NASA Contractor Report 182072
ICASE Report No. 90-48

ICASE

THE USE OF LANCZO'S METHOD TO SOLVE THE LARGE GENERALIZED SYMMETRIC EIGENVALUE PROBLEM IN PARALLEL

Mark T. Jones
Merrell L. Patrick

Contract No. NAS1-18605
July 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665-5225

The Use of Lanczos' Method to Solve the Large Generalized Symmetric Eigenvalue Problem in Parallel

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

Abstract

The generalized eigenvalue problem, $Kx = \lambda Mx$, is of significant practical importance, especially in structural engineering where it arises as the vibration and buckling problems. New software, **LANZ**, based on Lanczos' method has been developed for solving these problems and runs on SUN 3, SUN 4, Convex C-220, Cray 2, and Cray Y-MP systems.

Preliminary results of using the Force to obtain a multiprocessor implementation of **LANZ** on MIMD parallel/vector systems are reported here. A parallel execution time model of **LANZ** is defined and used to predict the performance of **LANZ** as well as examine hypothetical modifications to **LANZ**. The results of using dynamic shifting to improve parallelism are presented. Finally, the results of assigning a group of processors to separate shifts and finding all the desired eigenvalues using **LANZ** in parallel are reported.

[*]Department of Computer Science, Duke University, Durham, NC 27706

[†]

This research was supported by the National Aeronautics and Space Administration under NASA contract No. NAS1-18605 and AFOSR grant No. 88-0117 while the authors were in residence at ICASE. Additional support was provided by NASA grant No. NAG-1-466.

1 Introduction

The generalized eigenvalue problem, $Kx = \lambda Mx$, is of significant practical importance, especially in structural engineering where it arises as the vibration and buckling problems. New software, LANZ, based on Lanczos' method has been developed for solving these problems and been reported on in [1] [2]. LANZ uses a technique called dynamic shifting to improve the efficiency and reliability of the Lanczos algorithm [3]. Improved methods for solving symmetric indefinite linear systems and for finding eigenvalues of the tridiagonal matrices that arise when using Lanczos' method have been developed for use in LANZ [4]. Loop unrolling techniques were used to obtain improved performance on vector processing machines [5]. An improved version of Parlett and Scott's selective orthogonalization algorithm [6] was used to maintain orthogonality of the Lanczos vectors. Implementations of LANZ on a Convex C-220 were used to study the performance of LANZ and compare it with a subspace iteration code used by structural engineers. In all cases tested LANZ had superior performance to the subspace iteration code.

In this work, the Force is used to obtain a multiprocessor implementation of LANZ on MIMD parallel/vector systems. Strengths and weaknesses of the Force as a parallel programming language are examined in Section 2. A parallel execution time model of LANZ is defined and used to compare the actual parallel performance of LANZ against the performance predicted by the model in Section 3. It is also used to predict the effect of changes to LANZ without the actual implementation of the changes, e.g., it is used to compare the performance of vectorized versus non-vectorized versions of LANZ on a Cray Y-MP. Backward triangular matrix solution is shown to be a bottleneck in the parallel version of LANZ and the tradeoffs involved in designing algorithms for a parallel/vector machine are discussed in Section 4. The results of using dynamic shifting to reduce the number of backward solves by increasing the number of factorizations are reported in Subsection 5.1. Finally, the results of assigning groups of processors to work in parallel on separate shifts are reported in Subsection 5.2.

2 Parallel Implementation

LANZ consists of over 15,000 lines of FORTRAN source code with four major computations: 1) factorization, 2) triangular matrix solution, 3) matrix-vector multiplication, and 4) computation of the eigenvectors. Observation of the runtime profile shown in Figure 1 of LANZ on one processor of the Cray Y-MP reveals that over 97 percent of the execution time is spent on these four computations.[1]

[1] For this profile, LANZ took fifteen steps to find the five lowest eigenvalues.

Computation	Time (seconds)	Percentage
Factorization	13.98	67.9
Matrix Multiplication	3.09	15.0
Triangular Matrix Solution	2.73	13.3
Computation of Eigenvectors	0.27	1.4
Other	0.49	2.4

Figure 1: Execution profile of **LANZ** on a single processor of the Cray Y-MP

The parallelization of **LANZ** focuses on these four computations. Two constraints directed the selection of a parallel language to be used in carrying out the parallelization: 1) the language must require little rewriting of the non-parallel source code, and 2) the resulting implementation should be transportable to several shared-memory parallel architectures.

The Force is a set of extensions to FORTRAN that provides a shared memory model of parallel processing on MIMD shared memory architectures [7]. The Force includes constructs for both fine- and coarse-grain parallelism. The Force starts a process on each processor. Each Force process communicates through shared variables and synchronizes using barriers and critical regions. Loop iterations are partitioned among Force members by prescheduling or self-scheduling constructs. Subroutines must be declared as Forcesubs if Force constructs are to be used within them. These "Forcesub"s are called using Forcecall statements and all the processes must encounter this Forcecall statement and execute the subroutine. The Force is implemented as a preprocessor to FORTRAN on the following shared-memory computers: Cray Y-MP, Cray/2, Cray X-MP, Flex/32, Alliant FX/Series, Convex C220, Sequent Balance, and the Encore Multimax. The Force has been shown to be useful for implementing parallel linear algebra algorithms [8].

The use of the Force to parallelize such a large program revealed a shortcoming in the language. The Force assumes that all the processors are simultaneously executing a segment of code unless a synchronization construct has specified otherwise. This is a useful model when implementing a short algorithm such as a factorization subroutine [8]. However, in a large, practical code, such as **LANZ**, which contains several different algorithms and a fair amount of i/o, this is not the best model for computation. Most of the source code in such a large application is executed very few times, if at all, and is inherently sequential. The major portion of the execution time is spent in a few subroutines that execute factorization or matrix multiplication algorithms. In these subroutines the Force model of parallel computation is the preferred one. Using the Force required a large number of unnecessary synchronization constructs in the inherently sequential code. Although this did not have a large impact on performance, it was a time-consuming and error-prone programming task. Also, the Force does not allow a set of processors to execute one segment of code in parallel using barriers to synchronize while another set of processors executes a different segment of code. The barriers in the Force assume that all or none of the processors will encounter each barrier.

The addition of a few capabilities to the Force language would alleviate these problems and provide added functionality. If an entire subroutine could be designated as sequential, and yet retain the ability to declare shared variables, then the need for synchronization constructs in inherently sequential code would be alleviated. The ability to call a Force subroutine from a subroutine designated as sequential would also be necessary. For example, the Force requires that the main program in **LANZ** be declared a parallel routine so that it has the capability to declare shared variables and call subroutines that will be executed in parallel. However, nothing in the main program should be executed in parallel, therefore, several Barrier statements are required to force the code to execute sequentially. The ability to declare the main program to be sequential would alleviate the need for unnecessary Barrier statements and still allow the main program to declare shared variables and call parallel subroutines. The Forcecall subroutine should be extended to allow the number of processes executing the subroutine to be specified. This would allow sets of processors to execute different segments of code in parallel. The barrier construct would also have to be extended to accommodate situations in which all the processors are not executing the same code. The utility of this addition is illustrated in Subsection 5.2. The addition of these constructs would be very useful, but may be difficult or impossible to implement as a preprocessor.

In addition, the Barrier synchronization construct provided by the Force proved to be too expensive for use in the loops that occur in the factorization subroutine. In its place, a user-implemented construct which was much less expensive was used. The Force barrier can not take advantage of the fact that it resides in a loop, whereas, the user-implemented construct is specifically designed to execute in a loop. Conceptually, the synchronization desired is shown in Figure 2. If implemented using the Force, line 2 becomes a "Barrier" statement and line 4 becomes an "End barrier" statement. These statements are expanded by the preprocessor and the code is shown in Figure 3, where "LOCKON" and "LOCKOFF" are provided by the operating system and are very expensive. The user-implemented construct uses a Force "Barrier" before execution to initialize variables and then uses the shared array "commun" to synchronize. The loop as implemented using the user-implemented construct is shown in Figure 4, where "me" is a processor's number and "nprocs" is the number of processors. The user-implemented construct has the advantage of not requiring a call to an operating system routine.[2] The construct is not a

[2]The initialization step does require such a call.

```
1)      for i = 1, n do
2)         execute code in parallel
3)         wait for all processors to reach this point
4)         execute code with one processor
5)         tell all processors to resume execution at this point
6)         execute code in parallel
7) 10   continue
```

Figure 2: Loop with desired synchronization

```
C       assume that numhere is 0, the lock lockvar is off
C       and the lock lockwait is on
1)      do 10 i = 1, n
2)         execute code in parallel
3a)        call lockon(lockvar)
3b)        if (numhere.lt.(nprocs - 1)) then
3c)           numhere = numhere + 1
3d)           call lockoff(lockvar)
3e)           call lockon(lockwait)
3f)        endif
3g)        if (numhere .eq. (nprocs-1))  then
4)            execute code with one processor
5a)        endif
5b)        if (numhere.eq.0) then
5c)           call lockoff(lockvar)
5d)        else
5e)           numhere = numhere - 1
5f)           call lockoff(lockwait)
5g)        endif
6)         execute code in parallel
7)  10  continue
```

Figure 3: Loop with Force synchronization

```
0a)        commun(me) = 0
0b)        Barrier
0c)        End barrier
1)         do 10 i = 1, n
2)            execute code in parallel
3a)           if (me.eq.1) then
3b)              j = 2
3c) 20           continue
3d)              do 30 j = j, nprocs
3e)                 if (commun(j).ne.i) goto 20
3f) 30           continue
4)               execute code with one processor
5a)              commun(1) = i
5b)           else
5c) 40           continue
5d)              commun(me) = i
5e)              if (commun(1).ne.i) then
5f)                 goto 40
5g)              endif
5h)           endif
6)            execute code in parallel
7)  10     continue
```

Figure 4: Loop with user-implemented synchronization

general substitute for the "Barrier-End barrier" construct of the Force because one initialization step must take place for every user-implemented construct. This initialization step requires the equivalent of a Force "Barrier." The user-implemented construct is superior when used in a loop because the cost of the initialization step is amortized over the number of iterations in the loop. Extensive experiments have shown that the user-implemented construct is superior to the Force "Barrier" and to the synchronization constructs available in the Cray autotasking package on both the Cray Y-MP and Cray-2 when executing in dedicated mode[9].

3 Parallel Execution Model of LANZ

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 the parallel implementation of **LANZ** is performing as expected. Two other applications of the model are: 1) prediction of performance on different architectures, and 2) prediction of the effect of changes to **LANZ** without the actual implementation of the changes. Given the parameters listed in Figure 5, the cost of execution on p processors can be estimated using the following model:

$$T(p) = n_f t_f(p) + n_s t_{fs}(p) + n_s t_{bs}(p) + n_s t_{mm}(p) + n_e t_{ec}(p) + t_o(p). \qquad (1)$$

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

$$\begin{aligned}
t_f(p) &= \frac{n}{as}[f_s(p) + (\frac{as(as-1)c_m}{2} + \sum_{j=1}^{as} \frac{j(j-1)}{2}(c_m + c_a) + \qquad (2) \\
&\quad \sum_{i=1}^{as} c_{sx}((\beta - as),(i-1))) + \\
&\quad \frac{1}{p}((as(\beta - as)c_m) + ((\beta - as)c_{sx}(\frac{1}{2}(\beta - as), as)))] \\
t_{fs}(p) &= \frac{n}{as}[f_s(p) + \frac{as(as-1)}{2}(c_m + c_a) + c_{sx}(\frac{\beta - as}{p}, as)] \qquad (3) \\
t_{bs}(p) &= \frac{n}{as}[f_s(p) + \frac{as(as-1)}{2}(c_m + c_a) + as(p-1)c_a + \qquad (4) \\
&\quad c_{ip}(\frac{\beta - as}{p}, as)] \\
t_{mm}(p) &= \frac{2n}{p}c_{sax}(\alpha) + f_s(p) + c_{sx}(\frac{n}{p}, p-1) \qquad (5) \\
t_{ec}(p) &= \frac{n_s}{2}c_{sx}(\frac{n}{p}, 1) + (t_{mm} + c_{ip}(n,1) + c_{vm}(n)) \qquad (6) \\
t_o(p) &= [2c_{vm}(n) + 4c_{sx}(n,1) + 5c_{ip}(n,1)] + \qquad (7) \\
&\quad min(\frac{n_s}{2}, n_e)(t_{mm}(p) + 2c_{ip}(n,1) + c_{vm}(n) + c_{sx}(n,1)).
\end{aligned}$$

The operation, extended saxpy of size j, is used in the submodels and is defined as

$$x = x - \sum_{i=1}^{j} a_i y_i, \qquad (8)$$

where x is a vector, a_i is a scalar quantity, and each y_i is a separate vector. The extended saxpy operation takes full advantage of the vector processors on the Cray Y-MP.

The cost of a single factorization,[3] $t_f(p)$, is given Equation 2. The extended saxpy operations in the second line of the equation dominate the computational costs. To parallelize the computation, at each of the $\frac{n}{as}$ steps a processor is given $\frac{\beta - as}{p}$ extended saxpys to compute. Because this part of the computation parallelizes well and dominates the execution cost, good parallel speedup is expected in the factorization algorithm.

The models for triangular matrix solution, $t_{fs}(p)$ and $t_{bs}(p)$, are given in Equations 3 and 5.[4] These computations do not parallelize nearly as well as factorization. Two reasons for this poor parallelization are: 1) the ratio of computation to synchronization is much lower than for factorization, and 2) as the number of processors, p, increases, the efficiency of the vector operations in this computation deteriorates. The reason for this deterioration in vector performance is now given.

The dominant cost for forward triangular matrix solution is the extended saxpy operation. This operation has been parallelized by splitting the vectors into p pieces, thereby decreasing the vector lengths as p increases. Thus, as the parallelism increases, the vector efficiency decreases.

The case is somewhat more complicated for backward triangular matrix solution because the dominant operations are j inner products.[5] This analysis is based on the assumption that the inner products have been parallelized by splitting the vectors into p pieces as was done for forward triangular matrix solution. If this is done, then the same analysis that was used in forward triangular matrix solution holds.

Another approach is possible, however, if the size of the matrix-matrix operations, j, can be arbitrarily specified.[6] The alternative approach is to assign $\frac{j}{p}$, where j is evenly divisible by p, inner products to each processor to compute in parallel. Thus, the vector lengths are unchanged as p increases. However, this approach has two drawbacks: 1) if j is not evenly divisible by p, then poor load balancing occurs, and 2) a processor on the Y-MP can compute several inner products simultaneously more efficiently than it can one or two inner products, therefore if $\frac{j}{p}$ is small, the operations will be inefficient.

A model, $t_{mm}(p)$, for the cost of the multiplication of a sparse matrix [7] times

[3]This submodel is constructed under the assumption that a variable banded factorization algorithm utilizing matrix-matrix operations is being used. The advantages of matrix-matrix operations are discussed in [2].

[4]The assumption is again made that matrix-matrix operations will be used.

[5]j is the size of the blocks in the matrix-matrix operations.

[6]The block size can be arbitrarily specified if a symmetric positive definite matrix is being factored, but not if a symmetric indefinite matrix is being factored. This is discussed in more detail in [2].

[7]The assumption is made that M, if the vibration problem is considered, and K_G, if the buckling problem is considered, are sparse.

a vector is given in Equation 5. The computation is parallelized by assigning columns of the matrix to each processor. Each processor uses saxpy operations to compute the contribution of each of its columns to the result vector. Each processor stores the sum of the contributions of its columns in its own vector. After all the processors have finished computing their own vectors, each computes a portion of the result vector from the contributions of each processor using saxpy operations. Because only one synchronization step is required, a speedup of almost p on p processors can be expected.

This algorithm is superior to the obvious algorithm of assigning rows of the matrix to each processors and letting each processor compute a single element of the result vector. Although the obvious algorithm requires no synchronization, it does require the use of vector inner products which are far slower than the saxpy operations used in the previous algorithm. For large p the cost of communicating the partial results would become too expensive and the obvious algorithm would be superior.

The model for the cost of computing an eigenvector, $t_{ec}(p)$, is given in Equation 6. The major computations used to compute an eigenvector 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 to be 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.

The model for the other computations in **LANZ**, $t_o(p)$, includes the cost of the n-length vector operations. These operations have not been parallelized in **LANZ**. The cost of reorthogonalization is reflected in the second group of terms in Equation 8. Because the number of reorthogonalizations varies so widely between problems, this model approximates the cost based on the number of Lanczos steps and the number of eigenvalues computed. Because sparse matrix multiplication is the dominant computation in this portion of $t_o(p)$, good speedup can be expected from it.

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, n=12054, where the ten lowest eigenvalues were sought on an eight processor Cray Y-MP. An examination of Figure 6 reveals that the implementation times are very close to the predicted times from the model.

4 Analysis of the Parallel Implementation

When the performance data from **LANZ** in Figure 6 is transformed into the speedup curve in Figure 7, a plateau in the speedup curve can be observed as p exceeds four; there are several reasons for this early plateau.

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
as	average block size during the factorization
$f_s(i)$	cost of synchronization given i processors
$c_{ip}(i,j)$	cost of j simultaneous i-length vector inner products
$c_{sax}(\alpha)$	cost of sparse saxpy operation with α non-zeroes
c_m	cost of single multiplication
$c_{vm}(i)$	cost of i-length vector multiplication
c_a	cost of single addition
$c_{sx}(i,j)$	cost of i-length j size extended saxpy operation
n_s	number of Lanczos steps
n_e	number of eigenvalues
n_f	number of factorizations
$t_f(p)$	time for 1 factorization on p processors
$t_{fs}(p)$	time for 1 forward triangular matrix solution on p processors
$t_{bs}(p)$	time for 1 backward triangular matrix solution on p processors
$t_{mm}(p)$	time for 1 sparse matrix multiplication on p processors
$t_{ec}(p)$	time for 1 eigenvector calculation on p processors
$t_o(p)$	time for the other calculations in LANZ

Figure 5: Parameters required by parameterized model

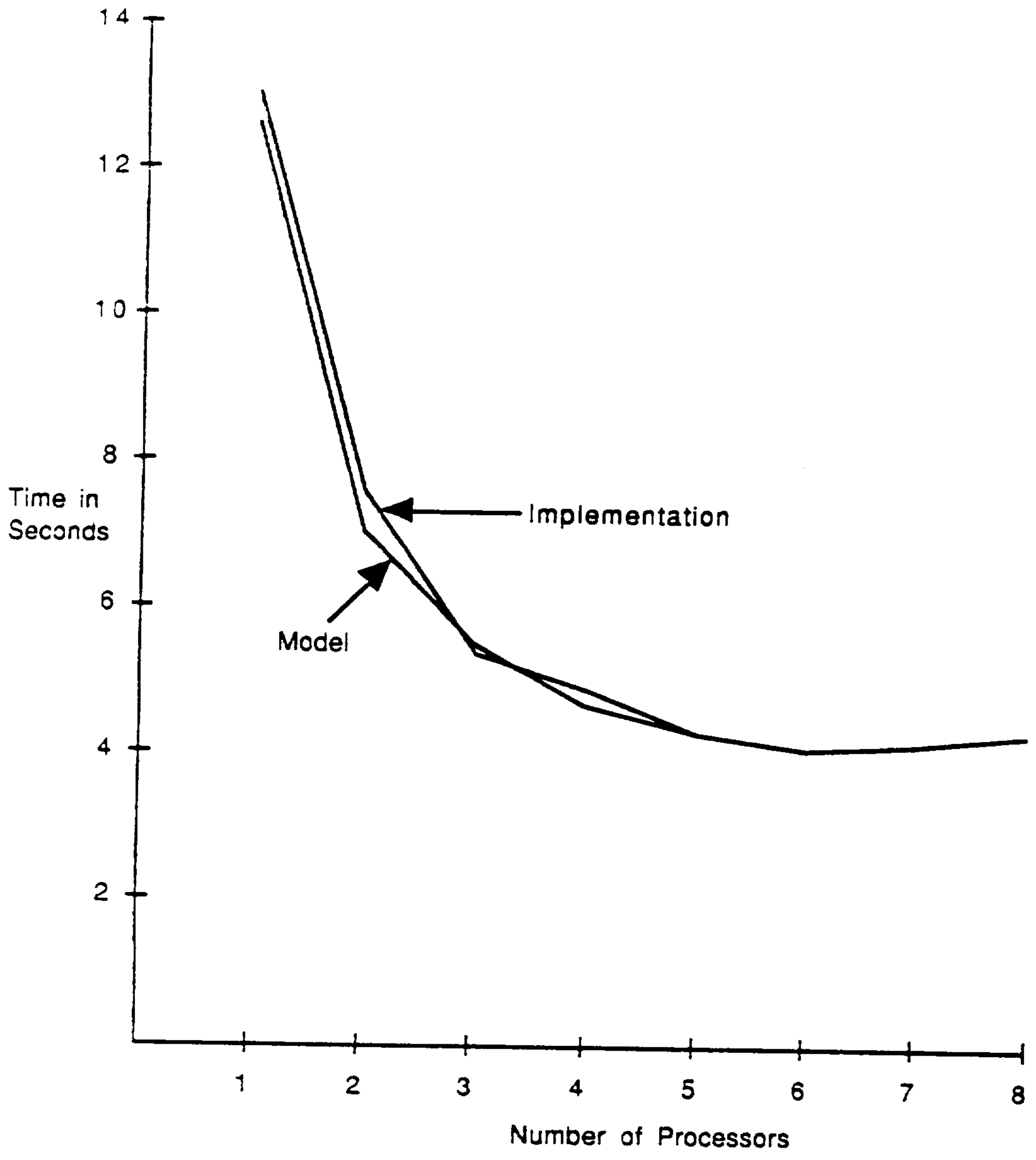

Figure 6: Comparison of the implementation vs. the model on a Cray Y-MP

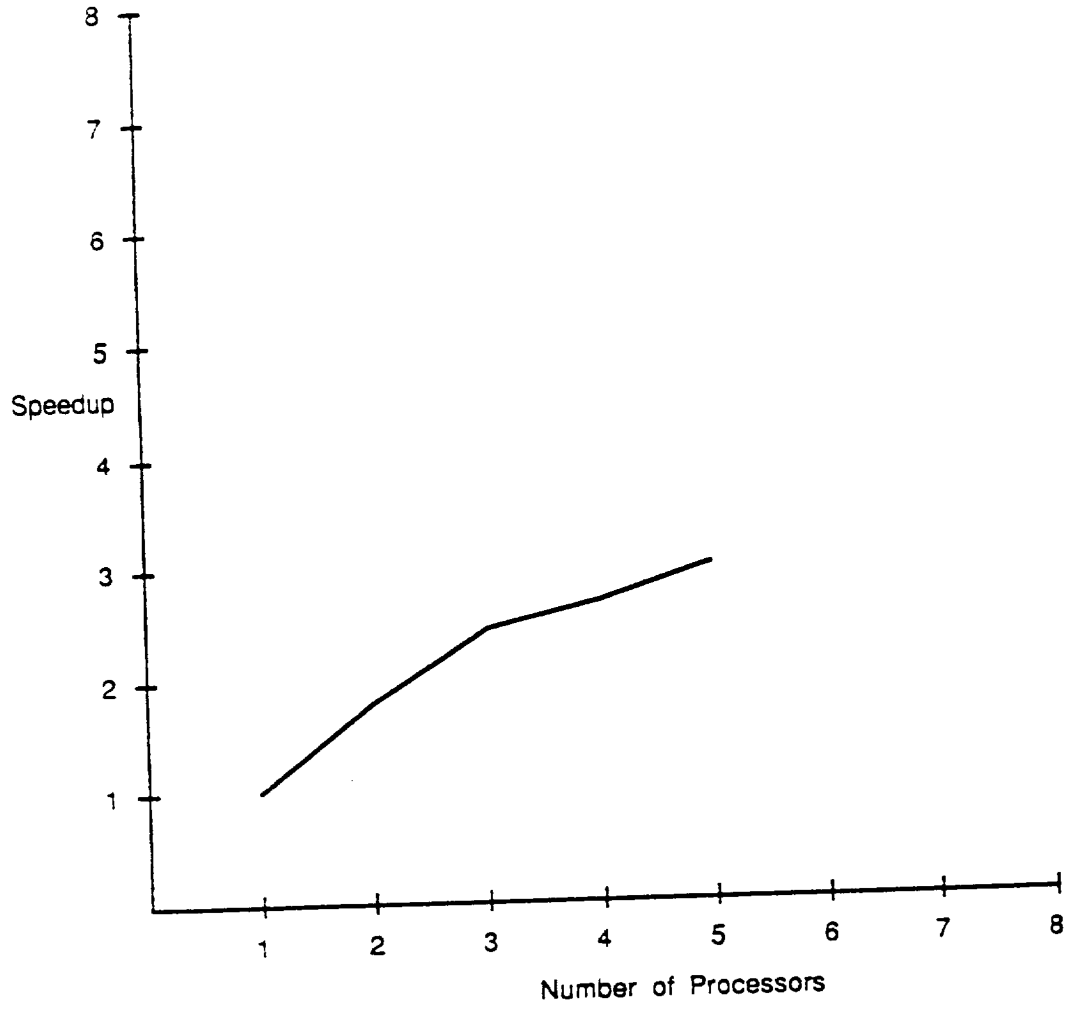

Figure 7: Speedup curve of **LANZ** on a Cray Y-MP

First, the parallel implementation of **LANZ** did not sacrifice the vector performance of the code in order to obtain better parallel speedup. If this had been done, then the speedup of the parallel implementation on p processors over the parallel version on one processor would be excellent; however, the uniprocessor vector implementation would be faster than the parallel implementation on four processors. The multiple segmented computational units in each processor of the Cray Y-MP are, in essence, usurping much of the parallelism inherent in **LANZ**. The plateau in the speedup curves for **LANZ** would occur later if vectorization was not used. To illustrate this point, parameters for the model were gathered from the Cray Y-MP with vectorization and without vectorization. The performance of the vectorized and unvectorized versions is plotted in Figure 8. From this figure it is clear that the plateau in speedup can be delayed significantly. This would be foolish, however, because the vectorized version is always faster than the unvectorized version. The ultimate goal should be to minimize execution time, rather than to maximize parallel efficiency.

Second, a comparison of the sequential execution profile in Figure 1 with the parallel execution profile in Figure 9 of the same computation on four processors of the Cray Y-MP reveals that the triangular matrix solutions are the bottleneck in the parallel version of **LANZ**. The percentage of execution time required for triangular matrix solve and the sequential computations increased from 16 percent on a single processor to 38 percent on four processors. This is not unexpected, because it was stated in the analysis of the implementation in the previous section that triangular matrix solution does not parallelize well. Two reasons for this poor parallelization can be observed in the submodels for forward and back triangular matrix solution: 1) the low ratio of computation to synchronization, and 2) the decreasing vector length as p increases, resulting in a degradation of vector performance.

Third, the type of problem being run will be a significant factor in determining the speedup. If the problem requires a large number of Lanczos steps and only one factorization to converge to the sought after eigenvalues, then the triangular matrix solutions will dominate the runtime and cause poor speedup. However, if the eigenvalues in a particular frequency range are being sought, at least two factorizations will be necessary and, possibly, only a few Lanczos steps will be required, resulting in excellent speedup. Therefore, the distribution of the eigenvalues as well as which eigenvalues are being sought will affect the speedup.

5 Improving Parallel Performance

Two approaches for improving the parallel performance of **LANZ** will be described. These approaches improve performance by either: 1) reducing the number of Lanczos steps taken and therefore reducing the number of triangular matrix solutions, or 2) allowing more than one triangular matrix solution to

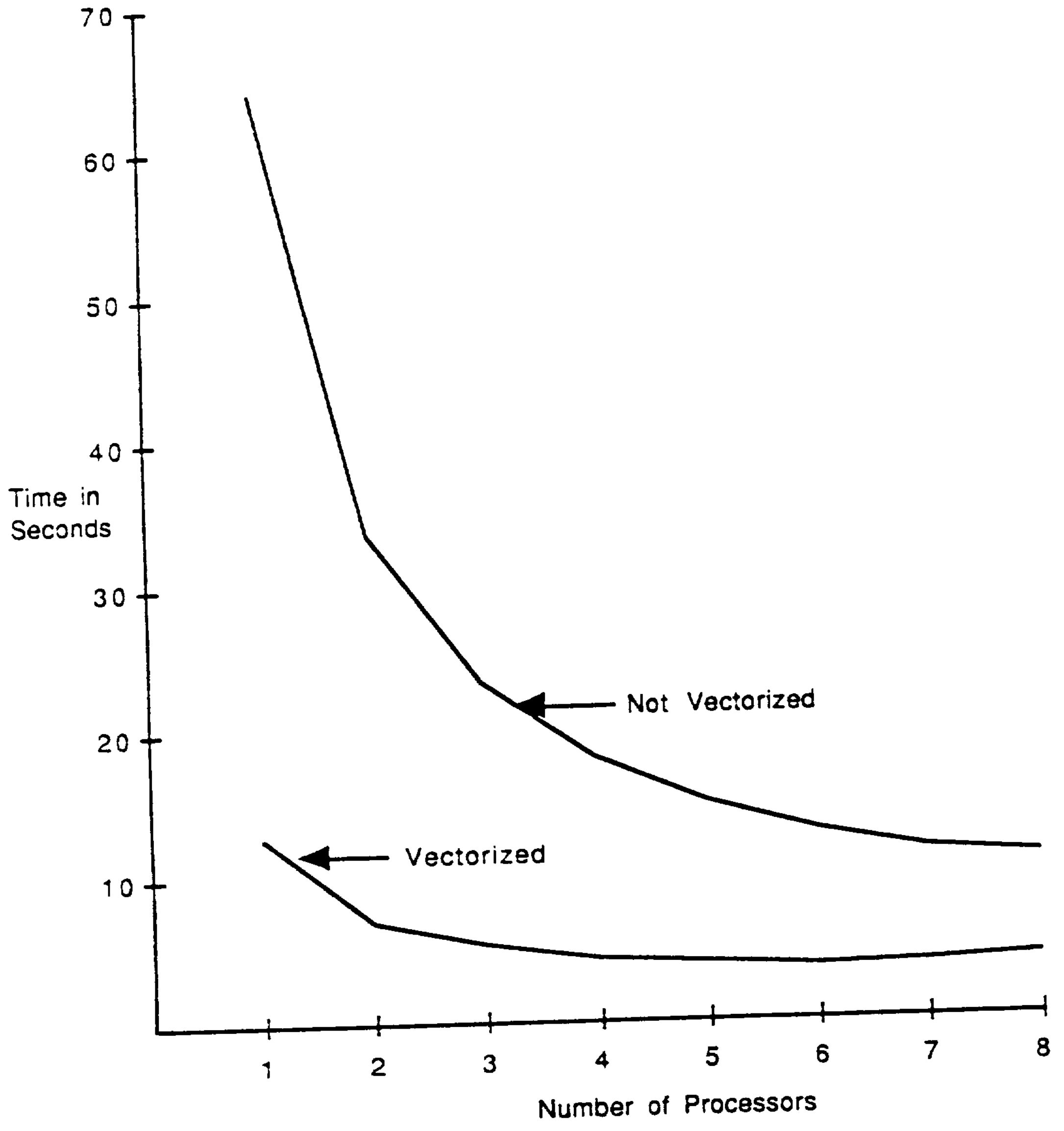

Figure 8: Estimated performance of vectorized vs. unvectorized versions

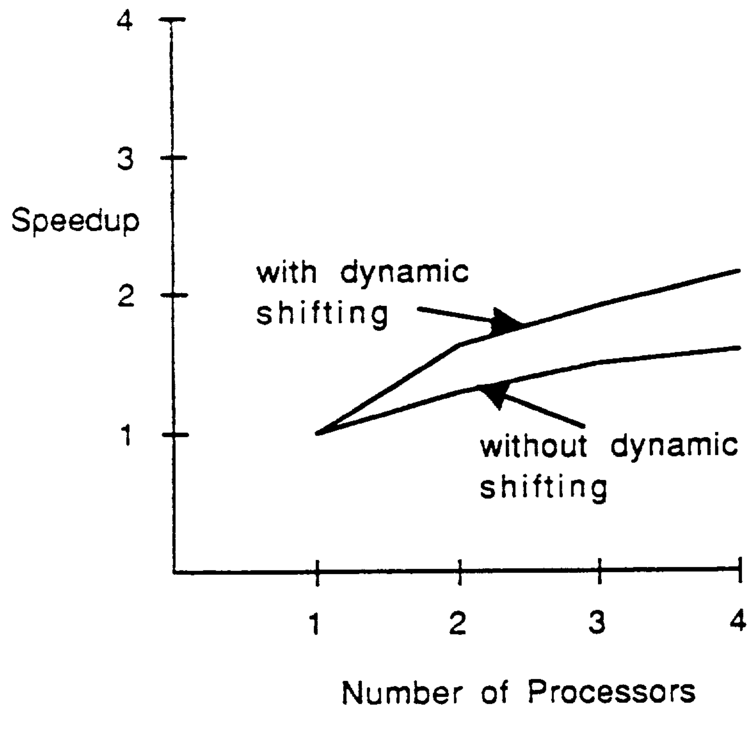

Figure 11: Speedup curves for LANZ with and without dynamic shifting

t of the lowest eigenvalues are required, or 2) all the eigenvalues in a particular range are required.

If all the eigenvalues in a range are required, then the selection of shifts is straightforward. A shift on each endpoint of the range is always required to compute the number of eigenvalues in the range. Additional shifts can then be selected in the interior of the range, if more than two groups of processors are available. Because the eigenvalues closest to a shift are found first, the groups working on the endpoints will calculate some eigenpairs outside the specified range. Initially, two assumptions will be made: 1) the eigenvalues are linearly distributed and, 2) each group will compute an equal number of eigenvalues around its shift. Given these assumptions, the initial shifts, μ_k, will be chosen such that each group will be assigned an equal amount of work. Given the endpoints of an interval, a and b, and the number of groups, m, the equations for computing the m μ_k's are (if $m > 1$):

$$\begin{aligned} \mu_1 &= a \\ \mu_k &= a + (k-1)\frac{b-a}{m-1} \quad for\ 1 < k < m \\ \mu_m &= b. \end{aligned} \tag{9}$$

If $m = 1$, then the shifts can be chosen in one of two ways: 1) two shifts will be chosen, a and b, or 2) three shifts will be chosen, a, b and the midpoint. The second method will result in fewer Lanczos steps than the first method, but will require an extra factorization. Because the cost of factorization and the cost of a Lanczos step depends on the problem, the selection of a shift selection method is problem dependent.

In this section, the focus is on the range case, however, a brief description of the algorithm for selecting shifts when the lowest t eigenvalues are sought is now given. Because only a starting point, 0, for the search is available, a group of processors will start with a shift of 0. As estimated eigenvalues are generated by the first group, more shifts can be selected based on the estimated eigenvalues. The shifts would be generated as soon as enough estimated eigenvalues are available to give a good indication of where shifts will be useful. Other groups of processors will begin work on these newly generated shifts. A tradeoff exists between putting the idle groups of processors to work as early as possible and making sure that the newly generated shifts are in areas of the eigenvalue spectrum where useful work is done. In addition, it would be desirable to have as many processors as possible work on the first factorization and then let a subset of these processors work on the computation of the Lanczos steps at this first shift. This algorithm is not as efficient as the algorithm for the range case, because not all the processors can begin working at the same time. For the group approach to be effective, a fairly large number of eigenvalues must be sought, otherwise, one group could quickly find the desired eigenvalues.[8].

[8]In this case, the normal parallel LANZ program is efficient because few Lanczos steps

Each group computes eigenpairs until:

a) it is finished with the sub-interval between its shift and the shift immediately to its right and it is finished with the sub-interval between its shift and the shift immediately to its left, or

b) the number of allowable steps is exceeded

c) the storage capacity of the group is exceeded

A group is finished with a sub-interval if:

d) all the eigenpairs in that sub-interval have been found, or

e) the group finds a converged eigenpair that is on the other side of an eigenpair found by another *currently working* group.

Figure 12: Algorithm for group approach

The algorithm for a group is given in Figure 12. Condition e allows the groups to negotiate the portion of the sub-interval that each will compute. This has a significant advantage over assigning half of the sub-interval to each group when the eigenvalues in the sub-interval are concentrated near one of the shifts. If each group is assigned half of the sub-interval, then the groups will take a very unequal number of steps, resulting in poor load balancing.

The data that need to be communicated between the groups are: 1) the shifts, 2) the inertia count at each shift, 3) the eigenpairs, and 4) the left and right endpoints of the range of eigenvalues computed by each group. This data is conceptually communicated via message-passing, however, shared variables protected by critical sections are used in the Force implementation.

To make good choices for the size and number of groups, as well as to predict the performance of the group approach, the model described in Section 3 will now be extended. The model for predicting the performance of the group approach is

$$T(p) = \max_{i=1,m}(t_{g_i}(1, p_i, n_{s_i}, n_{e_i})), \tag{10}$$

where the parameters are specified in Figure 13. Of course, the optimal number of groups, m, and processors per groups, p_i, given p cannot be known *a priori* for most problems. However, the model can be used to make general statements regarding the selection of m and p_i. From the model in Section 3 and the analyses in Section 4, it can be concluded that the parallel efficiency of a group, g_i, will deteriorate as p_i and n_{s_i}[9] increase. However, although the efficiency

are needed.

[9] In general, as n_{e_i} increases, n_{s_i} increases.

Parameter	Description
$t_{g_i}(a,b,c,d)$	for group i, the T_p from equation 1 where, a is the number of factorizations, b is the number of processors, c is the number of Lanczos steps, d is the number of eigenvalues found,
m	the number of groups
p_i	the number of processors in group i
n_{s_i}	the number of Lanczos steps taken by group i
n_{e_i}	the number of eigenpairs calculated by group i

Figure 13: Parameters for group model

decreases as p_i increases, the execution time does decrease.[10] In general, as the the number of groups, m, increases, the number of Lanczos steps per group, n_{s_i}, decreases.[11] The tradeoff, with fixed p, between the number of groups and the number of processors per group can be essentially characterized as trading off decreased factorization time for more steps per group, e.g. as p_i increases, and m decreases, the factorization time per group decreases, however, because the group is now responsible for computing more eigenvalues, more Lanczos steps are required. To illustrate this point more effectively, the time for factorization and the number of Lanczos steps have been plotted against the number of processors in group i. In the graph shown in Figure 14[12], p is assumed to be held constant, and, therefore, m is implicitly decreasing as p_i increases.

The group approach as described in this section has been implemented on the Cray Y-MP using the Force parallel language. Due to constraints imposed by the Force, however, each group consists of only one processor. Although this constraint does not allow full use of the group approach, results from this implementation, combined with the model and results from the parallel implementation in Section 2, allow very accurate assessments of the performance of the group approach. To make these estimates, n_{s_i} and n_{e_i} for group i in the group implementation were used to calculate the time for group i given different values of p_i.

First, performance results from the implementation on the Cray Y-MP will be given. The group approach using $1 \cdots 4$ processors was run for three separate situations on the same problem[13]: 1) all the eigenpairs in a range that contained

[10]The decrease continues until a plateau is reached. After this plateau is attained, the addition of more processors will only increase execution time.

[11]However, this is very dependent on the distribution of the eigenvalues. Also, in general $\sum_{i=1}^{m} n_{s_i}$ is not constant for different m in the same problem.

[12]No units for the y-axis are shown because the interest is only in the direction of the curves.

[13]The order of the eigenproblem was 1824 and the average semi-bandwidth was 127. The

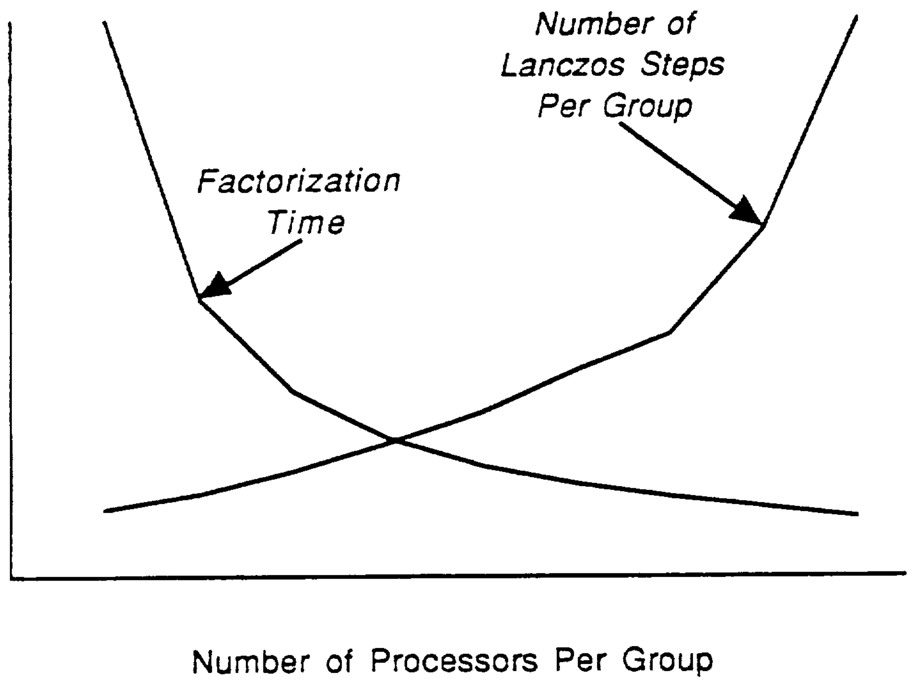

Figure 14: The tradeoff between factorization time and the number of Lanczos steps

20 eigenpairs were sought, 2) all the eigenpairs in a range that contained 40 eigenpairs were sought, and 3) all the eigenpairs in a range that contained 60 eigenpairs were sought. In cases 1 and 2, good speedup was achieved. However, a dropoff in speedup was seen in case 3 when m was increased from three to four because the majority of the eigenvalues were concentrated in the far left end of the range. This concentration of eigenvalues resulted in very poor load balancing. Speedup curves for all three cases are plotted in Figure 15.[14]

To give the reader a better understanding of how the group algorithm operates, a short discussion of the division of work among the groups for a specific example is now given. First, two aspects of the algorithm that should be noted are: 1) because eigenvalues closest to a shift are found first, the groups working on the outermost shifts will likely find eigenvalues outside the desired range, and 2) two groups may compute the same eigenpair. The division of the eigenvalue spectrum among the processors for case one[15] is shown in Figure 16. An examination of the division of work using two groups in the figure reveals that groups one and two computed several eigenvalues outside the desired range. Also, group one computed one eigenvalue in the range of group two and group two computed one eigenvalue in the range of group one, therefore, although group one found 21 eigenvalues, only 20 eigenvalue are seen in its range.

To assess the performance of the group algorithm with different values of p_i and m, results from the implementation in this section are combined with results from the implementation in the previous section. The runtime of the group algorithm is the runtime of the group that takes the most Lanczos steps. If all the groups are of equal size, then the number of Lanczos steps that each group takes is independent of p_i. Let

$$n_m = \max_{i=1\cdots m} n_{s_i}. \tag{11}$$

The implementation described in this section was run on a problem[16] and n_m was observed for $m = 1\cdots 4$. Then, the implementation described in the previous section was run on the same problem for n_m, $m = 1\cdots 4$, steps on p_i, $i = 1\cdots 4$, processors. The execution time observed for each of these runs would be the execution time of the group algorithm. The speedup curves for $m = 1\cdots 4$ groups are shown in Figure 17. From this graph, it can be concluded that, in general, the group algorithm is superior to the algorithm described in the previous section. It can also be concluded that, in general, the best performance of the group algorithm will not be attained with $m = p$.

size of the problem was restricted due to the limited stack size of the Cray Y-MP.

[14] If m=1, then the group approach is equivalent to the parallel implementation described in Section 2.

[15] The three cases are described in the previous paragraph.

[16] A vibration problem with n=12054 and an average bandwidth of 328. All the eigenpairs in a specified range containing twenty eigenpairs were sought.

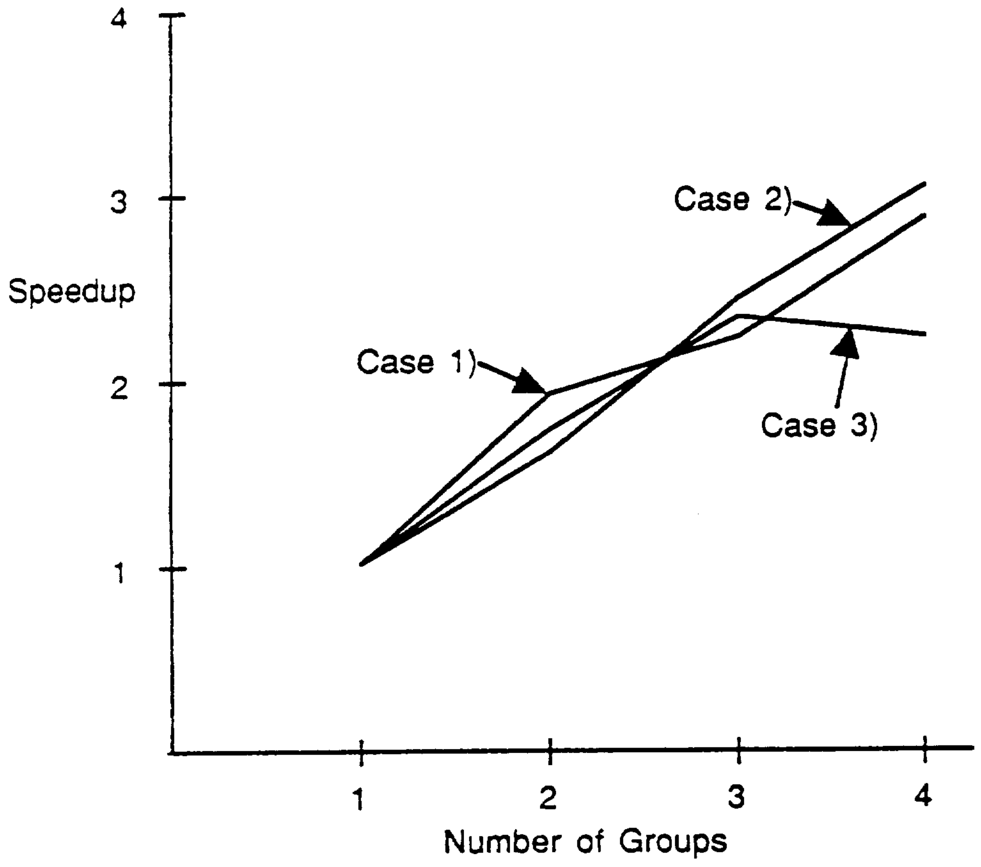

Figure 15: The performance of the group implementation

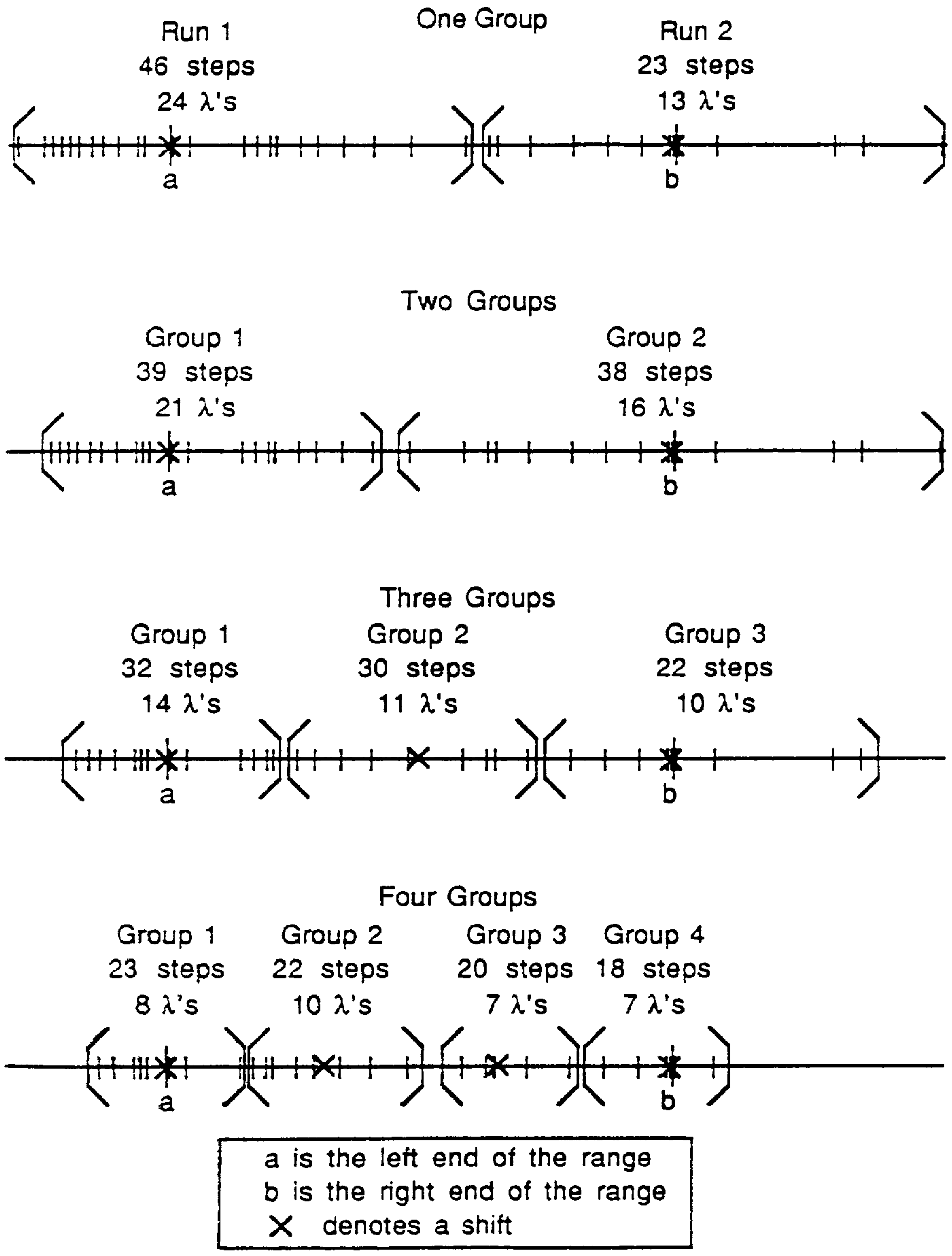

Figure 16: The division of work for case one

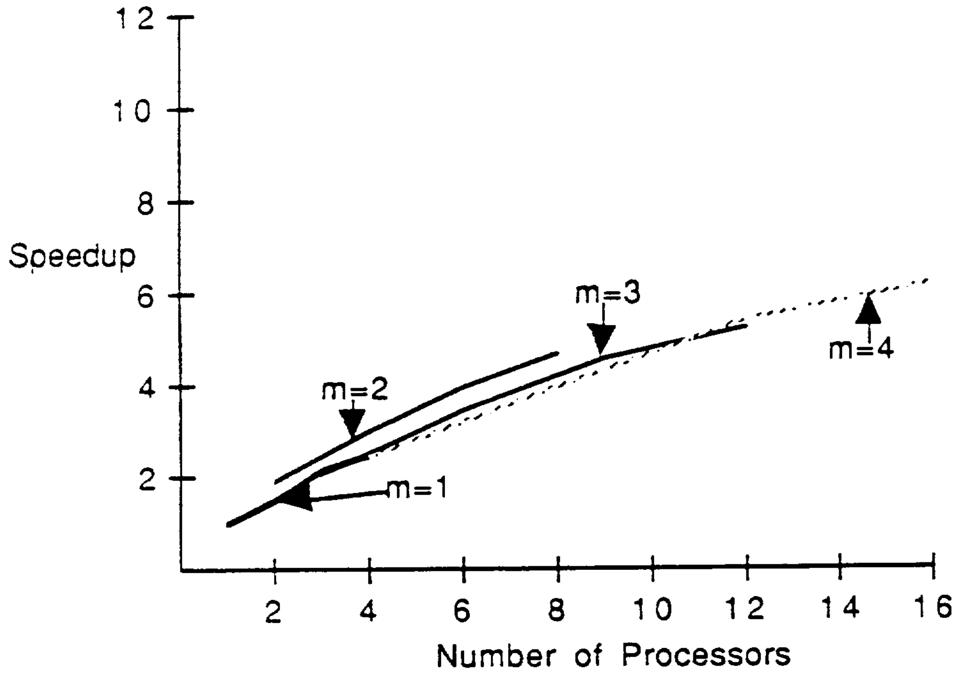

Figure 17: The estimated performance of the group approach

6 Concluding Remarks

The use of the Force parallel programming language on a large, practical code revealed some shortcomings in the language and several remedies were proposed. The parallelization of a Lanczos-based solver for the generalized eigenproblem was discussed. An analytical performance model for this code was used to explain the tradeoffs that were made when implementing the eigensolver on a parallel-vector computer. The analytical model also was used to show that a bottleneck in the parallel eigensolver is the forward and back substitution algorithms. Two algorithms for improving the parallel performance were given.

The parallel eigensolver presented in this report is suitable for parallel computers with a moderate number of processors as well as for parallel-vector computers with a small number of processors. However, due to the bottleneck imposed by the forward and back substitution algorithms, this eigensolver is not suitable for large scale parallelism. To exploit large-scale parallelism, the authors are seeking to eliminate this bottleneck by pursuing research into iterative methods for solving $(K - \sigma M)x = y$.

References

[1] M. T. Jones and M. L. Patrick, "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.

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

[3] R. G. Grimes, J. G. Lewis, and H. D. Simon, "The Implementation of a Block Shifted and Inverted Lanczos Algorithm for Eigenvalue Problems in Structural Engineering," ETA-TR-39, Boeing Computer Servies, Seattle, WA, August, 1986.

[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] M. T. Jones and M. L. Patrick, "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.

[6] B. N. Parlett and D. S. Scott, "Lanczos Algorithm With Selective Orthogonalization," *Mathematics of Computation*, vol. 33, pp. 217–238, 1979.

[7] H. Jordan, "The Force," Computer Systems Design Group, University of Colorado, 1987.

[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] E. Poole, "Personal communication," 1990.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-182072 ICASE Report No. 90-48	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle THE USE OF LANCZO'S METHOD TO SOLVE THE LARGE GENERALIZED SYMMETRIC EIGENVALUE PROBLEM IN PARALLEL		5. Report Date July 1990
		6. Performing Organization Code
7. Author(s) Mark T. Jones Merrell L. Patrick		8. Performing Organization Report No. 90-48
		10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Compuer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-18605
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code
15. Supplementary Notes Langley Technical Monitor: Richard W. Barnwell Final Report		

16. Abstract

The generalized eigenvalue problem, $K\chi = \lambda M\chi$, is of significant practical importance, especially in structural engineering where it arises as the vibration and buckling problems. New software, **LANZ**, based on Lanczo's method has been developed for solving these problems and uns on SUN 3, SUN 4, Convex C-220, Cray 2, and Cray Y-MP systems.

Preliminary results of using the Force to obtain a multiprocessor implementation of **LANZ** on MIMD parallel/vector systems are reproted here. A parallel execution time model of **LANZ** is defined and used to predict the performance of **LANZ** as well as examine hypothetical modifications to **LANZ**. The results of using dynamic shifting to improve parallelism are presented. Finally, the results of assigning a group of processors to separate shifts and finding all the desired eigenvalues using **LANZ** in parallel are reported.

17. Key Words (Suggested by Author(s)) generalized eigenvalue problem, Lanczo's method, parallel		18. Distribution Statement 61 - Computer Programming and Software 64 - Numerical Analysis Unclassified - Unlimited	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 28	22. Price A03

NASA FORM 1626 OCT 86