Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Computational Physics 201 (2004) 573–585
JOURNAL OF COMPUTATIONAL PHYSICS
www.elsevier.com/locate/jcp

A parallel/recursive algorithm

W.R. Gibbs *

New Mexico State University, Las Cruces, NM 88003, USA

Received 29 December 2003; received in revised form 15 June 2004; accepted 16 June 2004
Available online 23 July 2004

Abstract

An algorithm is discussed for converting a class of recursive processes to a parallel system. It is argued that this algorithm can be superior to certain methods currently found in the literature for an important subset of problems. The cases of homogeneous and non-homogeneous two term recursions are treated. The basic cost factor of the algorithm over non-parallel operations is 2 if only the final values of the sequence is needed and 4 if all elements are required. In practice, these factors can be reduced considerably. Applications to three problems (finding the eigenvalues of a tri-diagonal matrix, the solution of a radial wave equation and the solution of a tri-diagonal matrix) are discussed.

1. Introduction

The solution of some problems requires a greater number of operations for an appropriate parallel algorithm than one that would be used in a strictly serial calculation. A gain in speed can still be expected by running them on a parallel machine but there is a cost factor since one cannot expect to achieve an increase in speed equal to the number of processors when compared with the best non-parallel algorithm.

As an example, consider two term iteration

$$x_{i+1} = a_i x_i + b_i x_{i-1}, \quad i = 1, 2, \ldots, N. \tag{1}$$

One method suggested in the literature [1] is to replace the steps in the algorithm by matrix multiplication. This algorithm requires extra operations which will be discussed further at the end of Section 2.1.

Another possible algorithm (the one considered here) is based on the fact that there are only two independent solutions to Eq. (1). The proper linear combination of them to represent the actual solution can be

* Tel.: +1-505-646-6711; fax: +1-505-646-1935.
E-mail address: gibbs@nmsu.edu.

0021-9991/$ - see front matter
doi:10.1016/j.jcp.2004.06.008

determined by the starting values. In this paper, the application of such an algorithm for Eq. (1), as well as a similar one for the iteration when there is an additional term, c_i, on the right-hand side, is discussed.

2. General description of the algorithm

2.1. Homogeneous case

A recursion relation, such as Eq. (1), can be viewed as one long sequence of values which leads from a beginning pair of values to the end. A desirable procedure for a parallel system would be to cut up this sequence into separate strips (as many as there are processors) and let each processor work through its part independently. For the first processor, there is no problem since the starting values are known there. But the second processor (and the rest) will not have their starting values available (the final values in the previous processor), so this procedure does not seem possible. With a moderate expense, however, it can be done. Since there are only two independent solutions of Eq. (1) we can construct two (arbitrary but independent) solutions, which will provide basis functions, and combine them when the starting values for each processor are known from the result of the previous one. For simplicity, consider the same algorithm running on all processors ignoring the fact that it could be computed more efficiently on the first processor.

Let the total length of the recursion relation be $N+2$ (including the first two starting values, hence the 2) with M processors. Each processor will be assigned a recursion of length $L = N/M$ which is supposed to be integer and large. The work to be done by each processor will be proportional to M since it never has to calculate the first two values. In order to see how such an algorithm works, let us analyze a (very modest) system of 32 recursion steps to be calculated with four processors.

Each processor will do the recursion twice, once with starting values 0 and 1 and once with values 1 and 0. That is, each processor calculates the two basis functions starting with the first two values (1,0) and (0,1). It uses the appropriate values of a_i and b_i for its position in the global sequence, of course. For the first processor, the basis functions start with

$$y_0^{10} = 1;\ y_1^{10} = 0 \quad \text{and} \quad y_0^{01} = 0;\ y_1^{01} = 1. \tag{2}$$

Since any solution of the recursive formula can be written as a linear combination of the two basis functions

$$x_i = \alpha y_i^{10} + \beta y_i^{01}, \tag{3}$$

we can see from the definition of the initial values in the first processor that

$$x_0 = \alpha; \quad x_1 = \beta, \tag{4}$$

where x_0 and x_1 are the starting values for Eq. (1). We could find all of the values of the function in the first processor by calculating

$$x_i = x_0 y_i^{10} + x_1 y_i^{01} \quad \text{First Processor} \quad i = 0, 1, \ldots, 8, 9 \tag{5}$$

but it is better not to do that immediately. Since each value is independent, we may calculate only the last two values if we wish. These would be, in this simple case, x_8 and x_9. Notice that these are the starting values for the second processor. Thus, for the second processor since it started with $y_8^{10} = 1$; $y_9^{10} = 0$ and $y_8^{01} = 0$; $y_9^{01} = 1$ choosing the proper linear combination (α and β) to give the true values of x_8 and x_9 (known from the first processor) again we could calculate all of the values

$$x_i = x_8 y_i^{10} + x_9 y_i^{01} \quad \text{Second Processor} \quad i = 8, 9, \ldots, 16, 17. \tag{6}$$

Again, we need calculate only the last two (x_{16} and x_{17}) to get the starting values for the third processor. From these, we obtain the last values in the third processor x_{24} and x_{25} and the fourth processor x_{32} and x_{33}. Thus, we have found the last two values of the sequence with the evaluation (after the parallel compu-

tations) of eight equations. For four processors, there will always be eight equations regardless of the length of iteration, L, within each processor.

Table 1 gives the operations explicitly for this small example. In this case, each processor has only eight iterations to do. In a more practical, example numbers more like 10^6 might be expected. The table lists the initial conditions at the top followed by the iterations. The generic variable, y, indicates both y^{01} and y^{10} are to be calculated.

After this work has been done the following sequential steps need to be taken using only the last two values taken from each processor.

$$
\begin{aligned}
&x_8 = x_0 y_8^{10} + x_1 y_8^{01}, \quad x_9 = x_0 y_9^{10} + x_1 y_9^{01},\\
&x_{16} = x_8 y_{16}^{10} + x_9 y_{16}^{01}, \quad x_{17} = x_8 y_{17}^{10} + x_9 y_{17}^{01},\\
&x_{24} = x_{16} y_{24}^{10} + x_{17} y_{24}^{01}, \quad x_{25} = x_{16} y_{25}^{10} + x_{17} y_{25}^{01},\\
&x_{32} = x_{24} y_{32}^{10} + x_{25} y_{32}^{01}, \quad x_{33} = x_{24} y_{33}^{10} + x_{25} y_{33}^{01}.
\end{aligned}
$$

The absolute indices have been used above on the basis functions. It is often more convenient to use a combination of the local index and the processor number. The local index will be denoted by λ and runs from 0 to $L+1$.

In general, only two values in any processor need be computed to find the starting values and hence the coefficients of the two basis functions for the next processor. So the serial overhead is only twice 3 floating point operations per processor. Even if all of the values of the sequence are needed, it is better to do this operation first, because the intermediate values can then be found in parallel using the starting values obtained in this way.

Repeating the above argument for the general case with M processors labeled $\mu = 0, 1, 2, \ldots, M-1$ and $N+2$ total values of the indices of x_i, the resulting sequences are calculated in a (long) parallel calculation,

$$
{}^{\mu}y_{\lambda}^{10} \quad \text{and} \quad {}^{\mu}y_{\lambda}^{01}; \quad \lambda = 2, 3, \ldots, L+1; \quad [\mu = 0, 1, \ldots, M-1]. \tag{7}
$$

The square brackets indicate that the calculations for the different values of μ are done in parallel. After this step, the equations

$$
\left\{ x_{(\mu+1)L} = x_{\mu L}\, {}^{\mu}y_{L}^{10} + x_{\mu L+1}\, {}^{\mu}y_{L}^{01}, x_{(\mu+1)L+1} = x_{\mu L}\, {}^{\mu}y_{L+1}^{10} + x_{\mu L+1}^{\mu} y_{L+1}^{01} \right\}_{\mu=0,1,\ldots,M-1} \tag{8}
$$

are evaluated in a (short) sequential calculation. The number of equations is always twice the number of processors. The values of x have been written with absolute indices but we may use a notation for individual processors. The pre-superscript μ as used above denotes results from a given processor μ so we could write equally valid representations of x as

$$
x_{\mu L+\lambda} = {}^{\mu}x_{\lambda}. \tag{9}
$$

Table 1
Example of the algorithm for a small number of processors

Processor 0	Processor 1	Processor 2	Processor 3
$y_0^{01} = 0; \quad y_1^{01} = 1$ $y_0^{10} = 1; \quad y_1^{10} = 0$	$y_8^{01} = 0; \quad y_9^{01} = 1$ $y_8^{10} = 1; \quad y_9^{10} = 0$	$y_{16}^{01} = 0; \quad y_{17}^{01} = 1$ $y_{16}^{10} = 1; \quad y_{17}^{10} = 0$	$y_{24}^{01} = 0; \quad y_{25}^{01} = 1$ $y_{24}^{10} = 1; \quad y_{25}^{10} = 0$
$y_2 = a_1 y_1 + b_1 y_0$ $y_3 = a_2 y_2 + b_2 y_1$ $\vdots$	$y_{10} = a_9 y_9 + b_9 y_8$ $y_{11} = a_{10} y_{10} + b_{10} y_9$ $\vdots$	$y_{18} = a_{17} y_{17} + b_{17} y_{16}$ $y_{19} = a_{18} y_{18} + b_{18} y_{17}$ $\vdots$	$y_{26} = a_{25} y_{25} + b_{25} y_{24}$ $y_{27} = a_{26} y_{26} + b_{26} y_{25}$ $\vdots$
$y_7 = a_6 y_6 + b_6 y_5$ $y_8 = a_7 y_7 + b_7 y_6$ $y_9 = a_8 y_8 + b_8 y_7$	$y_{15} = a_{14} y_{14} + b_{14} y_{13}$ $y_{16} = a_{15} y_{15} + b_{15} y_{14}$ $y_{17} = a_{16} y_{16} + b_{16} y_{15}$	$y_{23} = a_{22} y_{22} + b_{22} y_{21}$ $y_{24} = a_{23} y_{23} + b_{23} y_{22}$ $y_{25} = a_{24} y_{24} + b_{24} y_{23}$	$y_{31} = a_{30} y_{30} + b_{30} y_{29}$ $y_{32} = a_{31} y_{31} + b_{31} y_{30}$ $y_{33} = a_{32} y_{32} + b_{32} y_{31}$

In matrix notation, we may write Eqs. (8) as

$$\begin{pmatrix} {}^{\mu+1}x_0 \\ {}^{\mu+1}x_1 \end{pmatrix} = \begin{pmatrix} {}^{\mu}y_L^{10} & {}^{\mu}y_L^{01} \\ {}^{\mu}y_{L+1}^{10} & {}^{\mu}y_{L+1}^{01} \end{pmatrix} \begin{pmatrix} {}^{\mu}x_0 \\ {}^{\mu}x_1 \end{pmatrix}_{\mu=0,1,\ldots,M-1}. \tag{10}$$

Table 2 shows the ratios of completion times to the purely sequential case to be expected with various recursion lengths and number of processors. The increase in speed over scalar is $M/2$ for large N. One sees that for moderate numbers of iterations, the scaling efficiency starts to fall of for a number of processors beyond 64. Thus, for most practical problems, the algorithm is expected to work best for relatively large numbers of iterations and a modest number of processors. This loss comes, of course, because of the need to compute the matching relations which requires a time proportional to the number of processors. The algorithm could be modified to spread this matching procedure over a number of processors but this extension is beyond the scope of the present work.

If only the end value of the sequence is needed one can stop at this point (the "short form" of the algorithm). This factor of two cost is not the best that can be obtained if the values of a_i and b_i are being calculated along with the iteration. The same values of these coefficients are used in each iteration and, if the time for the calculation of the coefficients is significant, the overhead to calculate two iterations rather than one (as would happen if the calculation were not in parallel) may be small.

The latency part of the communication time is proportional to $L = M/N$, so, for a fixed, moderate, number of processors and large N it may be made very small. There are four words per processor to be sent in order to make the connection between segments.

At this point (if needed) one can proceed to calculate the entire sequence of values in a second parallel calculation (the "long form" of the algorithm). These will be given by

$${}^{\mu}x_\lambda = x_{\mu L+\lambda} = x_{\mu L}{}^{\mu}y_\lambda^{10} + x_{\mu L+1}{}^{\mu}y_\lambda^{01}; \quad \lambda = 2,3,\ldots,L+1; \ \ [\mu = 0,1,\ldots,M-1]. \tag{11}$$

These evaluations come at the cost of an additional $3L$ floating point operations per processor, roughly a cost factor of 4, i.e., the speed increase is $M/4$ compared to the pure sequential algorithm. This cost can be reduced greatly in certain cases as we shall see later. It may be useful to leave the strip functions (or even the strip basis functions) in the processor where they were calculated.

Table 2
Strip iteration algorithm

n	$N=2^n$	Sequential	$m=2$	$m=3$	$m=4$	$m=5$	$m=6$	$m=7$	$m=8$
7	128	384	0.5625	0.3750	0.3750	0.5625	1.0312	1.0000	1.0000
8	256	768	0.5312	0.3125	0.2500	0.3125	0.5312	1.0156	1.0000
9	512	1536	0.5156	0.2812	0.1875	0.1875	0.2812	0.5156	1.0078
10	1024	3072	0.5078	0.2656	0.1562	0.1250	0.1562	0.2656	0.5078
11	2048	6144	0.5039	0.2578	0.1406	0.0938	0.0938	0.1406	0.2578
12	4096	12,288	0.5020	0.2539	0.1328	0.0781	0.0625	0.0781	0.1328
13	8192	24,576	0.5010	0.2520	0.1289	0.0703	0.0469	0.0469	0.0703
14	16,384	49,152	0.5005	0.2510	0.1270	0.0664	0.0391	0.0312	0.0391
15	32,768	98,304	0.5002	0.2505	0.1260	0.0645	0.0352	0.0234	0.0234
16	65,536	19,6608	0.5001	0.2502	0.1255	0.0635	0.0332	0.0195	0.0156
17	131,072	393,216	0.5001	0.2501	0.1252	0.0630	0.0322	0.0176	0.0117
18	262,144	786,432	0.5000	0.2501	0.1251	0.0627	0.0317	0.0166	0.0098
19	524,288	1,572,864	0.5000	0.2500	0.1251	0.0626	0.0315	0.0161	0.0088
20	1,048,576	3,145,728	0.5000	0.2500	0.1250	0.0626	0.0314	0.0159	0.0083

Column 3 gives the time to completion for straight iteration in one processor in units of the time required for one floating point operation. Columns 4–10 give the ratio of the time to completion to the corresponding time for a purely sequential realization on a single processor (column 3). The number of iterations is $N=2^n$ with a number of processors $M=2^m$. This "power of two" representation is only for simplicity and is not needed for the algorithm.

We can now compare with the matrix algorithm mentioned in Section 1. If we define

$$y_i = \begin{pmatrix} x_{i+1} \\ x_i \end{pmatrix}; \quad \alpha_i = \begin{pmatrix} a_i & b_i \\ 1 & 0 \end{pmatrix}, \tag{12}$$

then

$$y_i = \begin{pmatrix} x_{i+1} \\ x_i \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_i \\ x_{i-1} \end{pmatrix} = \alpha_i y_{i-1} \tag{13}$$

and the end member of the sequence is given by

$$\begin{pmatrix} x_{N+1} \\ x_N \end{pmatrix} = y_N = \alpha_N \alpha_{N-1} \alpha_{N-2} \cdots \alpha_1 \begin{pmatrix} x_1 \\ x_0 \end{pmatrix}. \tag{14}$$

The multiplication of matrices can be done pairwise on different processors. The first multiplication of the simple matrices can be done with two multiplications and one addition but this operation generates full two-by-two matrices so that the second step in the pairwise reduction of the multi-factor product is a complete matrix multiplication and requires eight multiplications and four additions. When compared with the two multiplications and one addition necessary to actually do the iteration, one sees that there is a cost of a factor of 4 which is paid in this case. Assuming 12 operations for all steps the maximum speed up with M processors is $M/4$. The matrix algorithm gives only the end point of the sequence (i.e., not the intermediate values) so is to be compared with $M/2$ from the algorithm just presented. With a modest number of processors, over half of the time of the matrix algorithm is spent in the first set of multiplications so that considerable savings can be achieved by considering the special case for the first operation.

This cost factor is not the only problem. The work done in each matrix multiplication is not very much (12 floating point operations). Hence, communication must take place between processors very often so that message passing time may dominate.

Another possible problem is that often the entire sequence of x_i is needed. This algorithm simply does not give it.

2.2. Inhomogeneous recursion relation

For the inhomogeneous recursion,

$$x_{i+1} = a_i x_i + b_i x_{i-1} + c_i, \tag{15}$$

three basis solutions are needed to provide a general representation. For the third basis function, we can take z_i^{00}, defined to have the starting conditions $z_{\mu L}^{00} = z_{\mu L+1}^{00} = 0$. The general form of the solution is thus

$$x_i = \alpha z_i^{10} + \beta z_i^{01} + \gamma z_i^{00}, \tag{16}$$

where z_i^{10}, z_i^{01} and z_i^{00} all separately satisfy Eq. (15). The requirement that this form satisfies the recursion relation is $\alpha + \beta + \gamma = 1$.

Using this last expression to replace γ, we can write

$$x_i = \alpha(z_i^{10} - z_i^{00}) + \beta(z_i^{01} - z_i^{00}) + z_i^{00}. \tag{17}$$

It is easy to show that

$$y_i^{10} \equiv z_i^{10} - z_i^{00} \quad \text{and} \quad y_i^{01} \equiv z_i^{01} - z_i^{00} \tag{18}$$

satisfy the homogeneous equation (i.e., with $c_i = 0$) with the same starting points as the z_i^{10} and z_i^{01}, exactly as in the homogeneous case treated before. Thus, the general form can be written as

$$x_i = \alpha y_i^{10} + \beta y_i^{01} + z_i^{00}, \tag{19}$$

where y_i^{10} and y_i^{01} can be calculated as in the previous section, i.e., without reference to c_i.

The procedure in this case is first to calculate in parallel (either along with the ${}^{\mu}y_{\lambda}^{ij}$ or separately) the (long) recursion

$${}^{\mu}z_{\lambda+1}^{00} = a_{\mu L+\lambda}\,{}^{\mu}z_{\lambda}^{00} + b_{\mu L+\lambda}\,{}^{\mu}z_{\lambda-1}^{00} + c_{\mu L+\lambda}; \quad \lambda = 1,2,3,\ldots,L; \ [\mu = 0,1,\ldots,M-1]. \tag{20}$$

Since both starting values of ${}^{\mu}z_{\lambda}^{00}$ are zero, the values of the coefficients of y_i^{10} and y_i^{01} are again just the last values from the previous processor so that the equations

$$\left\{x_{(\mu+1)L} = x_{\mu L}\,{}^{\mu}y_L^{10} + x_{\mu L+1}\,{}^{\mu}y_L^{01} + {}^{\mu}z_L^{00}, \quad x_{(\mu+1)L+1} = x_{\mu L}\,{}^{\mu}y_{L+1}^{10} + x_{\mu L+1}\,{}^{\mu}y_{L+1}^{01} + {}^{\mu}z_{L+1}^{00}\right\}_{\mu=0,1,\ldots,M-2} \tag{21}$$

or in processor notation,

$$\left\{{}^{(\mu+1)}x_0 = {}^{\mu}x_0\,{}^{\mu}y_L^{10} + {}^{\mu}x_1\,{}^{\mu}y_L^{01} + {}^{\mu}z_L^{00}, \quad {}^{(\mu+1)}x_1 = {}^{\mu}x_0\,{}^{\mu}y_{L+1}^{10} + {}^{\mu}x_1\,{}^{\mu}y_{L+1}^{01} + {}^{\mu}z_{L+1}^{00}\right\}_{\mu=0,1,\ldots,M-2} \tag{22}$$

need to be evaluated in a (short) sequential calculation.

If needed, the intermediate values in the recursive sequence can now be evaluated in parallel. These will be given by

$$x_{\mu L+\lambda} = x_{\mu L}\,{}^{\mu}y_{\lambda}^{10} + x_{\mu L+1}\,{}^{\mu}y_{\lambda}^{01} + {}^{\mu}z_{\lambda}^{00}; \quad \lambda = 2,3,\ldots,L+1; \ [\mu = 0,1,\ldots,M-1] \tag{23}$$

or

$${}^{\mu}x_{\lambda} = {}^{\mu}x_0\,{}^{\mu}y_{\lambda}^{10} + {}^{\mu}x_1\,{}^{\mu}y_{\lambda}^{01} + {}^{\mu}z_{\lambda}^{00}; \quad \lambda = 2,3,\ldots,L+1; \ [\mu = 0,1,\ldots,M-1]. \tag{24}$$

If the recursion relation is needed for a large number of functions, c_i, with the same a_i and b_i, then the basis functions y^{10} and y^{01} need be calculated only once.

3. Applications

3.1. Eigenvalues of a tri-diagonal matrix

The process of finding eigenvalues of a real tri-diagonal matrix plays a central role in the solution of the eigenvalue problem of more general real symmetric matrices. Commonly, algorithms are converted to a parallel environment either by having each processor search for an eigenvalue (method A) [2] or finding all of the eigenvalues by means of divide-and-conquer algorithms (method B) [3]. It is often useful be able to pick out only a few of the eigenvalues (the lowest ones) which is the case considered here.

For a symmetric tri-diagonal matrix,

$$A = \begin{pmatrix} a_1 & b_1 & 0 & 0 & 0 & \cdots & & & & \\ b_1 & a_2 & b_2 & 0 & 0 & \ldots & & & & \\ 0 & b_2 & a_3 & b_3 & 0 & \cdots & & & & \\ & & & & & \cdots & & & & \\ & & & & & \cdots & b_{N-2} & a_{N-1} & b_{N-1} & 0 \\ & & & & & \cdots & 0 & b_{N-1} & a_N & b_N \\ & & & & & \cdots & 0 & 0 & b_N & a_{N+1} \end{pmatrix} \tag{25}$$

there exists a well-known solution for the eigenvalues, Λ, (see, for example, [4,2]) based on Sturm sequences with bisection which allows the selection of eigenvalues. With the definition

$$x_0 = 1; \quad x_1 = a_1 - \Lambda \tag{26}$$

the recursion relation

$$x_{i+1} = (a_{i+1} - \Lambda)x_i - b_i^2 x_{i-1} \tag{27}$$

generates the determinant, $D(\Lambda)$, of $A - \Lambda I$ as the value of x_{N+1}. The eigenvalues of A can be found by locating the zeros of $D(\Lambda)$. Furthermore, the number of sign differences between successive members in the recursion sequence identifies the eigenvalues in order. For example, the lowest eigenvalue occurs at the transition from 0 to 1 sign differences in the sequence. The desired transition (and hence eigenvalue) can be found with Newton's method of bisecting some $\Lambda_{\min}$ and $\Lambda_{\max}$ at each step.

Implementing this recursion in a parallel fashion is straightforward using the algorithm given in Section 2.1. In order to calculate the number of sign differences, the individual members of the sequence need to be generated which requires the long form of the algorithm, thus seeming to cost a factor of 4 compared to a non-parallelized version. However, a hybrid method makes the cost factor closer to 2 than 4. When the difference in sign count has been reduced to unity between $\Lambda_{\min}$ and $\Lambda_{\max}$ it is known that a single eigenvalue lies in this region and that it is the correct one. From this point on, the method needs only the final value of the sequence (the determinant itself) which requires only half the time.

Hence, the algorithm can be thought of as proceeding in two phases. In the first phase, the desired eigenvalue is isolated by finding two values of the estimated eigenvalue with only a single zero of the determinant between them. After that, in the second phase, only the value of the determinant is needed and with those values one can estimate a new trial value more efficiently than a simple bisection by using

$$\Lambda = \frac{\Lambda_{\min} \mid D(\Lambda_{\max}) \mid + \Lambda_{\max} \mid D(\Lambda_{\min}) \mid}{\mid D(\Lambda_{\max}) \mid + \mid D(\Lambda_{\min}) \mid}. \tag{28}$$

One must be careful of the convergence, since it may come about with the trial eigenvalue approaching one of the limit eigenvalues without the two limits approaching each other. This improvement in efficiency is available to either the parallel or non-parallel version and tends to make the first phase dominant in time consumed.

A common method of implementing this general algorithm on parallel computers is to simply give each processor an eigenvalue to find (method A above). In this case, each eigenvalue is obtained in a purely sequential fashion but the values arrive in a parallel manner. In the present algorithm, each eigenvalue is calculated in a parallel manner and the values arrive one after the other. One improvement which is not generally available to method A is due to the availability of useful information after the first and subsequent eigenvalues are found in the first phase. It is only necessary to keep a table of the tested values of Λ vs. the corresponding number of sign differences. When the next eigenvalue is to be found, the table can be searched for the closest starting values. This table grows as the eigenvalues are found. If each processor is finding an eigenvalue starting from the outer bounds of the eigenvalue sequence this advantage is not available.

An important consideration in this algorithm (parallel or non-parallel) is the growing of the values with each step so that overflow occurs. One can solve that problem by performing a renormalization at regular intervals. In the non-parallel version, when the value of x_{i+1} is observed to exceed some predefined value then both x_{i+1} and x_i are multiplied by an appropriate constant reducing factor. This does not change the number of relative sign differences nor the sign of the last value. For the parallel version, both basis functions can be renormalized in the same way (both must be done at the same time) and the recursion is not destroyed since it is only the relative values of the basis functions which determines the number of sign differences and the sign of the final value. In the version tested here, a check for renormalization was made every 16 steps.

The algorithm was calculated for the matrix defined by

$$a_i = 0, \;\; i = 1, 2, \ldots N+1; \quad b_i^2 = i(N+1-i), \;\; i = 1, 2, \ldots N, \tag{29}$$

with N even. The eigenvalues are known to be the even integers from $-N$ to N as presented in [2,5]. In addition to the renormalization mentioned above, the entire system was renormalized such that the smallest b_i^2 was unity. Calculations were made for matrices of size up to $N=10{,}240{,}000$ finding eigenvalues to an accuracy of 1 part in 10^{11}. The method was tested on a Beowulf cluster with a 100 Mbit Ethernet (using MPICH [6,7]) and the scaling efficiency [defined as $T_1/(MT_M)$, where T_M is the time for execution on M processors] exceeded 0.96 up through 4 processors.

The "cost" of the method was calculated by comparing one-processor versions of the present algorithm with a simple sequential calculation but with the present algorithm taking advantage of the information contained in the table of the number of sign differences vs. trial Λ. For a single eigenvalue, the present method takes about 2.5 times longer than method A. For five eigenvalues, it still takes about 20% longer. For 10 eigenvalues, it is 0.85 as long, for 20 it is 0.72 as long and for 40 it requires 0.64 of the time for method A.

Procedure A also may suffer from an incommensurability with the number of processors. If one wishes 8 eigenvalues with 64 processors then only 1/8 of the capacity of the machine is being used. The calculation of the relative efficiency of the methods can be quite complicated, however. For example, if one wishes 16 eigenvalues with 8 processors then two passes will be made with method A and in the second pass tables of values accumulated in the first pass can be used. Incorporating this information could well make method A faster.

3.2. Solving wave equations

Wave equations (the Schrödinger equation is considered here) can be solved, after an expansion in Legendre polynomials, by means of one-dimensional second-order differential equations. An accurate solution can be obtained with Noumerov's method where the iteration equations for the reduced wavefunction are given by

$$\psi_{i+1}=\frac{2\psi_i-\psi_{i-1}-\frac{h^2}{12}[10w_i\psi_i+w_{i-1}\psi_{i-1}]}{1+\frac{h^2}{12}w_{i+1}}, \tag{30}$$

where

$$w(r)=k^2-\frac{2m}{\hbar^2}V(r)-\frac{\ell(\ell+1)}{r^2} \tag{31}$$

and h is the spacing in the radial variable r.

This form is converted readily into that considered in Section 2.1 with much of the work of the computation of a_i and b_i being done before the iteration. Keeping k^2 as a free parameter and precomputing one vector

$$\left\{1-h^2\left[\frac{2m}{\hbar^2}V(r)+\frac{\ell(\ell+1)}{r^2}\right]\right\}$$

the calculation of a_i and b_i requires seven floating point operations. Given that each iteration requires an additional three operations, we might expect that the parallel algorithm involves a cost of an increase from 10 to 13 operations or about 30% over the purely sequential one.

In the present case, the Schrödinger equation is solved for a bound state. The solution is started at the origin with zero for the first point and an arbitrary value for the second point and a search is made for a value of the energy (expressed here as $k^2=2mE/\hbar^2$) that causes the wavefunction at some large value of r to be zero.

The problem was treated with a near maximum precomputation of the values of a_i and b_i. Clearly, if one chooses not to compute as much as possible in advance, and hence to spend a larger amount of time in the computation of the coefficients, then the addition of a second iteration (the expense of this algo-

rithm) would make less difference. Thus, a true test of the algorithm relies on a realistic degree of precomputation.

The calculation was coded and tested for $\ell = 2$. The number of steps taken was 25.2×10^6. Because of the boundary conditions of the problem (reduced wavefunction zero at the origin) the calculation of the basis function y^{10} is not needed in the first processor. By comparing a one-processor calculation with it computed or not it was found that the cost factor of the algorithm was 1.31, in agreement with the 30% increase estimated above. The method was tested on the Beowulf cluster and no decrease in scaling efficiency was seen through 16 processors.

3.3. Tri-diagonal matrix solution

Consider the recursive solution to a tri-diagonal matrix (size $N+2$), at first without any parallelism.

$$\begin{pmatrix} a_0 & e_0 & 0 & 0 & 0 & \cdots & & & & \\ b_1 & a_1 & e_1 & 0 & 0 & \cdots & & & & \\ 0 & b_2 & a_2 & e_2 & 0 & \cdots & & & & \\ & & & & & \cdots & & & & \\ & & & & & \cdots & & & & \\ & & & & & \cdots & & & & \\ & & & & & \cdots & b_{N-1} & a_{N-1} & e_{N-1} & 0 \\ & & & & & \cdots & 0 & b_N & a_N & e_N \\ & & & & & \cdots & 0 & 0 & b_{N+1} & a_{N+1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ \cdots \\ \cdots \\ x_{N-1} \\ x_N \\ x_{N+1} \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \cdots \\ \cdots \\ \cdots \\ c_{N-1} \\ c_N \\ c_{N+1} \end{pmatrix}. \tag{32}$$

If any e_i (let the first occurrence be at $i=k$) is zero then the system can be reduced to two subsystems. To see this, observe that the first equation alone consists of one equation in two unknowns, the first two equations correspond to two equations in three unknowns, etc. If $e_k = 0$ then adding that equation to the system introduces no new unknown so the system of the first $k+1$ equations can be solved alone giving the value (among others) of x_k. In the remaining equations, the kth column can be taken to the right-hand side so that they can be solved. In this case, the system is separable. For a symmetric system, if $e_k = 0$ then $b_{k+1} = 0$ also and the two blocks are completely decoupled. Here, it is assumed that this is NOT the case so that NO $e_i = 0$. Thus, we can divide all equations by e_i or, equivalently, we can set $e_i = 1$ in the system we wish to consider. Of course, the conversion of a general system to this form entails the cost of one inverse and two multiplications per equation on the left-hand but needs to be done only once in the case of a number of different right-hand sides ($N+1$ more multiplications are necessary for each right-hand side).

For these reasons, the following system is considered:

$$\begin{pmatrix} a_0 & 1 & 0 & 0 & 0 & \cdots & & & & \\ b_1 & a_1 & 1 & 0 & 0 & \cdots & & & & \\ 0 & b_2 & a_2 & 1 & 0 & \cdots & & & & \\ & & & & & \cdots & & & & \\ & & & & & \cdots & & & & \\ & & & & & \cdots & & & & \\ & & & & & \cdots & b_{N-1} & a_{N-1} & 1 & 0 \\ & & & & & \cdots & 0 & b_N & a_N & 1 \\ & & & & & \cdots & 0 & 0 & b_{N+1} & a_{N+1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ \cdots \\ \cdots \\ x_{N-1} \\ x_N \\ x_{N+1} \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \cdots \\ \cdots \\ \cdots \\ c_{N-1} \\ c_N \\ c_{N+1} \end{pmatrix}. \tag{33}$$

Starting from the second row, the equations can be expressed as the recursion relation

$$x_{i+1} = -a_i x_i - b_i x_{i-1} + c_i; \quad i = 1, 2, 3, \ldots, N-1, N, \tag{34}$$

where neither the first nor last equations have been used.

The three basis solutions discussed in Section 2.2 (called here f_i^{10}, f_i^{01} and g_i^{00}) can be used to provide "global" basis functions (global in the sense that they represent the full recursion sequence to be distinguished from the strip functions to be discussed shortly) to express the solution. The first two basis solutions do not involve c_i and need only be calculated once for many right-hand sides. Thus, the solution separates into two parts, somewhat similar to the common factorization and back substitution methods. Once we have the basis solutions, we can apply the conditions implied by the first and last equation to determine the coefficients α and β in Eq. (19). For the first equation, we have

$$a_0 x_0 + x_1 = a_0(\alpha f_0^{10} + \beta f_0^{01} + g_0^{00}) + \alpha f_1^{10} + \beta f_1^{01} + g_1^{00} = c_0 \tag{35}$$

or

$$a_0 \alpha + \beta = c_0. \tag{36}$$

From the last equation, we have

$$a_{N+1} x_{N+1} + b_{N+1} x_N = a_{N+1}(\alpha f_{N+1}^{10} + \beta f_{n+1}^{01} + g_{N+1}^{00}) + b_{N+1}(\alpha f_N^{10} + \beta f_N^{01} + g_N^{00}) = c_{N+1} \tag{37}$$

or

$$\alpha(a_{N+1} f_{N+1}^{10} + b_{N+1} f_N^{10}) + \beta(a_{N+1} f_{N+1}^{01} + b_{N+1} f_N^{01}) = c_{N+1} - a_{N+1} g_{N+1}^{00} - b_{N+1} g_N^{00}. \tag{38}$$

From these two equations, we obtain α and β and all values of x_i can be obtained from Eq. (19).

As an alternative to Eq. (38), one can iterate one further step with Eq. (34) to obtain f_{N+2}^{10}, f_{N+2}^{01} and g_{N+2}^{00} and use the condition that $x_{N+2} = 0$ to find

$$\alpha f_{N+2}^{10} + \beta f_{N+2}^{01} = -g_{N+2}^{00}. \tag{39}$$

Returning to the parallel considerations, we can express the global basis functions in terms of the strip basis solutions in each processor, obtain the three global functions that were used in the above algorithm and then calculate the solution. However, it is much more efficient to combine the two operations.

First write the global recursion basis functions in terms of the strip basis functions

$$f_{\mu L+\lambda}^{10} = {}^{\mu}\alpha^{10\mu} y_\lambda^{10} + {}^{\mu}\beta^{10\mu} y_\lambda^{01}, \tag{40}$$

$$f_{\mu L+\lambda}^{01} = {}^{\mu}\alpha^{01\mu} y_\lambda^{10} + {}^{\mu}\beta^{01\mu} y_\lambda^{01}, \tag{41}$$

$$g_{\mu L+\lambda}^{00} = {}^{\mu}\alpha^{00\mu} y_\lambda^{10} + {}^{\mu}\beta^{00\mu} y_\lambda^{01} + {}^{\mu}z_\lambda^{00}, \tag{42}$$

where the ${}^{\mu}\alpha$ and ${}^{\mu}\beta$ are to be obtained from the matching conditions for $\mu = 1, 2, \ldots, M-1$.

$${}^{\mu}\alpha^{10} = {}^{\mu-1}\alpha^{10\mu-1} y_L^{10} + {}^{\mu-1}\beta^{10\mu-1} y_L^{01}, \tag{43}$$

$${}^{\mu}\beta^{10} = {}^{\mu-1}\alpha^{10\mu-1} y_{L+1}^{10} + {}^{\mu-1}\beta^{10\mu-1} y_{L+1}^{01}, \tag{44}$$

$${}^{\mu}\alpha^{01} = {}^{\mu-1}\alpha^{01\mu-1} y_L^{10} + {}^{\mu-1}\beta^{01\mu-1} y_L^{01}, \tag{45}$$

$${}^{\mu}\beta^{01} = {}^{\mu-1}\alpha^{01\mu-1} y_{L+1}^{10} + {}^{\mu-1}\beta^{01\mu-1} y_{L+1}^{01}, \tag{46}$$

$${}^{\mu}\alpha^{00} = {}^{\mu-1}\alpha^{00\mu-1} y_L^{10} + {}^{\mu-1}\beta^{00\mu-1} y_L^{01} + {}^{\mu-1}z_L^{00}, \tag{47}$$

$$ {}^{\mu}\beta^{00} = {}^{\mu-1}\alpha^{00\mu-1}y^{10}_{L+1} + {}^{\mu-1}\beta^{00\mu-1}y^{01}_{L+1} + {}^{\mu-1}z^{00}_{L+1}, \tag{48} $$

with the starting values

$$ {}^{0}\alpha^{10} = 1; \quad {}^{0}\alpha^{01} = 0; \quad {}^{0}\beta^{10} = 0; \quad {}^{0}\beta^{01} = 1; \quad {}^{0}\alpha^{00} = {}^{0}\beta^{00} = 0. \tag{49} $$

Using the last two values of the global basis functions calculated from Eqs. (40)–(42), we can solve for the global α and β (from Eqs. (36) and (38)) to write

$$ x_{\mu L+\lambda} = {}^{\mu}\alpha^{\mu}y^{10}_{\lambda} + {}^{\mu}\beta^{\mu}y^{01}_{\lambda} + {}^{\mu}z^{00}_{\lambda}; \quad \lambda = 2,3,\ldots,L+1 \ \ [\mu = 0,1,\ldots,M-1], \tag{50} $$

where the coefficients are given by

$$ {}^{\mu}\alpha = \alpha^{\mu}\alpha^{10} + \beta^{\mu}\alpha^{01} + {}^{\mu}\alpha^{00}, \tag{51} $$

$$ {}^{\mu}\beta = \alpha^{\mu}\beta^{10} + \beta^{\mu}\beta^{01} + {}^{\mu}\beta^{00}. \tag{52} $$

It is common to compare the relative speed of any algorithm for solving tri-diagonal matrices with Gaussian Elimination (GE) which is relatively efficient. For this case, GE becomes first for the LU reduction

$$ d_0 = 1/a_0 $$

$$ g_i = b_i d_{i-1}; \quad d_i = 1/(a_i - g_i), \quad i = 1,2,\ldots,N+1, \tag{53} $$

followed by the two back substitutions

$$ z^{g}_{0} = c_0; \quad z^{g}_{i} = c_i - g_i z^{g}_{i-1}, \quad i = 1,2,\ldots,N+1 \tag{54} $$

and

$$ x^{g}_{N+1} = z^{g}_{N+1}; \quad x^{g}_{i} = (z^{g}_{i} - x^{g}_{i+1})d_i, \quad i = N,N-1,\ldots,0. \tag{55} $$

If we assume that the equations are being solved for many right-hand sides, then we should compare the time for the solutions of the z^{00} equations and the calculation of the vector (Eqs. (20) and (50)) with the work of the two back substitutions in GE (Eqs. (54) and (55)). A first estimation can be made for the relative speed by counting the number of floating point operations per step (4 for GE and 8 for the parallel algorithm) to get a cost factor of 2. This is only a very crude estimate since the form of the equations is different. For example, (50) requires only the broadcast of a scalar instead of vector multiplication. Optimization or not of the G77 compiler was observed to make a large difference also. With no optimizing GE does better than this estimate giving a cost factor of 2.6. However, with optimization, the efficiency of the parallel method is improved more than GE to result in a cost factor of 1.4.

To save on message passing for the resultant vector, one may want to use the strips in the processor in which they were formed. In some cases, it may be more efficient never to construct the vectors at all. As an example of such a case, consider the problem of solving the set of equations for a large number of different right-hand side vectors which are a function of some parameter, η, hence, $c_i(\eta)$. Suppose also that we wish the sum (an integral perhaps) of some weighting function over the solution

$$ S(\eta) = \sum_{i=0}^{N+1} w_i x_i(\eta) \tag{56} $$

as a function of η. The sum can be distributed among the strip basis functions in the processors via Eq. (50). The y basis function integrals need be calculated only once. The z^{00} integral can be calculated as this basis function is generated. Only the strip *integrals* need to be sent to the master processor to be combined with the coefficients ${}^{\mu}\alpha$ and ${}^{\mu}\beta$. The calculation of the solution (Eq. (50)) is not needed. In this special case, a count of the number of floating point operations estimates the speed to be the same as GE (in the limit

of large N and large number of values of η). In one-processor tests, because of the simplicity of the equations mentioned above, the strip algorithm was found to run somewhat faster than GE.

Large systems ($720{,}720\times k$ with $k=10,20,40,80,$ and 160) with 100 values of η were tested on the 16 processor Beowulf cluster. Essentially no degradation of performance was seen with all scaling efficiencies $\geqslant 0.99$. The largest system tested ($N=115{,}315{,}200$) could only be run by spreading the solution basis vectors over 13 processors.

A common algorithm discussed in the literature is the parallel cyclic reduction of a matrix [8]. The basic cost of this algorithm has been reported to be a factor of 4 [9,10]. It requires frequent exchange of information and is not very efficient for multiple right-hand sides. The "divide-and-conquer" method [8] is also inefficient for multiple driving terms. Hence, the technique presented here would seem to offer an attractive alternative to these methods in some cases.

The restriction to $e_i\neq 0$ may prove to be inconvenient in some cases or the division may lead to large errors. Tests with $e_i\equiv 1$ showed that the stability of the method was as good or better than GE.

4. Discussion

These algorithms may also be useful on vector machines. For a processor with a 64 word vector register, for the case of the homogeneous recursion relation, the total length can be broken into 32 strips with each pair of words in the vector register acting as a processor. Thus, the iteration might take place as

$$\begin{pmatrix} y^{10}_{i+1}\\ y^{01}_{i+1}\\ y^{10}_{L+i+1}\\ y^{01}_{L+i+1}\\ \cdots\\ \cdots\\ y^{10}_{\mu L+i+1}\\ y^{01}_{\mu L+i+1}\\ \cdots\\ \cdots\\ y^{10}_{31L+i+1}\\ y^{01}_{31L+i+1} \end{pmatrix} = \begin{pmatrix} a_i\\ a_i\\ a_{L+i}\\ a_{L+i}\\ \cdots\\ \cdots\\ a_{\mu L+i}\\ a_{\mu L+i}\\ \cdots\\ \cdots\\ a_{31L+i}\\ a_{31L+i} \end{pmatrix} \times \begin{pmatrix} y^{10}_{i}\\ y^{01}_{i}\\ y^{10}_{L+i}\\ y^{01}_{L+i}\\ \cdots\\ \cdots\\ y^{10}_{\mu L+i}\\ y^{01}_{\mu L+i}\\ \cdots\\ \cdots\\ y^{10}_{31L+i}\\ y^{01}_{31L+i} \end{pmatrix} + \begin{pmatrix} b_i\\ b_i\\ b_{L+i}\\ b_{L+i}\\ \cdots\\ \cdots\\ b_{\mu L+i}\\ b_{\mu L+i}\\ \cdots\\ \cdots\\ b_{31L+i}\\ b_{31L+i} \end{pmatrix} \times \begin{pmatrix} y^{10}_{i-1}\\ y^{01}_{i-1}\\ y^{10}_{L+i-1}\\ y^{01}_{L+i-1}\\ \cdots\\ \cdots\\ y^{10}_{\mu L+i-1}\\ y^{01}_{\mu L+i-1}\\ \cdots\\ \cdots\\ y^{10}_{31L+i-1}\\ y^{01}_{31L+i-1} \end{pmatrix} \tag{57}$$

for $i=1,2,\ldots,L$.

The method can be generalized for a larger number of terms in the iteration (leading to larger width in banded matrices, for example).

There are clearly some limitations to the application of the algorithm. The conversion to a parallel system does not work for recursions nonlinear in x_i so most classical mechanics calculations are not possible with it.

While the problems treated are different, this method appears to have some overlap with the Domain Decomposition techniques for the solution of elliptic differential equations [11].

Acknowledgements

I thank Slava Solomatov for discussions and Alexei Vezolainen for help with one of Beowulf clusters in the Department of Physics. This work was supported by the National Science Foundation under Contract PHY-0099729.

References

[1] Jagdish J. Modi, Parallel Algorithms and Matrix Computation, Claredon Press, Oxford, 1988.
[2] A. Basermann, P. Weidner, A parallel algorithm for determining all eigenvalues of large real symmetric tridiagonal matrices, Parallel Computing 18 (1992) 1129–1141.
[3] F. Tisseur, J. Dongarra, A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures, SIAM J. Sci Comput. 20 (1999) 2223–2236.
[4] W.R. Gibbs, Computation in Modern Physics, World Scientific Publishing, Singapore, 1999.
[5] R.T. Gregory, D.L. Karney, A Collection of Matrices for Testing Computational Algorithms, Krieger Publ. Huntington, NY, 1978.
[6] W. Gropp, E. Lusk, N. Doss, A. Skjellum, Parallel Computing 22 (1996) 789.
[7] William D. Gropp, Ewing Lusk, User's Guide for `mpich`, a Portable Implementation of MPI, ANL-96/6, 1996.
[8] Duc Thai Nguyen, Parallel-Vector Equation Solvers for Finite Element Engineering Applications, Kluwer Academic/Plenum Publishers, 2002.
[9] I.N. Hajj, S. Skelboe, A multilevel parallel solver for block tridiagonal and banded linear systems, Parallel Computing 15 (1989) 21–45.
[10] F. Reale, A tridiagonal solver for massively parallel computer systems, Parallel Computing 16 (1990) 361–368.
[11] P.J. Roache, Elliptic Marching Methods and Domain Decomposition, CRC Press, 1995.