



ELSEVIER

Parallel Computing 21 (1995) 1769–1782

PARALLEL
COMPUTING

Algorithm for solving tridiagonal matrix problems in parallel

Nathan Mattor^{*}, Timothy J. Williams, Dennis W. Hewett

Lawrence Livermore National Laboratory, Box 808, Livermore, CA 94550, USA

Received 20 September 1993; revised 24 February 1994, 9 August 1994, 30 April 1995

Abstract

A new algorithm is presented, designed to solve tridiagonal matrix problems efficiently with parallel computers (multiple instruction stream, multiple data stream (MIMD) machines with distributed memory). The algorithm is designed to be extendable to higher order banded diagonal systems.

Keywords: Tridiagonal systems; Linear algebra; Distributed memory multiprocessor; Banded diagonal matrix; Interprocessor communication

1. Introduction

Currently, there are several popular methods for parallelization of the tridiagonal problem. The “most important” of these have recently been described with a unified approach, through *parallel factorization* [1]. Essentially, parallel factorization divides and solves the problem by the following steps:

- (1) Factor the original matrix into a product of a block matrix (that can be divided up between processors) and a reduced matrix, which couples the block problems.
- (2) Solve each block problem with one processor.
- (3) Solve the reduced matrix problem.

Here, we propose a new approach to parallel solution of such systems. It is conceptually different from parallel factorization, in that the first step is avoided: no manipulations are performed on the original matrix before subdividing it among

^{*} Corresponding author. Email: mattor@m5.llnl.gov

the processors. Avoiding this step has three advantages: simplicity, speed, and one less stability concern.

The method here is analogous to the solution of an inhomogeneous linear differential equation, where the solution is a “particular” solution added to an arbitrary linear combination of “homogeneous” solutions. The coefficients of the homogeneous solutions are later determined by boundary conditions. In our parallel method, each processor is given a contiguous subsection of a tridiagonal system. Even with no information about the neighboring subsystems, the solution can be obtained up to two constants. This completes the bulk of the necessary calculations. Once each processor has obtained such a solution, the global solution can be found by matching endpoints.

This approach allows a number of desirable features in both generality and efficient implementation. First, the method is readily generalizable to 5-diagonal and higher banded systems, as discussed in Section 6. Second, the particular and homogeneous solutions can be calculated quite efficiently, since there are a number of overlapping calculations. Usual serial LU decomposition of a single $M \times M$ tridiagonal system requires $8M$ floating point operations and a temporary storage array of M elements [3,6]. For the parallel routine below, the three solutions (one particular and two homogeneous) are calculated with $13M$ operations (not $3 \times 8M = 24M$) and *no* additional temporary storage arrays, while leaving the input data intact. Third, the interprocessor communication can be performed quite efficiently, by the all-to-all broadcast method described in Section 4. Finally, vector processors can be utilized effectively in cases where there are a number of banded diagonal systems to be solved.

This paper gives an implementation of this method. The algorithm is designed with the following objectives, listed in order of priority. First, the algorithm must minimize the number of interprocessor communications opened, since this is the most time consuming process. Second, the algorithm allows flexibility of the specific solution method of the tridiagonal submatrices. Here, we employ a variant of LU decomposition, but this is easily replaced with cyclic reduction or other. Third, we wish to minimize storage needs.

The remainder of this paper is organized as follows. Section 2 outlines the analysis underlying the routine. Section 3 describes an algorithm for computing the particular and two homogeneous solutions in $13M$ operations. Section 4 gives a method to assemble the reduced system efficiently in each processor, solve it, and complete the solution. Section 5 covers time consumption and performance of the algorithm. Section 6 gives some conclusions and generalizations of this routine. The Appendix gives program segments.

2. Basic algorithm

We consider the problem of solving the $N \times N$ linear system

$$AX = R, \tag{2-1}$$

where $x_{p,m}$ refers to the m th element of the appropriate solution from the p th submatrix.

The solution of Eq. (2-1), via subdivision and reassembly, is now complete. The outline of the algorithm is as follows:

- (1) For each processor, find x_p^R , x_p^{UH} , and x_p^{LH} by solving Eqs. (2-9)–(2-11).
- (2) Assemble the “reduced” system, Eq. (2-13), and solve for the ξ_p^{UH} and ξ_p^{LH} .
- (3) For each processor, compute the final solution by Eq. (2-12).

This algorithm is our main result. The remainder of this paper discusses practical details of reducing operation count, memory requirement, and communication time.

3. Computing the particular and homogeneous solutions

The three solutions x_p^R , x_p^{UH} , and x_p^{LH} are obtained by solving Eqs. (2-9)–(2-11). The method here is based on LU decomposition, which is usually the most efficient. With vector processors, cyclic reduction may be more efficient [4], but only if just one system need be solved. Many applications require solution of multiple tridiagonal systems, for which the most efficient use of vectorization is to use LU decomposition and vectorize across the multiple systems. This is discussed further in Section 6.

Equations (2-9)–(2-11) represent three $M \times M$ tridiagonal systems. If these systems were independent, then solution via LU decomposition would require $3 \times 8M = 24M$ binary floating point operations. However, exploiting overlapping calculations and elements with value 0, gives the following algorithm, which can be implemented with $13M$ binary floating point operations.

Forward elimination:

$$\omega_1 = \frac{c_1}{b_1} \quad \omega_i = \frac{c_i}{b_i - a_i \omega_{i-1}} \quad i = 2, 3, \dots, M$$

$$\gamma_1 = \frac{r_1}{b_1} \quad \gamma_i = \frac{r_i - a_i \gamma_{i-1}}{b_i - a_i \omega_{i-1}} \quad i = 2, 3, \dots, M$$

Back substitution:

$$x_M^R = \gamma_M \quad x_i^R = \gamma_i - \omega_i x_{i+1}^R \quad i = M - 1, M - 2, \dots, 1$$

$$x_M^{LH} = -\omega_M \quad x_i^{LH} = -\omega_i x_{i+1}^{LH} \quad i = M - 1, M - 2, \dots, 1$$

$$\omega_M^{UH} = \frac{a_M}{b_M} \quad \omega_i^{UH} = \frac{a_i}{b_i - c_i \omega_{i+1}^{UH}} \quad i = M - 1, M - 2, \dots, 1$$

Forward substitution:

$$x_1^{UH} = -\omega_1^{UH} \quad x_i^{UH} = -\omega_i^{UH} x_{i-1}^{UH} \quad i = 2, 3, \dots, M,$$

where the processor index p is implicitly present on all variables, and we have assumed that end elements a_1 and c_M are written in the appropriate positions in the a and c arrays. This algorithm is similar to the usual LU decomposition algorithm, (see, e.g. [6] or [3]), but with an extra forward substitution. The sample FORTRAN segment in the Appendix implements this with no temporary storage arrays.

If the tridiagonal matrix is constant, and only the right hand side changes from one matrix problem to the next, then the vectors ω_i , $1/(b_i - a_i\omega_i)$, x_i^{UH} , and x_i^{LH} can be precalculated and stored. The computation then requires only $5M$ binary floating point operations.

4. Construction and solution of the reduced matrix

Once each processor has determined \mathbf{x}_p^R , \mathbf{x}_p^{UH} , and \mathbf{x}_p^{LH} , it is time to construct and solve the reduced system of Eq. (2-13). This section describes an algorithm for this.

We assume that the following subroutines are available for interprocessor communication:

- **Send(ToPid, data, n)**: When this is invoked by processor FromPid, the array *data* of length n is sent to processor ToPid. It is assumed that **Send** is *nonblocking*, in that the processor does not wait for the data to be received by ToPid before continuing.
- **Receive(FromPid, data, n)**: To complete data transmission, **Receive** is invoked by processor ToPid. Upon execution, the array sent by processor FromPid is stored in the array *data* array of length n . It is assumed that **Receive** is *blocking*, in that the processor waits for the data to be received before continuing.

Opening interprocessor communications is generally the most time-consuming step in the entire tridiagonal solution process, so it is important to minimize this. The following algorithm consumes a time of $T = (\log_2 P)t_c$ in opening communication channels (where t_c is the time to open one channel).

- (1) Each processor writes whatever data it has that is relevant to Eq. (2-13) in the array **OutData**.
- (2) The **OutData** arrays from each processor are concatenated as follows (Fig. 1):
 - (a) Each processor p sends its **OutData** array to processor $p - 1 \pmod{P} + 1$, and receives a corresponding array from processor $p + 1 \pmod{P} + 1$, as depicted in Fig. 1(a). The incoming array is concatenated to the end of **OutData**.
 - (b) At the i th step, repeat the first step, except sending to processor $p - 2^{i-1} \pmod{P} + 1$, and receiving from processor $p + 2^{i-1} \pmod{P} + 1$ (Fig. 1b,c), for $i = 1, 2, \dots$. After $\log_2 P$ iterations (or the next higher integer), each processor has the contents of the reduced matrix in the **OutData** array.
- (3) Each processor rearranges the contents of its **OutData** array into the reduced tridiagonal system, and then solves. (Each processor solves the same reduced system.)

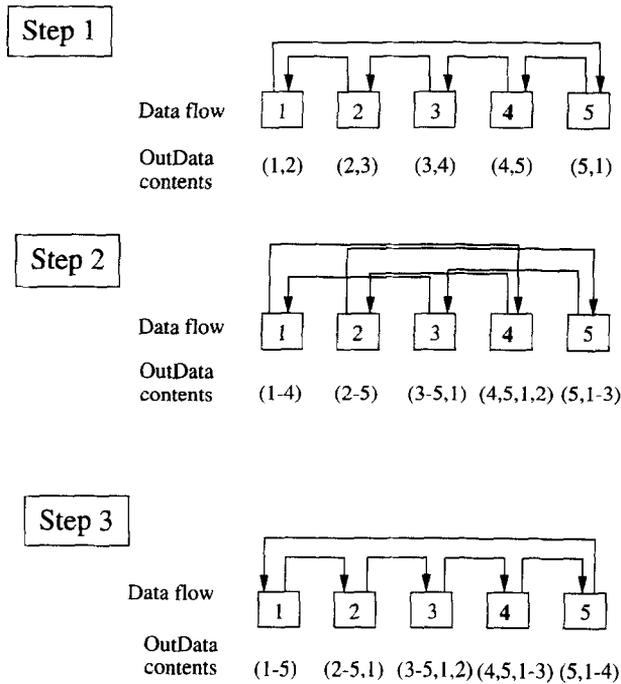


Fig. 1. Illustration of the method used to pass reduced matrix data between processors, with $P = 5$ for definiteness.

step 1: In the first iteration, each processor p sends data to processor $(p - 1) \pmod P + 1$, receives data from processor $(p + 1) \pmod P + 1$, and concatenates the data arrays. The result is that each `OutData` array contains the data from the processors shown, in order shown. The remaining elements of the `OutData` array are not used.

step 2: Each processor sends its `OutData` array to processor $(p - 2) \pmod P + 1$, receives data from processor $(p + 2) \pmod P + 1$, again concatenating.

step 3: In the i th iteration (here third and final), each processor sends its `OutData` array to processor $(p - 2^{i-1}) \pmod P + 1$, receives data from processor $(p + 2^{i-1}) \pmod P + 1$, again concatenating. The final `OutData` array contains all the information of the reduced matrix, ordered cyclically beginning with the contributions of the p th processor. The information beyond the element `OutData(8P)` is not used.

This communication is dense (every processor communicates in every step), and periodic, so that upon completion *every* processor contains the fully concatenated `OutData` array). A sample program segment is provided in the Appendix.

If the elements of the tridiagonal matrix are constants, then the reduced matrix can be precalculated and only the reduced right hand side needs to be assembled. In this case, the above routine could be rewritten to pass 1/4 as many real numbers. This does not represent a very large time savings, since generally it is the channel openings that is the most costly, not the amount of data passage.

At this point, the necessary components to Eq. (2-12) are stored in each processor. All that is left is the trivial task of picking out the correct coefficients

and constructing the final solution. A sample program segment is provided in the Appendix.

5. Performance

In this section we discuss execution time of this algorithm, and present scaling tests made with a working code.

The time consumption for this routine is as follows.

- (1) To calculate the three roots \mathbf{x}^R , \mathbf{x}^{UH} , and \mathbf{x}^{LH} requires $13M$ binary floating point operations by each processor, done in parallel.
- (2) To assemble the reduced matrix in each processor requires $\log_2 P$ steps where interprocessor communications are opened, and the i th opening passes $8 \times 2^{i-1}$ real numbers.
- (3) Solution of the reduced system through LU decomposition requires $8(2P - 2)$ binary floating point operations by each processor, done in parallel.
- (4) Calculation of the final solution requires $4M$ binary floating point operations by each processor, done in parallel.

If t_b is the time of one binary floating point operation, t_c is the time required to open a communication channel (latency), and t_p is the time to pass one real number once communication is opened, then the time to execute this parallel routine is given by (optimally)

$$\begin{aligned} T_p &= 13Mt_b + (\log_2 P)t_c + 8(P - 1)t_p + 8(2P - 2)t_b + 4Mt_b \\ &\approx (17M + 16P)t_b + (\log_2 P)t_c + 8Pt_p, \end{aligned} \quad (5-1)$$

for $P \gg 1$. For cases of present interest, T_p is dominated by $(\log_2 P)t_c$ and $17Mt_b$. The *parallel efficiency* is defined by $\epsilon_p \equiv T_s/PT_p$, where T_s is the execution time of a serial code which solves by LU decomposition. Since serial LU decomposition solve an $N \times N$ system in a time $T_s = 8Nt_b$, then the predicted parallel efficiency is

$$\epsilon_p = \frac{8}{17 + 16P^2/N + (\log_2 P)Pt_c/Nt_b + 8P^2t_p/Nt_b}. \quad (5-2)$$

To test these claims empirically, the execution times of working serial and parallel codes were measured, and ϵ_p was calculated both through its definition and through Eq. (5-2). Fig. 2 shows ϵ_p as a function of P for two cases, $N = 200$ and $N = 50,000$. We conclude from Fig. 2 that Eq. (5-2) (smooth lines) is reasonably accurate, both for the theoretical maximum efficiency (47%, achieved for small P and large N) and for the scaling with large P .

These timings were made on the BBN TC2000 MIMD machine at Lawrence Livermore National Laboratory. This machine has 128 M88100 RISC processors, connected by a butterfly-switch network. To calculate the predictions of Eq. (5-1), the times t_c , t_p , and t_b were obtained as follows. We chose $t_c = 750 \mu\text{sec}$, based on the average time of a send/receive pair measured in our code. Based on communications measurements for the BBN [5], we chose the passage time of a single 64-bit

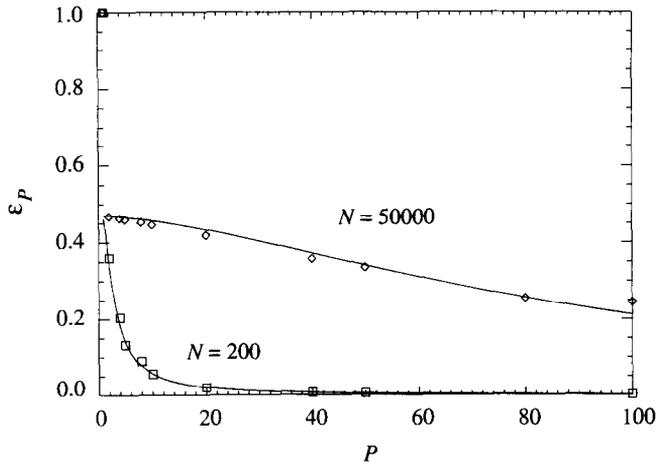


Fig. 2. Results of scaling runs, comparing the parallel time with serial LU decomposition time. Here, ϵ_P is the parallel efficiency and P is the number of processors. The smooth lines represent Eq. (5-2), and the points are empirical results. The upper line and diamonds use $N = 200$, and the lower line and squares use $N = 50,000$.

real number as $t_p = 9 \mu\text{sec}$. [In comparison, the peak bandwidth specification for the machine of 38 MB/sec per path [2] would yield $t_p = 0.2 \mu\text{sec}$. Using this value instead of $9 \mu\text{sec}$ makes no visible differences in Fig. 2.] We chose $t_b = 1.4 \mu\text{sec}$, based on our measured timing of 0.00218 sec for the serial algorithm on the $N = 200$ case. (In comparison, the peak performance specification of 10 MFLOPS for the M88100 on 64-bit numbers would yield $t_b = 0.1 \mu\text{sec}$. Using this value instead of $1.4 \mu\text{sec}$ would yield curves which fall well below our measured parallel efficiency values.) All measurements were made with 64-bit floating point arithmetic.

It is difficult to make a comprehensive comparison with all other parallel tridiagonal solvers, but we can compare our speed with a popular algorithm by H. Wang [7]. That routine requires $2P - 2$ steps where communications are opened (compared with $\log_2 P$ such steps here), and $21M$ binary operations per processor (compared with $17M$ here). We have not performed empirical comparisons, but since both dominant processes have lower counts, it seems reasonable to believe the present algorithm is generally faster.

6. Discussion and conclusions

In this paper, we have described a numerical routine for solving tridiagonal matrices using a multiple instruction stream/multiple data stream (MIMD) parallel computer with distributed memory. The routine has the advantage over existing methods in that the initial factorization step is not used, leading to a simpler, and probably faster, routine.

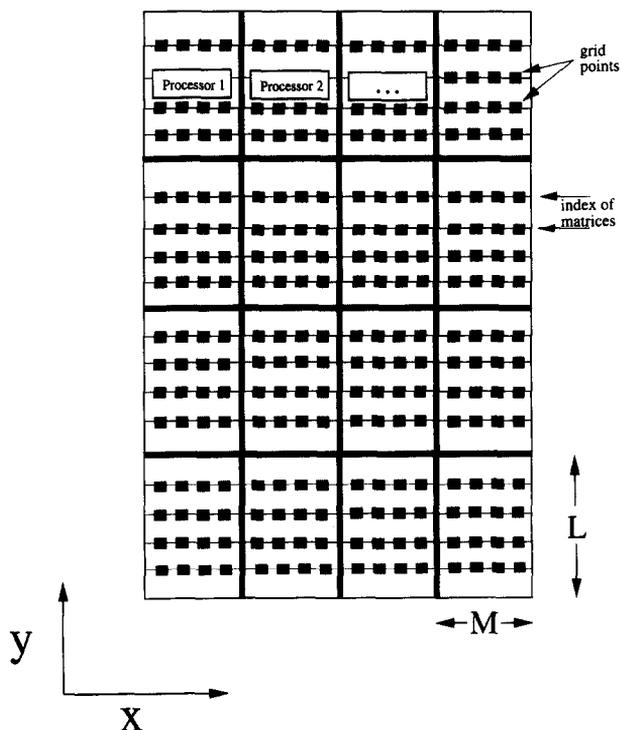


Fig. 3. Schematic representation of a typical set of tridiagonal systems that might arise in a two dimensional grid. For each grid line in y , there is a tridiagonal system to solve.

Stability of this algorithm is similar to that of serial LU decomposition of a tridiagonal matrix. If the L_i are unstable to LU decomposition, then pivoting could be used. If the L_i are singular, then LU decomposition fails and some alternative should be devised. If the large matrix A is diagonally dominant, ($|A_i| > |B_i| + |C_i|$) then so too are the L_i . If the reduced system is unstable to LU decomposition, this can be replaced by a different solution scheme, with little loss of overall speed (if $P \ll M$).

This routine is generalizable from tridiagonal to higher systems. For example, in a 5-diagonal system, there would be four homogeneous solutions, each with an undetermined coefficient. The coefficients of the homogeneous solutions would be determined by a reduced system analogous to Eq. (2-13), except with $O(4P)$ equations, not $2P - 2$.

We briefly discuss implementation of this routine in a problem where there are many tridiagonal systems to solve. This situation arises in many of the foreseeable applications of parallel tridiagonal solvers, such as solving differential equations by the alternating-direction implicit method (ADI) [6] on a multidimensional grid. For definiteness we consider the two dimensional grid depicted in Fig. 3. For each grid line in y , there is a tridiagonal system to solve, with the index running in the x

direction. The routine described in this paper is well suited for this problem, which may be handled efficiently as follows.

- (1) The grid is divided into block subdomains in x and y . Each processor is assigned one subdomain, with the submatrices and sub-right hand sides stored locally. If the number of y grid lines in each subdomain is L , and the number of x grid points is M , then each processor has L systems with M equations each.
- (2) Each processor finds \mathbf{x}^R , \mathbf{x}^{UH} , and \mathbf{x}^{LH} , for each y grid line in its subdomain, by the algorithm described in Section 3.
- (3) To assemble the reduced systems, each family of processors colinear in x passes the reduced system data for *all* y grid lines in the subdomain, by the algorithm described in Section 4. (Note that the number of communication openings can be minimized by passing all L reduced systems together.) After this step is complete, each processor contains L reduced systems, one for each y grid line in its subdomain.
- (4) Each processor solves the L reduced systems, then assembles the final solution, as described at the end of Section 4.

This method has several desirable features. First, the time spent opening interprocessor communications is $P_x \log_2 P_x$ (where P_x is the number of processors colinear in x), which is not greater than for solving a single system with P_x processors. Insofar as this is the most time consuming step, multidimensional efficiency is quite good for this algorithm. Second, if the processors are vector processors, the calculations for the y grid lines (steps 1 and 3 above) can be carried out in parallel. This would seem to be a more efficient use of vectorization than replacing the LU decomposition with cyclic reduction, since the former involves fewer operations. Third, the algorithm is easily converted to a system where the roles of x and y are reversed; all that needs to be done is exchange indices. Complicated rearrangement of subdomains is not necessary.

Acknowledgement

This work was performed for the U.S. Department of Energy at Lawrence Livermore National Laboratory under contract W7405-ENG-48.

Appendix

This Appendix gives sample FORTRAN routines for the algorithms in Sections 3 and 4. This makes the algorithms more concrete, and also gives some time and memory saving steps not mentioned above.

The particular and homogeneous solutions for each submatrix are computed by the algorithm in Section 3. This is to be run by each processor, and it is assumed that the arrays $a_1^p \dots a_M^p$, $b_1^p \dots b_M^p$, $c_1^p \dots c_M^p$, and $r_1^p \dots r_M^p$ are stored locally in each processor, with the index p omitted. No temporary arrays are needed; all

intermediate storage is done in the final solution arrays, *xr*, *xuh*, and *xlh* (each with *M* elements).

!forward elimination:

```
xuh(1) = c(1) / b(1)
xlh(1) = r(1) / b(1)
do i = 2, M, 1
  denom = b(i) - c(i)*xuh(i-1)
  if (denom.eq.0) pause !LU decomposition fails
  xuh(i) = c(i) / denom
  xlh(i) = (r(i) - a(i)*xlh(i-1)) / denom
end do
```

!back substitution:

```
xr(M) = xlh(M)
xlh(M) = -xuh(M)
xuh(M) = a(M) / b(M)
do i = M-1, 1, -1
  xr(i) = xlh(i) - xuh(i)*xr(i+1)
  xlh(i) = -xuh(i)*xlh(i+1)
  denom = b(i) - c(i)*xuh(i+1)
  if (denom.eq.0) pause !LU decomposition fails
  xuh(i) = -a(i) / denom
end do
```

!forward substitution:

```
xuh(1) = -xuh(1)
do i = 2, M, 1
  xuh(i) = -xuh(i)*xuh(i-1)
end do
```

Section 4 describes how the reduced matrix is written to each processor. A sample routine follows, to be executed by each processor. We assume the processors are numbered $p = 1, 2, \dots, P$. The integer *pid* is the local processor number (called *p* in the mathematical parts of this paper), and the integer *nprocs* is the code name for *P*. The integer $\log_2 P$ is the smallest integer greater than or equal to $\log_2(P)$. The real array *OutData* has $8 \times 2^{\log_2 P}$ elements. The subroutine *tridiagonal* (*a, b, c, r, sol, n*) (not given here) is a serial subroutine that returns the solution *sol* for the tridiagonal system with subdiagonal *a*, diagonal *b*, superdiagonal *c*, and right hand side *r*, all of length *n*.

!write contributions of current processor into OutData:

```
OutData(1) = -1.
Outdata(2) = xuh(1)
Outdata(3) = xlh(1)
Outdata(4) = -xr(1)
```

```

Outdata(5) = xuh(M)
Outdata(6) = xlh(M)
OutData(7) = -1.
OutData(8) = -xr(M)

```

!concatenate all the OutData arrays:

```

log2P = log(nprocs) / log(2)
if (2**log2P.lt.nprocs) log2P = log2P + 1
do i=0, log2P-1, 1
  nxfer = 8*(2**i)
  ToProc = 1+mod(pid-2**i+2*nprocs,nprocs)
  FromProc = 1+mod(pid+2**i,nprocs)
  call Send(ToProc,OutData,nxfer)
  call Receive(FromProc,OutData(nxfer+1),nxfer)
end do

```

!Put OutData into reduced tridiagonal form:

```

nsig = 8*nprocs !no. of significant entries in OutData
ifirst = 8*(nprocs-pid) + 5 !index of a(1) in OutData
do i=1, 2*nprocs-2, 1
  ibase = mod(ifirst+4*(i-1),nsig)
  reduca(i) = OutData(ibase)
  reducb(i) = OutData(ibase+1)
  reducc(i) = OutData(ibase+2)
  reducr(i) = OutData(ibase+3)
end do

```

!solve reduced system:

```

call tridiagonal(reduca,reducb,reducc,reducr,coeffs,
2*nprocs-2)

```

Once the reduced matrix is solved, then the solution can be assembled in each processor as follows.

!pick out the appropriate elements of coeffs:

```

if (pid.ne.1) then
  uhcoeff = coeffs(2*pid-2)
else
  uhcoeff = 0.
end if
if (pid.ne.nprocs) then
  lhcoeff = coeffs(2*pid-1)
else
  lhcoeff = 0.
end if

```

!compute the final solution:

```
do i = 1, M, 1
  x(i) = xr(i) + uhcoeff*xuh(i) + lhcoeff*xlh(i)
end do
```

References

- [1] P. Amodio and L. Brugnano, Parallel factorizations and parallel solvers for tridiagonal linear systems, *Linear Algebra and Its Applications* 172 (1992) 347–364.
- [2] BBN Advanced Computers Inc., *Inside the TC2000 Computer* (Cambridge, MA, 1989).
- [3] R.W. Hockney and J.W. Eastwood, *Computer Simulation Using Particles* (Adam Hilger, Bristol, 1988) 185.
- [4] R.W. Hockney and C.R. Jesshope, *Parallel Computers 2* (IOP, Bristol, 1988).
- [5] C.E. Leith, Domain decomposition message passing for diffusion and fluid flow, in *The MPC1 Yearly Report: The Attack of the Killer Micros*, UCRL-ID-107022 (1991).
- [6] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes* (Cambridge University Press, 1986) 19–40.
- [7] H.H. Wang, A parallel method for tridiagonal equations, *ACM Trans. Math. Software* 7 (1981) 170–183.