12 (19[illegible]), pp. 617–629.

[15] B. N. PARLETT, *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

[16] B. N. PARLETT AND B. NOUR-OMID, *The Use of a Refined Error Bound When Updating Eigenvalues of Tridiagonals*, Linear Algebra and its Applications, 68 (1985), pp. 179–219.

[17] B. N. PARLETT, B. NOUR-OMID, AND Z. A. LIU, *How to Maintain Semi-Orthogonality Among Lanczos Vectors*, PAM-420, Center for Pure and Applied Mathematics, University of California, Berkeley, July 1988.

[18] E. POOLE, *The Solution of Linear Systems of Equations with a Structural Analysis Code on the NAS Cray 2*, Tech. Rep. CR-4159, CSM Branch, NASA Langley Research Center, Hampton, Va., 1988.

[19] H. D. SIMON, *The Lanczos Algorithm With Partial Reorthogonalization*, Mathematics of Computation, 42 (1984), pp. 115–142.

[20] C. B. STEWART, *The Computational Structural Mechanics Testbed Procedure Manual*, Computational Structural Mechanics Branch, NASA Langley Research Center, 1989.