Distributing Matrix Eigenvalue Calculations over Transputer Arrays

Tim Hopkins and Barry Vowden
Computing Laboratory and Mathematical Institute
University of Kent
Canterbury, CT2 7NF
Kent, UK.

March 29, 2001

Abstract

We discuss the parallel numerical solution of the matrix eigenvalue problem for real symmetric tridiagonal matrices. Instances occur frequently in practice. Two implementations of the Sturm sequence algorithm on transputer arrays are described. For the first the maximum size of matrices which may be accommodated is restricted by the amount of local memory available. The second implementation removes this constraint but requires an increased execution time.

1 Introduction

Matrices play a key role in the analysis of mathematical models built to study systems where there are a large number of variables, and where these variables are linearly related. Frequently the matrix arising from a system model is used to represent the action or transformation process implicit in the system. When this is so, the action may be resolved into a number of simple, primitive components, in terms of which the complex behaviour of the full system may be explicitly described. The matrix eigenvalues specify the strengths or force of these primitive constituents, and the associated eigenvectors define the directions in which they act and so how they combine to determine the system as a whole. In almost every application of matrices the eigenvalues and eigenvectors relate to underlying explanatory variables in the context considered.

Because of this, tools for the numerical determination of matrix eigenvalues and eigenvectors are of considerable practical importance. Problems are of varying kinds; in some cases it may be necessary to compute all the eigenvalues which fall within a specified interval, together with the corresponding eigenvectors; in others the first few smallest, or largest, eigenvalues and their eigenvectors are needed, whilst in the case of medium sized matrices an application may require knowledge of the complete eigensystem. No one algorithm is effective in all circumstances, but there do now exist comprehensive suites of FORTRAN programs in both the NAG [2] and EISPACK (see Smith et al. [4] and Garbow et al. [1]) software libraries, and these may be relied upon to solve accurately and efficiently a very wide range of eigensystem problems involving matrices of modest size (with matrix order not exceeding 500). Larger scale problems arise abundantly; the matrix order is frequently as much as 10^4, and sometimes exceeds 10^6.

The advent of parallel processing has contributed means for undertaking large-scale eigenvalue calculations. However most algorithms from the existing well-proven collections have been finely

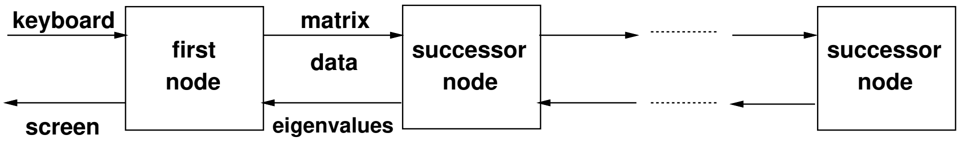

Figure 1: The pipeline organisation.

tuned to sequential processing, and need substantial adaptation before they can realise fully the gains deriving from parallel processing. In the sequel we describe the improvements in performance, both in terms of the size of problem treated and also the processing speed, which may be achieved with parallel implementations on transputer arrays of a certain important eigenvalue algorithm. In addition to the capacity to undertake larger scale calculations, it is valuable to achieve a speed up in the processing of small and medium sized matrices, because of potential real-time applications.

Our implementations are in Occam, using a Meiko Computing Surface, the TDS 700D Occam Compiler, 20Mhz T800 transputers, with 4-cycle external DRAM and 20 Mbaud link speeds.

2 The Sturm sequence method

A class of matrices important because of their frequent occurrence, and through their role as intermediaries in eigenvalue calculations, are the symmetric tridiagonal matrices. A matrix is tridiagonal if its non-zero entries lie along the main diagonal or along the first super- and sub-diagonals of the matrix, and is symmetric if corresponding entries in the super- and sub-diagonals are equal. Physical systems consisting of linearly linked individual components and where interaction between components is restricted to nearest neighbours will generate matrices which are symmetric tridiagonal. In addition several algorithms suitable for determining the eigensystems of more general matrices proceed by first building an auxiliary symmetric tridiagonal matrix whose eigenvalues either coincide with or are close to the eigenvalues of the original matrix.

A well-tried and proven sequential algorithm for computing those eigenvalues of a symmetric tridiagonal matrix which lie within a specific range, is the Sturm sequence, or bisection, algorithm.

Suppose a given symmetric tridiagonal matrix has diagonal entries a_1 to a_n and non-zero off-diagonal entries b_2 to b_n; so that the matrix is

$$\begin{bmatrix} a_1 & b_2 & & & & 0 \\ b_2 & a_2 & b_3 & & & \\ & b_3 & a_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & a_{n-1} & b_n \\ 0 & & & & b_n & a_n \end{bmatrix}$$

The Sturm sequence method requires the repeated evaluation, for emerging values of λ, of the Sturm polynomials $p_i(\lambda)$ for i between 0 and n, which are generated by means of the three term recurrence relation

$$p_i(\lambda) = (a_i - \lambda)p_{i-1}(\lambda) - b_i^2 p_{i-2}(\lambda),$$

where to begin we define $p_0(\lambda) = 1$ and $p_1(\lambda) = a_1 - \lambda$.

The key property of these Sturm polynomials is that the count of disagreements in sign between consecutive members of the sequence

$$p_0(\lambda), p_1(\lambda), p_2(\lambda), \cdots, p_n(\lambda)$$

equals the number of matrix eigenvalues which are less than or equal to λ. Should it happen that one of the numbers, say $p_i(\lambda)$, is zero then, for the purposes of the count, the sign opposite to that of $p_{i-1}(\lambda)$ is allocated to $p_i(\lambda)$. An iterated bisection process may now be used to locate any individual eigenvalue of the matrix.

When calculations are performed in floating-point arithmetic, later $p_i(\lambda)$ frequently fall beyond the range of machine supported reals. Both underflow and overflow occur. In the case of underflow, substituting zero for the numbers could result in incorrect sign determinations. To avoid this difficulty the numerical algorithm substitutes for the sequence of $p_i(\lambda)$ a sequence of $q_i(\lambda)$, defined by

$$q_i(\lambda) = p_i(\lambda)/p_{i-1}(\lambda).$$

The $q_i(\lambda)$ satisfy $q_1(\lambda) = a_1 - \lambda$ and the recurrence

$$q_i(\lambda) = (a_i - \lambda) - b_i^2/q_{i-1}(\lambda).$$

The number of negative $q_i(\lambda)$ replaces the count of sign disagreements in the sequence of $p_i(\lambda)$.

If a group of eigenvalues are sought and these are calculated in order of increasing value, information generated whilst finding one eigenvalue may be used to expedite the bisection process for determining the eigenvalues which follow.

The Sturm sequence algorithm has been widely used for the numerical isolation of matrix eigenvalues since its introduction by Givens in 1954. A more detailed account of the underlying theory may be found in Wilkinson [5], and Wilkinson and Reinsch [6].

3 A distributed implementation of the Sturm sequence algorithm

The implementation exploits the concurrency available when an OCCAM program is distributed over a transputer array, by sharing the individual eigenvalue calculations amongst the available processors. The program is organised as a pipeline, which is illustrated in Figure 1. To begin the matrix data is sent along the pipe, each processor taking a local copy and passing the data on. The calculations undertaken by the various processors then advance in parallel. Finally each processor reports back, first with the eigenvalues it has determined, and then with those which it collects from its successors along the pipe.

The first node of the pipe, prompted from the keyboard, sets up and distributes the matrix data to the other processors. It does a share of the eigenvalue calculations and finally collects and reports results to the screen. The remainder of the pipe is built from several duplicate successor nodes. Equal numbers, so far as is possible, of eigenvalues are assigned to individual nodes. Figures 2 and 3 illustrate the organisation of the first node and successor node processes.

In outline the node computation process takes the form

```
...  set up node copy of matrix
...  initialise search interval
...  allocate eigenvalues to be calculated by this node
...  find eigenvalues
...  report results
```

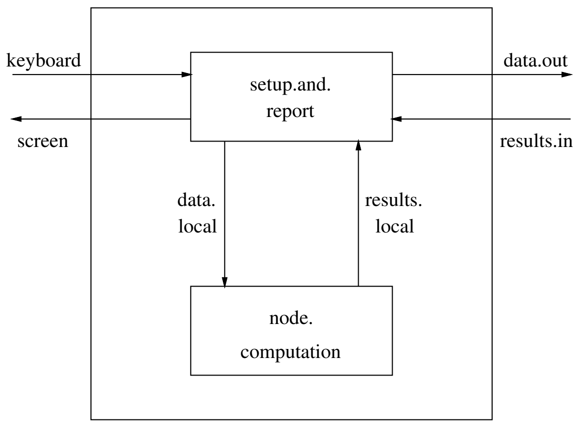

Figure 2: The first node process.

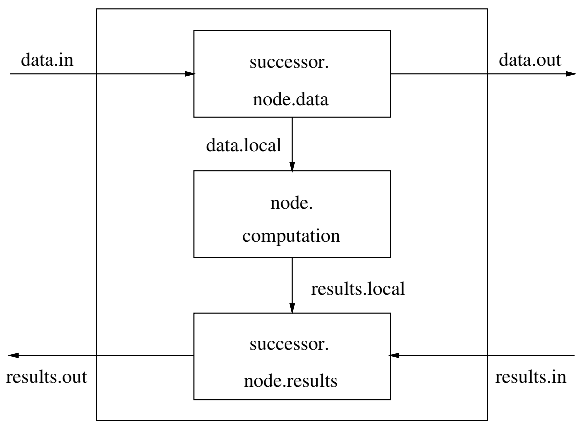

Figure 3: The successor node process.

The code implementing the Sturm sequence and bisection algorithm is included in the CRfind eigenvalues process, which undertakes the task of computing the group of eigenvalues assigned to the current node. It incorporates the technique, referred to in 2, for using information generated during one eigenvalue determination to narrow the initial search intervals for the eigenvalues following later in the group.

4 Performance under test

The following two families of symmetric tridiagonal matrices were used in tests to assess the performance of our distributed implementations of the Sturm sequence algorithm. In both cases explicit analytic formulae are known for the corresponding eigenvalues, thus enabling a ready check on the accuracy of the numerical results delivered by the algorithm.

Test matrix 1 is defined by

$$a_1 = a_2 = \cdots = a_n = 2$$

$$b_2 = \cdots = b_n = 1;$$

its eigenvalues are known to be

$$\lambda_k = 2\left(1 - \cos\frac{k\pi}{n+1}\right).$$

The eigenvalues all lie within the interval 0 to 4. The distribution is not even; as well as a spread through the interval, sets of values are clustered at each of the endpoints.

Corresponding formulae defining test matrix 2 and the associated eigenvalues are

$$a_i = (2i-1)(n-1) - 2(i-1)^2$$

$$b_i = (i-1)(n-i+1),$$

and

$$\lambda_k = k(k-1).$$

Here there is a large spread in the eigenvalues, with a wide separation at the top of the range.

In our implementation of the algorithm described in the previous section, as a result of the loop required to generate a single Sturm sequence, the time to compute a fixed proportion of all the eigenvalues may be expected to vary as n^2, with increasing n. Table 1 shows, in the case of matrix 1, a sample of measured run times corresponding to a range of matrix orders n, and for distribution over a varying number p of processors.

The matrix orders were chosen to advance in geometric progression so that the logarithmic plot of run times against n, exhibited in Figure 4, could be used to assess the n^2 dependence. The graphical analysis confirms that as soon as the order n of the matrix exceeds about 200, the logarithm of the run time is a linear function of $\log n$, with gradient equal to 2.

To optimise the gain in performance available when an algorithm is distributed over a number of processors it is important to ensure that the overall task is decomposed into component subtasks which when assigned to the various processors keeps each of these as active as possible during the lifetime of the complete process. A measure of the extent to which this is achieved is the *efficiency:* specifically the efficiency E_p of distribution for p processors is defined as

$$E_p = \frac{\textit{time taken by algorithm on one processor}}{p \times (\textit{time taken by algorithm on} p \textit{processors})}.$$

matrix order	Average time (in millisecs) to calculate all eigenvalues when the algorithm is configured on p transputers.					
	$p=1$	$p=4$	$p=8$	$p=12$	$p=16$	$p=20$
25	210	59	38	38	38	46
40	515	137	80	63	58	68
63	1235	328	176	127	115	105
100	3021	784	419	294	229	209
158	7359	1920	981	662	530	449
251	18149	4688	2387	1616	1256	1040
398	44692	11518	5830	3895	2988	2434
631	109949	28304	14237	9614	7268	5903
1000	270402	69452	34896	23285	17726	14182

Figure 4: Run times for the Sturm sequence algorithm and matrix 1. (Eigenvalues computed to 10dp.)

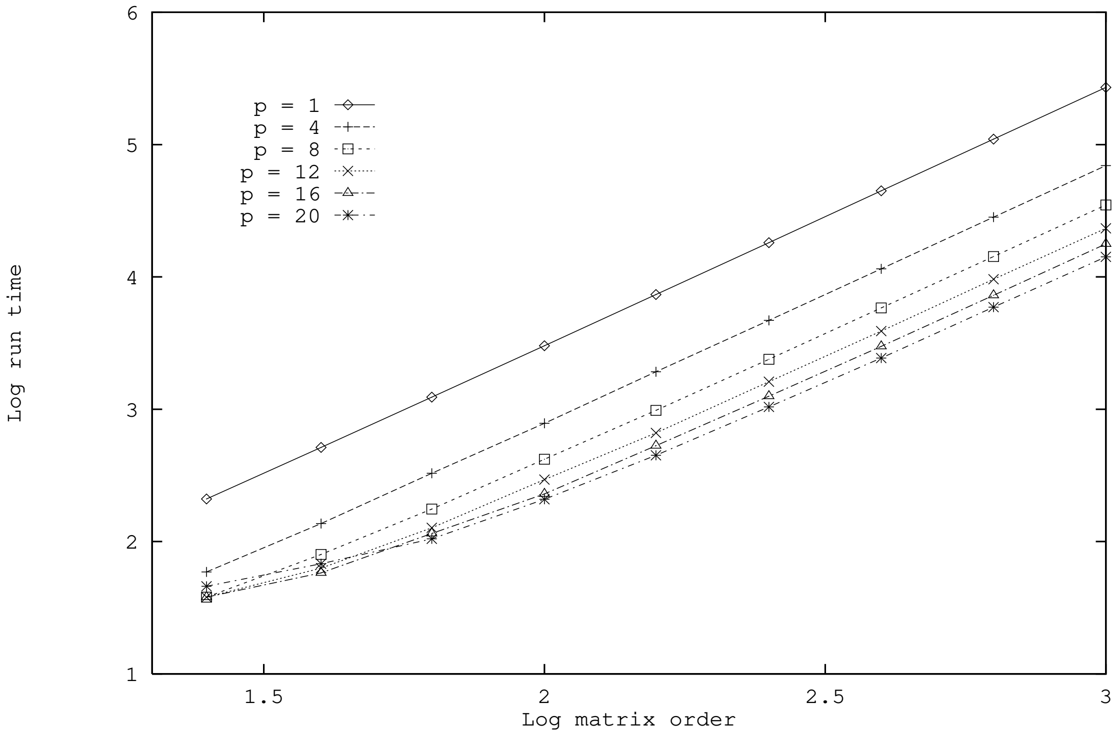

Figure 5: Logarithmic plots of run times versus matrix order.

matrix order	Average time (in milliseconds) to calculate eigenvalues when the algorithm is configured on p transputers.					
	$p = 2$	$p = 4$	$p = 8$	$p = 12$	$p = 16$	$p = 20$
398†	106360	78830	40460	26850	20460	16490
1000†	644440	474740	243020	162410	123200	97690
4000‡	444300	320260	168780	113470	87040	70290

† times for all eigenvalues ‡ times for the first 200 eigenvalues

Figure 6: Run times for the mobile data version of the Sturm sequence algorithm and matrix 1. (Eigenvalues calculated to 10 decimal places.)

To assess the efficiency of distribution for our implementation of the Sturm sequence algorithm, values of E_p were calculated by averaging the run times associated with matrix orders in the range 316 to 1000. (It is inappropriate to use data corresponding to values of n not falling in the linear portion of the log plots of run time against n : extraneous factors predominate in determining run time performances for low values of n, and invalidate comparisons for varying p.) The results we found are as follows

p	4	8	12	16	20
E_p	0.973	0.966	0.961	0.947	0.940

and they record virtually no degradation in efficiency as the number of processors increases. The small decline may be explained on the basis that equal numbers of eigenvalues are assigned to the various processors. As remarked in 2, the algorithm incorporates a technique for expediting the calculation of succeeding eigenvalues by resorting to information about them which arises spontaneously during the calculation of those eigenvalues emerging fist. This is most effective when eigenvalues are clustered, so that in the case of matrix 1 the processors near the end of the pipeline can be expected to complete their work before those situated in the middle. The effect is more pronounced when less accuracy is required and calculations are carried through retaining fewer significant figures.

The results arising from tests run on matrix 2 were broadly similar : run times were 30% longer, accounted for by the much larger spread in the eigenvalues, and the efficiencies found are as follows

p	4	8	12	16	20
E_p	0.965	0.952	0.941	0.939	0.938

5 A mobile data implementation of the Sturm sequence algorithm

The implementation of the Sturm sequence algorithm previously introduced, works by first delivering a copy of the matrix data to the various processors in the pipeline. Each processor then busies itself computing a share of the eigenvalues and upon completion reports its findings. The T800 transputer contains 4K bytes of on-chip RAM. Because of this constraint, when large matrices are processed matrix data will be stored in external memory. There will be an overhead for retrieving this data which will limit the run time performance of the implementation. It is also preferable not to make excessive demands on external memory by storing multiple copies (one for each processor) of the matrix data.

We describe an alternative implementation which reduces the amount of data the individual processors need to store, and thereby opens the way to calculations on larger matrices. This involves sending the matrix entries in a steady stream along the pipeline, each processor absorbing, putting to use and discarding the data as it passes. The first node organises the availability of data for broadcasting along the pipeline. In the prototype version matrix data was fabricated within this node (for matrix 1 and matrix 2), but it is envisaged that a general purpose implementation might entail regular fast transfers of suitably sized blocks of matrix data from some backing store into the local memory accessible to the first node. To start the processing, the first node communicates to the successor nodes the matrix order and the index range for the eigenvalues to be calculated. The first node then transmits in turn pairs of matrix entries (a_1, b_1), (a_2, b_2), (a_3, b_3), ... after the last pair (a_n, b_n) is sent out the transmission recommences with (a_1, b_1), (a_2, b_2), ... and so on. Each node uses the data stream of (a_i, b_i) to calculate first the initial search interval and then Sturm sequences. For example a node calculates q_i from q_{i-1} by means of the formula (according to 2)

$$q_i = a_i - \lambda - b_i^2/q_{i-1}$$

when it receives the pair (a_i, b_i); q_{i+1} is in turn calculated from q_i with the arrival next of (a_{i+1}, b_{i+1}). The negative count accumulation and the bisection process work exactly as in the previous implementation and, when found, eigenvalues are reported back just as before.

The run time performance for this second implementation of the Sturm sequence algorithm is summarised in Table 2. Comparison with the times displayed in Table 1 shows that the new version performs unsuccessfully when matched against our first, described in 3; run times are as much as seven times longer, and tests using matrix 2 confirm this. But the design is primitive, and should be considered as no more than a prototype. Communications, which are known to be relatively time consuming, and so to require careful management, are extensively employed here. Improvements in performance can be expected to result from reckoning the cost of inter processor communications, assessing the savings to be achieved by transmitting packets of matrix data rather than individual items, and by buffering, so that transmissions along the pipeline are not dependent upon the readiness of particular processes to take in data. Initial findings indicate that the run times reported in Table 2 can be reduced by 50% at least when these techniques are incorporated. Data appearing in the final line of Table 2 demonstrate the viability of this implementation for undertaking larger scale matrix eigenvalue calculations.

6 Conclusion

To gauge the effectiveness of the transputer implementations we have presented for the Sturm sequence algorithm, comparisons of their run time performances were made with other algorithms and implementations. If the task is to calculate all the eigenvalues of a symmetric tridiagonal matrix, then the implicit ql algorithm (see Wilkinson and Reinsch [6]) is the most effective tool in the sequential domain. (It is, however, a feature of this algorithm that its design is strongly sequential in nature, and it does not easily adapt for use as a tool in parallel processing.) We found that an Occam version sited on a single transputer delivered run times equivalent to our first implementation when configured over 5 to 6 processors. This rises to 7 or 8 when comparison is made with the NAG fortran version on a DEC VAX 8800 computer, running a VMS operating system. Thus the Sturm sequence algorithm in our first implementation is an efficient means of finding matrix eigenvalues when the calculations are distibuted over a transputer array comprising 10 or more processors.

Ralha and Thomas [3] report a quite different transputer implementation of the Sturm sequence algorithm. A pipeline is again employed, but used instead to distibute the calculation of a single process gathers the information emerging from the pipeline in the form of negative counts for the Sturm sequences, and updates its record of the whereabouts of all the eigenvalues accordingly. Unfortunately the numerical testing reported for their implementation is somewhat limited. A strict comparison with our results is made difficult because of the small size of their test matrices, and the T414 transputers utilised; but our first implementation would appear to run between 2 and 4 times faster. However this may be an inappropriate comparison. Rather, it would seem that the two approaches should be regarded as complementary. For a given symmetric tridiagonal matrix, if all or a substantial proportion of the eigenvalues are sought, then our versions provide the more effective tool, but when a small number only of the eigenvalues are required the Ralha and Thomas implementation may be preferable.

References

[1] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines: EISPACK Guide Extensions*, volume 51 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1977.

[2] Numerical Algorithms Group Ltd., Oxford, UK. *The Mk13 NAG Fortran Library Manuals*, 1989.

[3] R. Ralha and K. S. Thomas. Solution of eigenvalue problems in occam and transputers. Technical report, Department of Electronics and Computer Science, University of Southampton, UK, 1988.

[4] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines: EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1976.

[5] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, 1965.

[6] J. H. Wilkinson and C. Reinsch. *Handbook for Automatic Computation Volume II - Linear Algebra*. Springer-Verlag, New York, 1971.