# Experiments with an ordinary differential equation solver in the parallel solution of method of lines problems on a shared-memory parallel computer *

D.K. Kahaner
*National Institute of Standards and Technology, Gaithersburg, MD 20899, United States*

E. Ng
*Oak Ridge National Laboratory, Oak Ridge, TN, United States*

W.E. Schiesser
*Lehigh University, Bethlehem, PA, United States*

S. Thompson
*Radford University, Radford, VA, United States*

Received 3 September 1990
Revised 1 June 1991

*Abstract*

Kahaner, D.K., E. Ng, W.E. Schiesser and S. Thompson, Experiments with an ordinary differential equation solver in the parallel solution of method of lines problems on a shared-memory parallel computer, Journal of Computational and Applied Mathematics 38 (1991) 231–253.

We consider method of lines solutions of partial differential equations on shared-memory parallel computers. Solutions using the ordinary differential equation solver SDRIV3 (which is similar to the well-known LSODE solver) are considered. It is shown that portions of the solver may be implemented in parallel. In particular, formation of the Jacobian matrix and the linear algebra required to solve the corrector equations are natural candidates for parallel implementation since these portions dominate the cost of solving large systems of equations. A variant of Gaussian elimination is described which allows efficient parallel solution of systems of

* The research on which this paper is based was performed in Oak Ridge, Tennessee at the Mathematical Sciences Section of the Oak Ridge National Laboratory, and supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, US Department of Energy, under contract DE-AC05-84OR21400 with Martin-Marietta Energy Systems, Inc.

linear equations. An implementation of SDRIV3 which performs the Jacobian related calculations in parallel and which uses this variant of Gaussian elimination is described. The modified solver is used to solve a model hyperbolic fluid flow problem. Timing results, obtained using a Sequent Balance parallel computer, are given which demonstrate that substantial speedups are possible. Extensions of the techniques to sparse problems are discussed and illustrated for a problem involving a humidification column which contacts air and water.

*Keywords:* Ordinary differential equations, ODE, numerical analysis, method of lines, MOL, mathematical software.

## 1. Introduction

This paper is a sequel to [14] which considered the solution of method of lines (MOL) problems on shared-memory parallel computers. A brief description of the method of lines is provided in the Appendix. The basic idea is to implement portions of a standard ordinary differential equation (ODE) solver in a parallel manner which takes advantage of the architecture of such a computer. Most operations performed by ODE solvers are inherently sequential in nature and do not lend themselves to parallel solution. There are important exceptions however. The two operations which dominate the cost of using such a solver are the approximation of the Jacobian matrix, and the linear algebra necessary to solve the corrector equations. This observation has generated considerable interest in the development of techniques which exploit problem structure to reduce the costs associated with these operations. For example, versions of the celebrated LSODE [11] ODE solver now exist for banded or sparse problems as well as several other types of problems (e.g., implicit equations). More recently, considerable attention has been devoted to the question of using iterative methods for the solution of the corrector equations [3].

Since these operations are amenable also to parallel implementation, it is natural to consider the feasibility of parallel solutions. This question was addressed for a simplified ODE solver and for LSODE in [14]. We will again consider this question in this paper. We chose to use the SDRIV3 ODE solver [13] which is similar in construction and in spirit to its relative LSODE. SDRIV3 will be used in conjunction with spatial differencing routines from [15] and parallel Gaussian elimination software. Results will be presented for two representative model problems. The results demonstrate that an essentially linear speedup in the number of processors is possible for such problems. Several techniques will be discussed including fully dense solutions, mixed dense-sparse solutions and, finally, (almost) fully sparse solutions. Since solvers such as LSODE and SDRIV3 (and their variants) currently represent the state-of-the-art for solving ODEs, any improvement in their performance is worthy of note. Even though we limit our attention to the use of SDRIV3 in this paper, we emphasize that the techniques discussed are applicable to most of the other well-known stiff ODE software. See [4] for a review of currently available software to which the techniques discussed in this paper are applicable.

The remainder of this paper is organized as follows. Section 2 describes the parallel Gaussian elimination techniques used. Section 3 describes the parallel method of lines techniques used. Section 4 describes the two model problems which were used to test the feasibility of solving method of lines problems in this context. Section 5 contains a discussion of selected numerical results for these two problems.

## 2. Parallel Gaussian elimination

In this section we describe an algorithm for the solution of dense systems of linear equations $Ax = b$ on multiprocessors with shared memory. Since a description of the algorithm does not appear elsewhere, considerable detail will be included herein. Here $A$ is an $n \times n$ matrix and $b$ is an $n$-vector. Our approach is to factor the matrix using Gaussian elimination with row interchanges. We denote the decomposition as

$$A = P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} U,$$

where $P_k$ is an elementary permutation matrix, $L_k$ is a unit lower triangular matrix whose $k$th column contains the multipliers at the $k$th step, and $U$ is the upper triangular factor. This section can be regarded as a sequel to [9] in which an algorithm was developed for solving dense positive definite systems in such an environment. The computing regime we adopt employs the notion of a *pool of tasks* whose parallel execution is controlled by a *self-scheduling discipline* [12]. In our context, the tasks are those computations associated with columns of $A$, and thus have a well-defined order associated with them.

In some parallel algorithms, specific tasks are mapped onto specific processors in advance of initiating the computation. In this situation, effective (static) load balancing among the processors requires that the distribution of work be reasonably uniform. Self-scheduling can be regarded as a mechanism for implementing dynamic load balancing; $p$ processes are initiated to perform $T$ tasks ($p \leqslant T$). When a given process completes a task, it checks to see if any unassigned tasks remain, and if so, it is assigned to the next one. Thus, if a process happens to have drawn a relatively small task, it will become free to perform another one sooner than a process occupied by a larger task. In this way, processors tend to be kept busy even if the tasks vary in their computational requirements. This self-scheduled pool-of-tasks approach is flexible in that it is not very strongly dependent on the number of processors available. Furthermore, this approach is appropriate for multiprocessors with shared memory since the pool of tasks must be made available to each processor.

We first present a parallel algorithm for Gaussian elimination with partial pivoting on a shared-memory multiprocessor. Parallel algorithms for the solution of $Ax = b$ using the resulting triangular factors are then discussed.

### *A serial algorithm*

The factorization algorithm we consider is column-oriented and is described below.

```
for $j = 1$ to $n$ do
   for $k = 1$ to $j - 1$ do
      apply $P_k$ to column $j$ of $A$;
      for $i = k + 1$ to $n$ do
         $a_{ij} := a_{ij} - a_{ik} * a_{kj}$;
         determine $P_j$ and apply $P_j$ to column $j$ of $A$;
         for $i = j + 1$ to $n$ do
            $a_{ij} := a_{ij}/a_{jj}$;
```

For convenience, we define exchange($j$, $k$), cmod($j$, $k$) and cdiv($j$) as follows:

- exchange($j$, $k$): the subtask of applying $P_k$ to column $j$ of $A$, $k \leqslant j$;
- cmod($j$, $k$): the subtask of modifying column $j$ by column $k$, $k < j$; and
- cdiv($j$): the subtask of dividing column $j$ by a scalar.

Using these subtasks, the factorization algorithm can be expressed in a compact form, as shown below.

```
for $j = 1$ to $n$ do
    for $k = 1$ to $j - 1$ do
        perform exchange($j$, $k$);
        perform cmod($j$, $k$);
    determine $P_j$;
    perform exchange($j$, $j$);
    perform cdiv($j$);
```

At the end of the algorithm, the lower and upper triangular parts of $A$ will be overwritten by the lower and upper triangular factors, respectively. In this approach, column $j$ of $A$ is modified by column $k$ of $L_k$, $1 \leqslant k \leqslant j - 1$, and then column $j$ of $L_j$ is computed.

It should be noted that this is not the only column-oriented algorithm for factoring $A$. Suppose column $j$ of $A$ has been modified by columns $k$ of $L_k$, $1 \leqslant k \leqslant j - 1$. An alternate approach is as follows. Column $j$ of $L_j$ is computed and then it is used immediately to modify column $k$ of $A$, for $j + 1 \leqslant k \leqslant n$. This second approach differs from the first one only in the order in which the computations are performed.

## *A parallel algorithm*

There is a fairly high degree of parallelism in the column-oriented factorization algorithm described in the previous subsection. Once a column of the triangular factors has been computed, it can be used to modify the remaining columns of $A$ and the modifications may be performed concurrently. Moreover, once $P_k$ has been determined, it can be applied to the remaining columns even before column $k$ of the triangular factors is computed. Thus this gives the following parallel algorithm for dense Gaussian elimination. The task of computing a column of the triangular factors is denoted by Factor. We will use an integer array ready for synchronization purposes. Column $j$ of the triangular factors has not been computed when ready[$j$] is zero. When $P_j$ has been determined, ready[$j$] is set to 1, and when column $j$ has been computed, ready[$j$] is set to 2. This allows the exchange($j$, $k$) and cdiv($k$) operations to be overlapped.

```
for $j = 1$ to $n$ do
    ready[$j$] := 0;
for $j = 1$ to $n$ do
    schedule Factor($j$);

Factor($j$):
    for $k = 1$ to $j - 1$ do
        wait until ready[$k$] > 0;
        perform exchange($j$, $k$);
        wait until ready[$k$] = 2;
        perform cmod($j$, $k$);
    determine $P_j$;
    ready[$j$] := 1;
    perform exchange($j$, $j$);
    perform cdiv($j$);
    ready[$j$] := 2;
```

It is important to note that hardware synchronization is not needed in the algorithm above. In particular, updating the ready is not a critical region since ready[$j$] is modified by only the processor which computes column $j$. Overall synchronization is achieved implicitly using the ready array. A final comment is that there are $n$ tasks to be scheduled in this approach.

There is an alternate way of computing the columns of the triangular factors. The tasks that are being performed in parallel are those of modifying the remaining columns of the matrix. More precisely, when column $j$ of $L_j$ is computed, it is used to modify the remaining $n-j$ columns of $A$ and the modifications are performed concurrently. Thus there are $n-j$ tasks to be scheduled at step $j$. A possible drawback of this approach is the cost of scheduling the tasks. If it is expensive to schedule a task, the latter algorithm wil probably take longer to execute since there are a total of $O(n^2)$ tasks to be scheduled.

*Parallel triangular solutions and post-processing of the triangular factors*

After the triangular factors ($L_k$ and $U$) have been computed, the solution to the original system is obtained by solving the two triangular systems which result:

$$P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} y = b,$$
$$Ux = y.$$

A column-oriented serial algorithm for solving the first triangular system is given below. The elements of the $j$th column of $L_j$ are ntoed by $l_{kj}$. The right-hand side vector $b$ is overwritten by the solution $y$.

```
for $j = 1$ to $n - 1$ do
    apply $P_j$ to $b$;
    for $k = j + 1$ to $n$ do
        $b_k := b_k - b_j * l_{kj}$;
```

This serial algorithm is difficult to parallelize, as the following discussion shows. Consider step 2 in the forward solve. In order to compute $y_2$, $b_2$ may have to be replaced by another component in the right-hand side vector, say $b_s$, depending on the permutation $P_2$. Then $b_2$ has to be modified by the product of $l_{21}$ and $x_1$. Suppose $x_1$ has been computed. It is then used, together with column 1 of $L_1$, to modify $b$. In a parallel implementation, we can compute $y_2$ once $x_1$ has been computed and $b_s$ has been modified by $l_{s1}$ and is available for the interchange. This may happen even before the right-hand side has been completely modified by column 1 of $L_1$. However, note that in the worst case, $l$ could be equal to $n$. Thus, this may potentially cause the parallel implementation to behave like the serial algorithm.

To alleviate this problem, we post-process the elements in $L_j$. The elements of the $j$th column of $L_j$ are permuted to the right order (according to $P_1, P_2, \ldots, P_j$) before the forward solve begins. This approach eliminates the need for interchanging the components of $b$ during forward solve; the permutations $P_j$ can be applied to $b$ before the forward solve begins. Note that these columns of $L_j$ are independent of each other and hence the post-processing can be performed concurrently.

### *A parallel forward solve algorithm*

We shall assume that the elements of the $j$th column of $L_j$ have been permuted to the right order. Following is a parallel algorithm for forward solve. It makes use of an integer array nmod. The value of nmod[$j$] at any time indicates the number of modifications that have been applied to the $j$th component of the right-hand side vector and it is initially set to 0. The array nmod is used to synchronize the computations so that each element of $b$ is modified only by one procesor at any time.

The task of computing the $j$th component of the solution vector is denoted by Forward-solve($j$).

```
for $j = 1$ to $n$ do
    nmod[$j$] := 0;
for $j = 1$ to $n$ do
    schedule Forward-solve($j$);

Forward-solve($j$):
    wait until nmod[$j$] = $j - 1$;
    for $i = j + 1$ to $n$ do
        wait until nmod[$i$] = $j - 1$;
        $b_i := b_i - b_j * l_{ij}$;
        nmod[$i$] := nmod[$i$] + 1;
```

### *A parallel backward solve algorithm*

The backward solve algorithm is similar to the forward solve algorithm. It makes use of an integer array flag; the value of flag[$j$] at any time indicates the solution component that can be used to modify $b_j$ and it is initially set to $n$. This array is used to synchronize the modifications of the elements of $b$. The elements of the upper triangular matrix $U$ are denoted by $u_{ij}$. The task of computing the $j$th component of the solution vector is denoted by Backward-solve($j$).

```
for $j = 1$ to $n$ do
    flag[$j$] := $n$;
for $j = 1$ to $n$ do
    schedule Backward-solve($j$);

Backward-solve($j$):
    wait until flag[$j$] = $j$;
    $x_j := b_j / u_{jj}$;
    for $i = j - 1$ to 1 by $-1$ do
    wait until flag[$i$] = $j$;
    $b_i := b_i - x_j * u_{ij}$;
    flag[$i$] := flag[$i$] − 1;
```

In the algorithm above, it appears that flag[$i$] would be modified by more than one processor at a given time. However this is not the case since the wait statement in the inner loop will force exactly one processor to update $b_i$ at any time. In the sequential algorithm, the order in which the right-hand side components are modified by $x_j$ at step $j$ is not crucial. However this is not the case in the parallel algorithm. In order to increase the degree of parallelism, the components $b_i$ must be modified in the order $j-1, j-2, \dots, 1$ at step $j$.

The parallel algorithms described above have been implemented on a Sequent Balance 8000 multiprocessor in FORTRAN. Several experiments were performed to test the efficiency of the algorithms. Preliminary experience is that the performance of the parallel factorization algorithm is very good with speedup ratios that approach the number of processors used as the problem size increases, and with efficiencies over 97% for large problems. Further details can be obtained from the second author. These results are reflected in those reported below for the method of lines problems considered in this paper.

The sparse algorithms used in the present experiments are structured in a fashion similar to the dense algorithms described above, but in which sparse data structures from the well-known SPARSPAK [6,10] are also employed. More details will be given elsewhere.

## 3. Parallel method of lines solutions

Generally, in order to minimize the reduction in performance of a parallel solution due to residual sequential code, it is desirable to do as much of the solution as possible in parallel. However, for ODE solvers, the bulk of the time is spent in Jacobian related calculations (i.e., approximating the Jacobian, factoring the iteration matrix, and solving the linear equations required by the corrector iteration). Reference [14] contains the relative solution times for the first problem described in the next section using the well-known LSODE solver [11]. Typically, 90–99% of the overall execution time for the solver is spent doing Jacobian related calculations. (For the two problems discussed in this paper, the percentages are 97% and 99%, respectively.) Hence, it is possible to obtain significant overall code speedups by doing the only Jacobian related calculations in parallel.

We chose the SDRIV3 ODE solver for the present tests. SDRIV3 contains a particularly attractive USERS option suggested by Mac Hyman of the Los Alamos National Laboratory. This option allows the user complete control over all aspects of the Jacobian related calculations. We chose to model the USERS routine after the corresponding routines which were developed for the LSODE solver in [14]. Consequently, no modifications whatsoever were required for SDRIV3. The present results suggest that the inclusion of such an option in other standard ODE solvers would be very worthwhile.

The first big ticket candidate for parallel solution is the linear algebra as discussed in the previous section. Indeed, the linear algebra is generally the most computationally intensive portion of the solution. The second candidate for parallel solution is the approximation of the Jacobian matrix. For many problems for which the derivative evaluation is very expensive (such as those discussed in [19]), the cost involved in approximating the Jacobian matrix can actually exceed the cost of the associated linear algebra. The standard way to approximate a dense Jacobian matrix is to perturb the solution one component at a time, calculate the derivative for each perturbed solution, and use finite differences to approximate the corresponding column of the Jacobian matrix. Since these calculations are independent, they can be done in parallel by dividing the columns among multiple processors. This strategy requires that each processor be given a copy of the coding required to calculate the system derivatives. To minimize memory contention in the present case, the Jacobian matrix-related code for each processor is given its own copy of the solution vector. (Standard solvers such as LSODE and SDRIV3 save the original solution and overwrite the solution vector with the perturbed solutions as necessary.) The derivatives for the perturbed solutions are calculated in the corresponding columns of the Jacobian matrix to avoid the necessity of providing additional storage for the derivative calculation. The resulting speedup for this portion of the solution is almost exactly linear in the number of processors regardless of the problem size; the speedup is also problem independent.

For sparse problems further savings are possible. The standard column grouping algorithm [7] can be used to reduce the number of derivative evaluations required to approximate the Jacobian. These evaluations can then be divided among the different processors. Of course, the resulting speedup is problem dependent. Results which demonstrate the possible savings for sparse Jacobians will be discussed in a later section.

For many realistic problems, the derivative evaluation generally is very expensive due to the use of water property calculations and other auxiliary calculations (see [19]). Although we do not consider the question in this paper, it is possible to reduce costs further by performing the actual derivative evaluation in parallel or, perhaps more appropriately, by vectorizing the derivative evaluation. This idea is particularly attractive in light of recent and coming extensions of the software available for performing basic linear algebra operations [1].

In the present experiments, we considered three types of solutions. First, solutions were considered in which the Jacobian is formed as a dense matrix and the subsequent linear algebra is performed in a dense fashion. This corresponds to the type of solution employed by most standard ODE solvers. The second solution considered is one in which the Jacobian is formed as a sparse matrix using the previously mentioned column grouping algorithm in parallel but the linear algebra is done in a dense fashion. Our interest in this second mixed approach is motivated by pragmatic observations regarding several of the problems discussed in [19]. Several of these problems have relatively sparse Jacobian matrices (typically, on the order of 10–25% nonzero elements) for which the column grouping technique requires roughly $N/10$ derivative evaluations but for which fill-in increases the overall sparse matrix storage requirements considerably. Such problems arise frequently and naturally when lumped parameter calculations are used in engineering models (e.g., a single pressure calculation is performed rather than a node wise calculation; this has the effect of introducing horizontal and vertical "feelers" in the Jacobian). The second problem discussed in the next section is similar to these problems which generally are not sparse enough to gain the full advantage of a sparse linear equation solver but are sparse enough to take advantage of special techniques for forming the Jacobian matrix. Interest in such problems suggests the attractiveness of a standard ODE solver option to form the Jacobian matrix as a sparse matrix but to then process it as a dense matrix. Results given in the next section illustrate the speedups possible with this approach for the present problems. Finally, we considered fully sparse solutions and performed some preliminary tests. (By fully sparse we mean that the Jacobian is approximated as a sparse matrix and the linear systems are solved using sparse techniques. The techniques used to obtain the present results only perform the matrix factorization in parallel and solve the subsequent linear equations serially. We will say more about this in a later section.) This solution strategy suggests several further things that can be done in this context particularly with respect to the manner in which the parallel linear algebra is performed. The resulting questions associated with fully sparse parallel method of lines solutions merit further attention. Therefore, only preliminary results are given in this paper.

It is important to note that the parallel solution techniques are not limited to method of lines problems and they are in fact applicable to the solution of any stiff system of ODEs. They can be used to improve the performance of any standard stiff ODE solver. In fact, use of the techniques described in this paper for implicit Runge–Kutta solvers is currently under investigation.

## 4. Description of the test problems

This section describes two problems which were used to obtain the results reported in this paper.

*Model problem 1*

The first problem is a mock-up of a fluid flow problem. This problem shares many of the characteristics associated with complex problems sometimes solved by automatic ODE solvers. (See [19] for descriptions of several such problems.) The problem is very stiff and has damped oscillatory solution components that must be tracked over long time intervals. It has been used previously to benchmark the performance of several ODE solvers [17,20] and to illustrate various pragmatic observations about the performance of sparse ODE solvers [18]. Another attractive feature of the problem is that it causes virtually all of the coding in a good adaptive solver such as SDRIV3 to be exercised and hence provides an indication of how such a solver will perform on the types of problems described in [19].

Consider the following formulation of the one-dimensional Euler equations:

$$\frac{\partial U}{\partial t} + A\frac{\partial U}{\partial z} = C, \quad 0 \leqslant t, \ 0 \leqslant z \leqslant L, \tag{4.1}$$

where

$$U = (\rho, G, T)^{\mathrm{T}}, \tag{4.2}$$

$$C = \left(0, \ -KG\left|\frac{G}{\rho}\right| - \rho g_a \sin\theta, \ \frac{a^2 \Phi P_H \kappa}{C_p A_f}\right)^{\mathrm{T}}, \tag{4.3}$$

$$A = \begin{pmatrix} 0 & 1 & 0 \\ \dfrac{1}{\rho\kappa} - \dfrac{G^2}{\rho^2} & 2\dfrac{G}{\rho} & \dfrac{\beta}{\kappa} \\ \dfrac{-a^2\beta\overline{T}G}{\rho^2 C_p} & \dfrac{a^2\beta\overline{T}}{\rho C_p} & \dfrac{G}{\rho} \end{pmatrix}, \tag{4.4}$$

and

$$a,\ \kappa,\ \beta,\ C_p = f(T,\ \rho), \quad \text{equation of state.} \tag{4.5}$$

(The results given in this paper were obtained using tabular values for the fluid properties in order to avoid the use of a water property package.)

The following boundary conditions will be used:

$$\rho(0,\ t) = \rho_0 = 795.521, \qquad T(0,\ t) = T_0 = 255.000, \qquad G(L,\ t) = G_0 = 270.900. \tag{4.6}$$

The test problem can be solved by substituting finite-difference approximations for the spatial ($z$) derivatives in (4.1) and integrating the resulting discretized system of ODEs with respect to time. A partition $z_1, \ldots, z_{M+1}$ with $z_i = L(i-1)/M$, $i = 1, \ldots, M = 1$, is first defined. After the spatial differences are defined at each spatial node $z_i$ and the boundary conditions are applied, there results a system containing $3M$ ODEs. For a given set of initial conditions, this system of ODEs may be integrated in time to obtain the desired solution. For the results reported in this paper, a linear rise was used for temperature. (Corresponding densities were obtained using these temperatures and a constant pressure. A constant value was used for the initial mass fluxes.)

The spatial discretization is performed using a pseudo-characteristics approach [5] as follows. The eigenvalues of $A$ are

$$\frac{G}{\rho}, \quad \frac{G}{\rho} + a \quad \text{and} \quad \frac{G}{\rho} - a. \tag{4.7}$$

In the usual method of characteristics solution, one would first reduce the equations to characteristic form by diagonalizing $A$. This requires finding a nonsingular matrix $B$ for which

$$BAB^{-1} = D, \tag{4.8}$$

where $D$ is a diagonal matrix whose diagonal elements are the above eigenvalues. One such matrix is

$$B = \begin{pmatrix} \beta a^2 \bar{T} & 0 & -\rho C_p \\ -G\kappa a + 1 & \rho\kappa a & \rho\beta \\ G\kappa a + 1 & -\rho\kappa a & \rho\beta \end{pmatrix}. \tag{4.9}$$

Multiplying the terms in (4.2) by this matrix gives the following characteristic form of the equations:

$$B\frac{\partial U}{\partial t} + DB\frac{\partial U}{\partial z} = BC. \tag{4.10}$$

The idea behind the pseudo-characteristic solution for (4.10) is as follows. At each spatial node, one-sided difference approximations are calculated for the spatial derivatives:

$$\begin{array}{lll} \rho_{z,0}, & G_{z,0}, & T_{z,0}, \\ \rho_{z,+}, & G_{z,+}, & T_{z,+}, \\ \rho_{z,-}, & G_{z,-}, & T_{z,-}. \end{array}$$

The subscript $z$ denotes partial differentiation with respect to $z$. The subscripts 0, + and − indicate the one-sided differences are computed with the direction of the differencing dictated by the sign of the local characteristics $G/\rho$, $G/\rho + a$, $G/\rho - a$, respectively. For each local characteristic, backward differences are used if the characteristic is positive; otherwise forward differences are used (hence the terminology pseudo-characteristic method).

When the resulting values are substituted into the characteristic equation (4.10), there results a linear system of three equations in the three unknowns

$$\frac{\mathrm{d}\rho}{\mathrm{d}t}, \quad \frac{\mathrm{d}G}{\mathrm{d}t} \quad \text{and} \quad \frac{\mathrm{d}T}{\mathrm{d}t}$$

at each node. At node $z_i$ the $3 \times 3$ system of linear equations to be solved is:

$$B\left(\frac{\mathrm{d}\rho_i}{\mathrm{d}t}, \frac{\mathrm{d}G_i}{\mathrm{d}t}, \frac{\mathrm{d}T_i}{\mathrm{d}t}\right)^{\mathrm{T}} = E, \tag{4.11}$$

where $B$ in (4.9) is evaluated at $z_i$ using $\rho_i$, $G_i$ and $T_i$, and the vector $E$ is defined in the following manner using $C$ in (4.3) and the spatial differences:

$$\begin{aligned}
E_1 &= \sum_{j=1}^{3} B_{1j}C_j - \left(\frac{G_i}{\rho_i}\right)\{B_{11}\rho_{z,0} + B_{12}G_{z,0} + B_{13}T_{z,0}\},\\
E_2 &= \sum_{j=1}^{3} B_{2j}C_j - \left(\frac{G_i}{\rho_i} + a_i\right)\{B_{21}\rho_{z,+} + B_{22}G_{z,+} + B_{23}T_{z,+}\},\\
E_3 &= \sum_{j=1}^{3} B_{3j}C_j - \left(\frac{G_i}{\rho_i} - a_i\right)\{B_{31}\rho_{z,-} + B_{32}G_{z,-} + B_{33}T_{z,-}\}.
\end{aligned} \tag{4.12}$$

The solution is $(\mathrm{d}\rho_i/\mathrm{d}t, \mathrm{d}G_i/\mathrm{d}t, \mathrm{d}T_i/\mathrm{d}t)^{\mathrm{T}}$ which defines the system time derivatives. Direct calculation shows that the solution of (4.11) is given by:

$$\frac{\mathrm{d}T_i}{\mathrm{d}t} = \frac{\frac{1}{2}(E_2 + E_3)a^2\beta\bar{T}_i - E_1}{\rho_i\left(c_p + a^2\beta\bar{T}_i\right)}, \tag{4.13}$$

$$\frac{\mathrm{d}\rho_i}{\mathrm{d}t} = \tfrac{1}{2}(E_2 + E_3) - \rho_i\beta\frac{\mathrm{d}T_i}{\mathrm{d}t}, \tag{4.14}$$

$$\frac{\mathrm{d}G_i}{\mathrm{d}t} = -\frac{E_3 - (G_i\kappa a + 1)\frac{\mathrm{d}\rho_i}{\mathrm{d}t} - \rho_i\beta\frac{\mathrm{d}T_i}{\mathrm{d}t}}{(\rho_i\kappa a)}. \tag{4.15}$$

*Model problem 2*

The physical system to be analyzed, illustrated in Fig. 1, has two basic components, air and water. The physical phenomenon (humidification) described by the PDE model is well-known. The mathematical model for the system is a system of three nonlinear, one-dimensional, initial-value PDEs, which are well suited for MOL solution. The resulting system of MOL

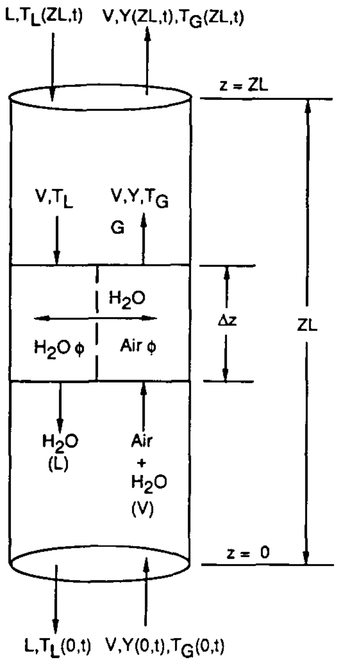

Fig. 1. Humidification column.

ODEs has interesting properties as reflected in the map and eigenvalues of the Jacobian matrix.

The humidification column of Fig. 1 is packed with a porous medium which permits the flow of water down through the column by gravity, and the flow of air up due to an imposed pressure at the bottom of the column (for example, from a centrifugal blower). The packing in the column promotes contact between the water and air, and therefore enhances the exchange of mass and energy between the two streams (we will neglect the exchange of momentum, although this is manifest as a drop of the air pressure along the column).

To fully analyze the operation of the column, we require the calculation of the following dependent variables (see also Table 1).

(1) The air humidity $y(z, t)$ as a function of position along the column $z$ and time $t$. We define the humidity as the ratio of the moles of water to the moles of dry air.

(2) The air temperature $TG(z, t)$.

(3) The water temperature $TL(z, t)$.

The humidity $y$ is computed from a mass balance for the air written on a section of the column of length $\Delta z$, followed by $\Delta z \to 0$. The final result is

$$y_t = -\frac{V}{GS} y_z + \frac{k_y a_v}{G}(y_s - y). \tag{4.16}$$

A subscript with respect to an independent variable ($z$ or $t$) denotes a partial derivative with respect to that variable. Thus, the combination of the $t$ derivative on the left-hand side of (4.16) and the $z$ derivative in the first right-hand side term is the advection group, representing accumulation of water in the air and convection of water in the air within a differential volume

Table 1
Nomenclature for test problem 2

| Symbol | Property | SI metric unit |
|---|---|---|
| *TG* | Gas temperature | °C |
| *EV* | Internal energy of the gas stream | cal/gm mol |
| *EP* | Enthalpy of the gas stream | cal/gm mol |
| *TL* | Liquid temperature | °C |
| *Y* | Mole ratio of $H_2O$ in the gas | gm mol $H_2O$/gm mol dry air |
| *YS* | Mole ratio of $H_2O$ in the gas which would be in equilibrium with the liquid at temperature *TL* | gm mol $H_2O$/gm mol dry air |
| *T* | Time | |
| *Z* | Axial position along the column | cm |
| *ZL* | Length of the column | cm |
| *V* | Dry air molar flow rate | cm/hr |
| *G* | Dry air molar holdup | gm mols/$cm^3$ |
| *S* | Column cross sectional area | $cm^2$ |
| *KY* | Mass transfer coefficient | gm mols dry air/(hr-$cm^3$) |
| *AV* | Heat and mass transfer areas per unit volume of column | cal/(hr-$cm^3$- °C) |
| *CVV* | Specific heat of water vapor at constant volume | cal/(gm mol- °C) |
| *CPV* | Specific heat of water vapor at constant pressure | cal/(gm mol- °C) |
| *CVA* | Specific heat of dry air at constant volume | cal/(gm mol- °C) |
| *CPA* | Specific heat of dry air at constant pressure | cal/(gm mol- °C) |
| *CL* | Specific heat of water | cal/(gm mol- °C) |
| *L* | Liquid ($H_2O$) molar flow rate | gm mols/hr |
| *H* | Liquid ($H_2O$) molar holdup | gm mols/$cm^3$ |
| *DHVAP* | Heat of vaporization of water | cal/gm mol |
| *X* | Control value stem position | |
| *XSS* | *X* at steady state | |
| *KC* | Controller gain | 1/ °C |
| *TI* | Controller integral time | hr |
| *CVDP* | Product of the control valve constant and square root of the pressure drop across the valve | gm mols/hr |
| *TLSET* | Controller set point | °C |

of the column, respectively. The air velocity $V/(GS)$ is positive since the air flows up from the bottom of the column (in the positive $z$ direction).

The second right-hand side term of (4.16) represents the mass transfer of water to or from the air due to the humidity difference between the air and water $y_s - y$. When $y_s - y$ is negative (the air humidity exceeds the saturation humidity), condensation of water from the air takes place; if $y_s - y$ is positive, evaporation of water into the air (humidification) takes place.

The temperature of the air is computed by writing an energy balance for the air in the differential section of the column

$$EV_t = -\frac{V}{GS}EP_z + \frac{ha_v}{G}(TL - TG) + \frac{k_y a_v}{C_L H}(y_s - y)(C_{vv}TG + \Delta H_{\text{vap}}). \qquad (4.17)$$

The left-hand side and first right-hand side terms of (4.17) are again the advection group for the air (the air velocity $V/(GS)$ is positive since the air enters the bottom of the column and flows in the positive $z$ direction). However, there is one difference in this advection group we did not observe in (4.16); the dependent variable in the time derivative is the air internal energy $EV$ while the dependent variable in the spatial derivative is the air enthalpy $EP$.

The second right-hand side term of (4.17) is the heat transfer rate between the water and air due to the temperature difference $TL - TG$. The third right-hand side term of (4.17) represents the gain or loss of energy by the air due to mass transfer of water to the air (evaporation) or to the water (condensation); this transfer occurs as a result of differences between the air humidity $y$ and the saturation humidity $y_s$ as in (4.16). Note also that when (4.17) is integrated in time, the computed dependent variable is the air internal energy $EV$. This must then be converted to the corresponding air temperature $TG$ for use in the heat transfer term of (4.17).

Finally, the water temperature $TL(z, t)$ is computed by writing an energy balance on the water in the differential section of the column

$$TL_t = \frac{L}{HS} TL_z - \frac{ha_v}{C_L H}(TL - TG) - \frac{k_y a_v}{C_L H}(y_s - y)(C_{vv} TG + \Delta H_{\text{vap}}). \qquad (4.18)$$

The advection group for the liquid temperature is apparent. The second right-hand side term of (4.18) is the heat transfer rate between the water and air. The third right-hand side term of (4.18) represents the gain or loss of energy by the water due to mass transfer of water from the air to the water (condensation) or to the air from the water (evaporation); this transfer occurs as a result of differences between the air humidity $y$ and the saturation humidity $y_s$ as in (4.16) and (4.17).

The initial conditions for (4.16)–(4.18) are

$$\begin{aligned} &y(z, 0) = 0.01 \text{ gm moles water/gm mole dry air},\\ &EV(z, 0) = C_{va} TG(z, 0) + y(z, 0)\big(C_{vv} TG(z, 0) + \Delta H_{\text{vap}}\big)\\ &\qquad = (5.3)(43.33) + 0.01((5.3)(43.33) + 9443.6) \text{ cal/gm mole air}, \qquad (4.19)\\ &TL(z, 0) = 43.33\,^\circ\text{C}. \end{aligned}$$

The boundary conditions for (4.16)–(4.18) are

$$\begin{aligned} &y(0, t) = 0.01 \text{ gm moles water/gm mole dry air},\\ &EP(0, t) = C_{pa} TG(0, t) + y(0, t)\big(C_{pv} TG(0, t)\ \Delta H_{\text{vap}}\big)\\ &\qquad = (5.3)(43.33) + 0.01((5.3)(43.33) + 9443.6) \text{ cal/gm mole air}, \qquad (4.20)\\ &TL(z_l, t) = 43.33\,^\circ\text{C}. \end{aligned}$$

These boundary conditions reflect the entering air humidity $y(0, t)$, the entering air enthalpy $EP(0, t)$ and the entering liquid temperature $TL(z_l, t)$, where $z_l$ is the column length.

If $M$ denotes the number of spatial nodes, the resulting ODE system for this problem contains $3M + 1$ equations. The Jacobian matrix for this system has an interesting structure which changes with time. Figure 2 depicts the structure of the Jacobian for $M = 11$ and $t = 0.5$. (The numbers given in Fig. 2 represent the relative magnitudes of the corresponding elements in the Jacobian.) The Jacobian is nonbanded and in fact has a structure similar to several of the problems discussed in [19]. The system is also relatively stiff. For example, if $M = 11$ and $t = 0.5$, the nonzero eigenvalues range from about $-1.5$ to $-4286$. Reference [16] contains a detailed discussion of this problem.

## 5. Discussion of test results

All tests described in this paper were performed at the Oak Ridge National Laboratory using double-precision arithmetic on the Sequent Computer Systems Balance 8000 Parallel Processor, hereafter referred to as the B8000. (Some of the tests were also duplicated on a similarly configured system at the National Institute of Standards.) The B8000 system consists of 12 CPUs (NSC 32032 processors) and 16 million bytes of shared memory (i.e., memory shared by all processors). Each of the processors is functionality equivalent and provides performance approximately equal to that of a VAX 11/750. The system bus has a sustained data transfer rate of 26.7 megabytes per second. The B8000 operates under a UNIX operating system. For parallel applications, the available multitasking primitives are similar to those for other shared-memory computers such as the CRAY X-MP/4. Readers not familiar with the use of such computers are referred to [2] which contains a discussion of parallel programming for shared-memory parallel computers.

In this paper, efficiency is defined in the usual manner as:

$$\text{efficiency} = \frac{100}{P}\,\frac{T(1)}{T(P)}\,\%,$$

ODE Dependent Variables

ODE Temporal Derivatives

```
               1111111111222222222233333
      1234567890123456789012345678901234
 1
 2     8887                 55         6
 3     8787                 5 5        6
 4     8887                 4  5       5
 5     78887                4   5      5
 6      78887               4    5     5
 7       78887              4     5    5
 8        78887             4      5   5
 9         78887            5       5  6
10          78887           5        5 6
11          78898           5         66
12
13     9999       8887      89         9
14     9999       8787      8 9        9
15     9999       8888      8  9       9
16     99999      78888     8   9      9
17      99999      78888    8    9     9
18       99999      78888   8     9    9
19        99999      78888  9      9   9
20         99999      78888 9       9  9
21          99999      788889        9 9
22          99999      889989         99
23                          77776
24     8          5         67766
25      8          5         67766
26       8          5         67766
27        8          5         67766
28         8          5         67766
29          8          4         67766
30           8          4         6776
31            8          4        6666
32            , 8          4       6676
33
34                          4
```

Fig. 2. Jacobian matrix map for problem 2.

Table 2
Results for problem 1 (300 ODEs, dense-dense solution)

| Number of processors | Total execution time | Jacobian formation time and speedup | Linear algebra time and speedup | Total speedup achieved |
|---|---|---|---|---|
| 1 | $0.101 \cdot 10^5$ | $0.326 \cdot 10^4$ (1.0) | $0.655 \cdot 10^4$ (1.0) | 1.0 |
| 2 | $0.519 \cdot 10^4$ | $0.163 \cdot 10^4$ (2.0) | $0.333 \cdot 10^4$ (2.0) | 1.9 |
| 3 | $0.355 \cdot 10^4$ | $0.109 \cdot 10^4$ (3.0) | $0.224 \cdot 10^4$ (2.9) | 2.9 |
| 4 | $0.273 \cdot 10^4$ | $0.820 \cdot 10^3$ (4.0) | $0.169 \cdot 10^4$ (3.9) | 3.7 |
| 5 | $0.226 \cdot 10^4$ | $0.657 \cdot 10^3$ (5.0) | $0.137 \cdot 10^4$ (4.8) | 4.5 |
| 6 | $0.194 \cdot 10^4$ | $0.547 \cdot 10^3$ (6.0) | $0.116 \cdot 10^4$ (5.6) | 5.2 |
| 7 | $0.171 \cdot 10^4$ | $0.471 \cdot 10^3$ (6.9) | $0.101 \cdot 10^4$ (6.5) | 5.9 |
| 8 | $0.153 \cdot 10^4$ | $0.412 \cdot 10^3$ (7.9) | $0.887 \cdot 10^3$ (7.4) | 6.6 |
| 9 | $0.141 \cdot 10^4$ | $0.368 \cdot 10^3$ (8.9) | $0.809 \cdot 10^3$ (8.1) | 7.2 |
| 10 | $0.129 \cdot 10^4$ | $0.329 \cdot 10^3$ (9.9) | $0.720 \cdot 10^3$ (9.1) | 7.8 |

where $P$ is the number of processors and $T(P)$ is the solution time for the $P$-processor solution. By speedup we mean the quantity $T(1)/T(P)$. An efficiency of 100% therefore corresponds to a perfect speedup, that is, a reduction in execution time by a factor of $1/P$. The value of $T(1)$ used to compute the efficiencies is the execution time for the parallel solution with one processor rather than the execution time for the corresponding sequential solution. Since additional overhead is involved with the parallel 1-processor solution, this results in slightly higher efficiencies being reported in the tables. However, these two times are virtually identical. (Extensive testing of the parallel software and comparisons with standard LINPACK [8] serial solution software for problems of different size indicate differences in the solutions on the order of 0–3% are incurred. Further details are available from the second author.) Therefore, this does not have a significant impact on the results.

Table 2 contains the execution times and speedups for the solution of the first problem with 300 ODEs and the dense Jacobian formation, dense linear algebra solution strategy for various numbers of processors. The results of most interest are the overall speedups (given in the last column). For 10 processors, an efficiency of 78% was obtained; the raw speedup was by a factor of almost 8. For 600 ODEs the efficiency increases to 90% for 10 processors. Since both the Jacobian and the linear algebra are done in dense mode, these speedups are problem independent. Thus, similar speedups can be expected for other problems. This is illustrated by the results for the second test problem. Table 3 contains the results for the solution of this problem. Use of 10 processors yielded an efficiency of 83%. For 598 ODEs the efficiency increases to 89% for 10 processors. For smaller ODE systems, the corresponding results follow patterns similar to those given in [14] for the LSODE solution.

Table 4 contains results for the first problem using the various sparse techniques discussed earlier. Table 5 contains similar results for the second problem. In each case the speedup reported is relative to the dense Jacobian, dense linear algebra solution using 1 processor. As mentioned earlier, the results for the sparse solutions are problem dependent. This is due to the fact that the column grouping algorithm used to approximate a sparse Jacobian requires 8 and 15 derivative evaluations, respectively, for problems 1 and 2, instead of $N$ evaluations

Table 3
Results for problem 2 (298 ODEs, dense-dense solution)

| Number of processors | Total execution time | Jacobian formation time and speedup | Linear algebra time and speedup | Total speedup achieved |
|---|---|---|---|---|
| 1 | $0.258 \cdot 10^4$ | $0.728 \cdot 10^3$ (1.0) | $0.183 \cdot 10^4$ (1.0) | 1.0 |
| 2 | $0.133 \cdot 10^4$ | $0.370 \cdot 10^3$ (2.0) | $0.932 \cdot 10^3$ (2.0) | 1.9 |
| 3 | $0.898 \cdot 10^3$ | $0.246 \cdot 10^3$ (3.0) | $0.626 \cdot 10^3$ (2.9) | 2.9 |
| 4 | $0.691 \cdot 10^3$ | $0.185 \cdot 10^3$ (3.9) | $0.478 \cdot 10^3$ (3.8) | 3.7 |
| 5 | $0.565 \cdot 10^3$ | $0.149 \cdot 10^3$ (4.9) | $0.389 \cdot 10^3$ (4.7) | 4.6 |
| 6 | $0.478 \cdot 10^3$ | $0.124 \cdot 10^3$ (5.9) | $0.327 \cdot 10^3$ (5.6) | 5.4 |
| 7 | $0.420 \cdot 10^3$ | $0.107 \cdot 10^3$ (6.8) | $0.285 \cdot 10^3$ (6.4) | 6.1 |
| 8 | $0.375 \cdot 10^3$ | $0.939 \cdot 10^2$ (7.8) | $0.253 \cdot 10^3$ (7.2) | 6.9 |
| 9 | $0.341 \cdot 10^3$ | $0.837 \cdot 10^2$ (8.7) | $0.229 \cdot 10^3$ (8.0) | 7.6 |
| 10 | $0.310 \cdot 10^3$ | $0.753 \cdot 10^2$ (9.7) | $0.206 \cdot 10^3$ (8.9) | 8.3 |

where $N$ is the number of equations in the system. For problem 1, this accounts for the speedups greater than the number of processors.

For problem 2, the speedups given in Table 5 are less dramatic. In fact, the sparse Jacobian, sparse linear algebra solution is slower than the sparse Jacobian, dense linear algebra solution for this problem. This is due to the fact that we are not actually performing a fully sparse solution. Because of the difficulty of doing forward solves in parallel (a result of the sparse matrix data structure employed), the sparse software used in the present experiments only performs the matrix factorization in parallel. The solution of the subsequent linear equations is done serially. For some problems like the second one, the serial solution of the linear equations

Table 4
Results for problem 1 (300 ODEs)

| Solution option | Total solution time | Speedup |
|---|---|---|
| Dense Jacobian<br>Dense linear algebra<br>1 processor | $0.299 \cdot 10^4$ | 1.0 |
| Dense Jacobian<br>Dense linear algebra<br>8 processors | $0.441 \cdot 10^3$ | 6.8 |
| Sparse Jacobian<br>Dense linear algebra<br>8 processor | $0.337 \cdot 10^3$ | 8.9 |
| Sparse Jacobian<br>Sparse linear algebra<br>8 processors | $0.897 \cdot 10^2$ | 33.3 |

Table 5
Results for problem 2 (298 ODEs)

| Solution option | Total solution time | Speedup |
|---|---|---|
| Dense Jacobian<br>Dense linear algebra<br>1 processor | $0.260 \cdot 10^4$ | 1.0 |
| Dense Jacobian<br>Dense linear algebra<br>8 processors | $0.376 \cdot 10^3$ | 6.9 |
| Sparse Jacobian<br>Dense linear algebra<br>8 processors | $0.301 \cdot 10^3$ | 8.6 |
| Sparse Jacobian<br>Sparse linear algebra<br>8 processors | $0.504 \cdot 10^3$ | 5.2 |

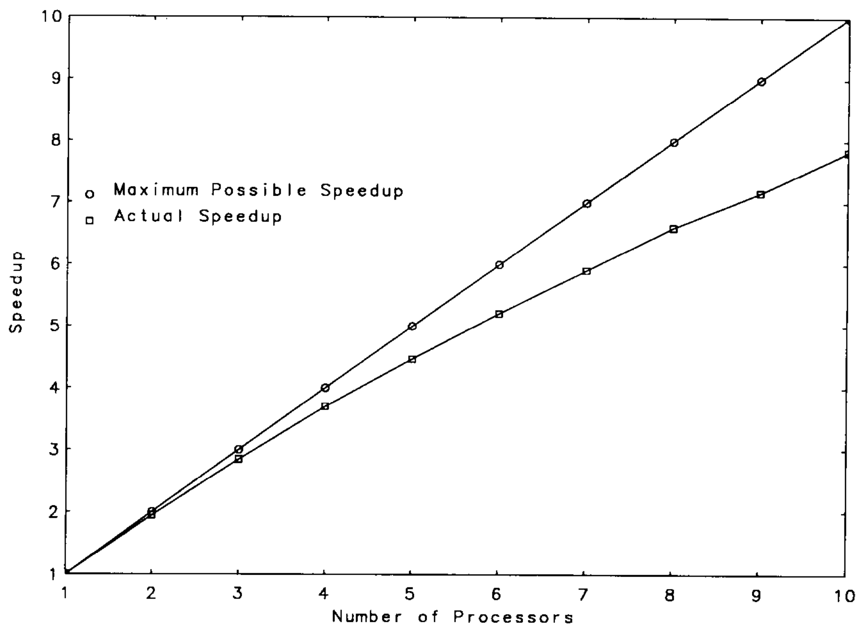

Fig. 3. Overall speedups for problem 1.

is actually more expensive than the parallel matrix factorization. We are currently investigating fully sparse solution techniques in which the linear equation solving is also done in parallel. Results will be reported elsewhere. The preliminary results in Tables 4 and 5 indicate that substantial speedups are possible. See Figs. 3 and 4.

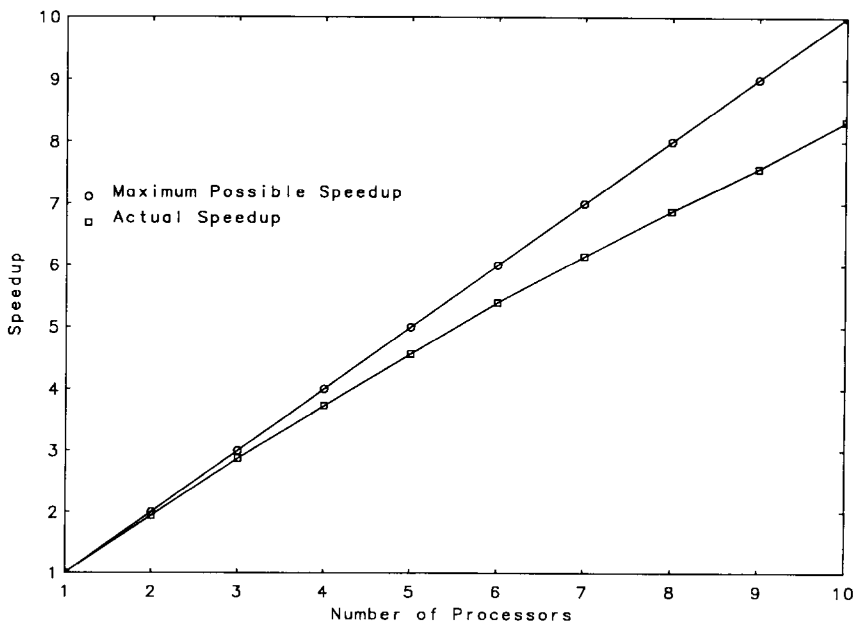

Fig. 4. Overall speedups for problem 2.

In the present studies we considered the question of parallel MOL solutions only on shared-memory parallel computers. Work is currently underway to implement the various parallel solution strategies on distributed-memory computers. Results will be reported elsewhere.

## 6. Summary and conclusions

The solution of method of lines problems using ordinary differential equation solvers on a shared-memory parallel computer was discussed in this paper. The ideas involved were illustrated by applying them to two model problems obtained by using the method of pseudo-characteristics to discretize spatially the one-dimensional Euler equations in one case and by applying upwind differences to approximate the cooling of a humidification column in the second case. Parallel solution techniques were incorporated in the well-known SDRIV3 solver and results were given to demonstrate the speedups obtained by applying the parallel version of the model problems. The results indicate that a modern stiff ODE solver is ideally suited for parallel processing on shared-memory computers. For moderately sized systems, efficiencies that are nearly linear in the number of processors are possible. The results also indicate that the performance of modern ODE solvers can be enhanced through the inclusion of options such as the USERS option in SDRIV3, and the ability to perform mixed dense-sparse solutions.

Experiments were performed using three solution strategies: dense Jacobian formation and dense linear algebra, sparse Jacobian formation and dense linear algebra, and fully sparse solutions. The results suggest significant speedups are possible for ODE solvers in this context. The speedups are essentially problem independent for the dense-dense solution strategy. For the sparse-dense solution strategy, the Jacobian formation speedup is problem dependent while the linear algebra speedup is problem independent. More detailed results will be presented elsewhere for the fully sparse solution strategy.

## Appendix

We present here a brief introduction to the method of lines. Consider the one-dimensional heat conduction equation in Cartesian coordinates (Fourier's second law):

$$\frac{\partial T}{\partial t} = \frac{k}{\rho C_p} \frac{\partial^2 T}{\partial x^2}. \tag{A.1}$$

Equation (A.1) can be written in subscript notation (with the thermal diffusivity $k/(\rho C_p) = 1$):

$$T_t = T_{xx}. \tag{A.2}$$

Equation (A.2) is first-order in $t$ and second-order in $x$, so it requires one initial condition and two boundary conditions:

$$T(x, 0) = \sin\left(\frac{\pi x}{L}\right), \tag{A.3}$$

$$T(0, t) = 0, \tag{A.4}$$

$$T(L, t) = 0. \tag{A.5}$$

If we consider how we might proceed to produce a computer solution of equations (A.2)–(A.5), we quickly realize there is no direct way to tell the computer about a problem in PDEs (working, for example, with a standard compiler like FORTRAN); computers do not naturally understand partial derivatives. Rather, we must state the problem in PDEs in a format which can then be programmed using a standard compiler; basically, this means replacing the original PDE problem with an equivalent problem in algebra.

For example, if we seek $T(x, t)$ which satisfies equations (A.2)–(A.5), we might consider the variation of $T$ with respect to $x$ to take place along a grid in $x$ where a particular value of $x$ will be specified in terms of an integer index $i$. Thus, $x(1)$ corresponds to $x = 0$ and $x(N)$ to $x = L$, where $N$ is the total number of grid points in $x$. A particular value of $x$ is then given by $x(i) = (i-1)\,\Delta x$, $i = 1, 2, \ldots, N$, where $\Delta x$ is the grid spacing; i.e., $\Delta x = L/(N-1)$. Similarly, the variation in $T$ with respect to $t$ could be specified in terms of an integer index $j$, so $t(1)$ corresponds to $t = 0$ and $t(j) = (j-1)\,\Delta t$, $j = 1, 2, \ldots,$ corresponds to a grid spacing in $t$ of $\Delta t$.

With these two subscripts $i$ and $j$ we could specify a value of $T$ corresponding to particular values of $x$ and $t$, $T(i, j) = T((i-1)\,\Delta x, (j-1)\,\Delta t)$. Then we could replace the partial derivatives in (A.2) with algebraic approximations evaluated at a general point with indices $(i, j)$; this would lead to a set of algebraic equations which approximate (A.2). Once the approximating algebraic equations have been defined, they could be solved using any standard linear equation solver [13] to obtain an approximate numerical solution to (A.2); of course, the auxiliary conditions (A.3)–(A.5) would also have to be included in the algebraic equations. This procedure is the basis for well-known classical finite-difference, finite-element and finite-volume methods for PDEs. The *method of lines* (MOL) is really just a small departure from this basic approach.

In the MOL, we retain the index $i$ to account for variations of $T$ with $x$, but we treat $t$ as a continuous variable (rather than $t$ evaluated at discrete points corresponding to the index $j$). Thus, we will replace the partial derivative $T_{xx}$ in (A.2) with an algebraic approximation evaluated at point $i$, but keep the derivative $T_t$; this will lead to a system of differential equations in $t$, and since we now have only one independent variable $t$, the differential equations will be ODEs.

Since one example is probably worth a thousand words, we now consider an MOL formulation of (A.2)–(A.5):

$$\frac{\mathrm{d}T_i}{\mathrm{d}t} = \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}, \quad i = 2, 3, \ldots, N-1, \tag{A.6}$$

where we have used the well-known three-point, second-order, centered finite-difference approximation for the partial derivative $T_{xx}$ in (A.2).

Initial condition (A.4) is

$$T_i = \sin\left(\frac{(i-1)\,\Delta x\,\pi}{L}\right), \quad i = 2, 3, \ldots, N-1. \tag{A.7}$$

Boundary conditions (A.4) and (A.5) are simply

$$T_1 = 0, \tag{A.8}$$

$$T_N = 0. \tag{A.9}$$

Thus, we have a system of $N-2$ ODEs in the $N-2$ unknowns $T_2, T_3, \ldots, T_{N-1}$, which can be numerically integrated by an initial-value ODE solver (once the number of grid points is selected).

Equation (A.2) is, of course, particularly simple. The MOL can be extended to quite general systems of nonlinear PDEs in one, two and three dimensions. This generalization is illustrated by the following system of PDEs:

$$\begin{aligned} u_{1t} &= f_1\Big(\bar{x}, t, u_1, u_2, \ldots, u_n, u_{1\bar{x}}, u_{2\bar{x}}, \ldots, u_{n\bar{x}}, u_{1\bar{x}\bar{x}}, u_{2\bar{x}\bar{x}}, \ldots, u_{n\bar{x}\bar{x}}, \ldots\Big), \\ u_{2t} &= f_2\Big(\bar{x}, t, u_1, u_2, \ldots, u_n, u_{1\bar{x}}, u_{2\bar{x}}, \ldots, u_{n\bar{x}}, u_{1\bar{x}\bar{x}}, u_{2\bar{x}\bar{x}}, \ldots, u_{n\bar{x}\bar{x}}, \ldots\Big), \\ &\vdots \\ u_{nt} &= f_n\Big(\bar{x}, t, u_1, u_2, \ldots, u_n, u_{1\bar{x}}, u_{2\bar{x}}, \ldots, u_{n\bar{x}}, u_{1\bar{x}\bar{x}}, u_{2\bar{x}\bar{x}}, \ldots, u_{n\bar{x}\bar{x}}, \ldots\Big), \end{aligned} \tag{A.10}$$

where $u_1, u_2, \ldots, u_n$ is the vector of dependent variables of length $n$ to be computed by the MOL, $t$ is the initial-value independent variable, typically time, $f_1, f_2, \ldots, f_n$ is the vector of right-hand side functions defined for a particular PDE problem, and $\bar{x}$ is the vector of boundary-value (spatial) independent variables, e.g., $[x, y, z]$ for Cartesian coordinates, $[r, \theta, z]$ for cylindrical coordinates, $[r, \theta, \phi]$ for spherical coordinates.

Note that in accordance with the usual practice in the numerical analysis literature, we denote the dependent variable as $u$ (rather than $T$). As usual, a subscript with respect to $t$ or $\bar{x}$ indicates a partial derivative with respect to $t$ or $\bar{x}$. Also, we have departed from the usual notation for a vector, e.g., $u_1, u_2, \ldots, u_n$ is used in place of $[u_1, u_2, \ldots, u_n]^{\mathrm{T}}$, where the superscript T denotes a vector transpose. An overbar is also used to denote a vector, e.g., $\bar{x} = x, y, z$ or $x_1, x_2, x_3$ (or $[x, y, z]^{\mathrm{T}}$, $[x_1, x_2, x_3]^{\mathrm{T}}$).

Note that the vector of right-hand side functions of (A.10), $f_1, f_2, \ldots, f_n$, contains first-order derivatives in $\bar{x}$, $u_{1\bar{x}}, u_{2\bar{x}}, \ldots, u_{n\bar{x}}$, second-order derivatives in $\bar{x}$, $u_{1\bar{x}\bar{x}}, u_{2\bar{x}\bar{x}}, \ldots, u_{n\bar{x}\bar{x}}$, and suggests third- and higher-order derivatives in $\bar{x}$ with the "and so forth" notation, ...). Thus, (A.10) is quite general since we have not placed any restrictions on the form of the right-hand side functions, $f_1, f_2, \ldots, f_n$, or the maximum order of the spatial derivatives in $\bar{x}$. Equation (A.10) is limited to first-order derivatives in $t$, but this is really not a restriction since a PDE $n$th-order in $t$ can easily be written as a system of $n$ PDEs first-order in $t$.

Equation (A.10) also requires an initial condition vector

$$\begin{aligned} u_1(\bar{x}, t_0) &= g_1(\bar{x}), \\ u_2(\bar{x}, t_0) &= g_2(\bar{x}), \\ &\vdots \\ u_n(\bar{x}, t_0) &= g_n(\bar{x}), \end{aligned} \tag{A.11}$$

and a vector of boundary conditions

$$\begin{aligned}
&h_1(\bar{x}_b, t, u_1(\bar{x}_b, t), u_2(\bar{x}_b, t),\dots,u_n(\bar{x}_b, t), u_{1\bar{x}}(\bar{x}_b, t),\\
&\quad u_{2\bar{x}}(\bar{x}_b, t),\dots,u_{n\bar{x}}(\bar{x}_b, t),\dots) = 0,\\
&h_2(\bar{x}_b, t, u_1(\bar{x}_b, t), u_2(\bar{x}_b, t),\dots,u_n(\bar{x}_b, t), u_{1\bar{x}}(\bar{x}_b, t),\\
&\quad u_{2\bar{x}}(\bar{x}_b, t),\dots,u_{n\bar{x}}(\bar{x}_b, t),\dots) = 0,\\
&\vdots
\end{aligned} \tag{A.12}$$

where $t_0$ is the initial value of $t$, $g_1, g_2,\dots,g_n$ is the vector of initial condition functions, $h_1$, $h_2,\dots$ is the vector of boundary condition functions, and $\bar{x}_b$ are the boundary values of $\bar{x}$.

The length of the boundary condition vector $h_1, h_2,\dots$ cannot be stated generally for (A.12), since it will depend on the number and order of the spatial derivatives in (A.10). Also, $\bar{x}_b$, which generally denotes the boundary values of $\bar{x}$, cannot be stated more explicitly, since it will depend on the number of boundary value independent variables in (A.10) and (A.11) (typically, one, two or three for each PDE).

Equations (A.10)–(A.12) can be stated in a more concise vector form as

$$\bar{u}_t = \bar{f}\left(\bar{x}, t, \bar{u}, \bar{u}_{\bar{x}}, \bar{u}_{\bar{x}\bar{x}},\dots\right), \tag{A.13}$$

$$\bar{u}(\bar{x}, t_0) = \bar{g}(\bar{x}), \tag{A.14}$$

$$\bar{h}(\bar{x}_b, t, \bar{u}(\bar{x}_b, t), \bar{u}_{\bar{x}}(\bar{x}_b, t),\dots) = \bar{0}. \tag{A.15}$$

In general, to use the MOL, one must specify

$$\bar{f}\left(\bar{x}, t, \bar{u}, \bar{u}_{\bar{x}}, \bar{u}_{\bar{x}\bar{x}},\dots\right), \quad \bar{g}(\bar{x}) \quad \text{and} \quad \bar{h}(\bar{x}_b, t, \bar{u}(\bar{x}_b, t), \bar{u}_{\bar{x}}(\bar{x}_b, t),\dots).$$

## References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, LAPACK: a portable linear algebra library for high-performance computers, in: *Proceedings of Supercomputing '90* (IEEE Press, New York, 1990) 1–10.

[2] Balance 8000 guide to parallel programming, 1003-41030 Rev. A, Sequent Computer Systems, Inc., November 1985.

[3] P.N. Brown and A.C. Hindmarsh, Reduced storage matrix methods in stiff ODE systems, *J. Appl. Math. Comput.* **31** (1989) 40–91.

[4] G.D. Byrne and A.C. Hindmarsh, Stiff ODE solvers: a review of current and coming attractions, *J. Comput. Phys.* **70** (1987) 1–62.

[5] M.B. Carver, Pseudo-characteristic method of lines solution of first order hyperbolic equation systems, in: *Advances in Computer Methods for Partial Differential Equations III, Proc. 3rd IMACS Internat. Symp. on Computer Methods for Partial Differential Equations*, Bethlehem, PA (IMACS, New Brunswick, NJ, 1979) 227–230.

[6] E.C.H. Chu, J.A. George, J.W.-H. Liu and E.G.-Y. Ng, User's guide for SPARSPAK-A: Waterloo sparse linear equations package, Technical Report CS-84-36, Dept. Comput. Sci., Univ. Waterloo, Waterloo, Ontario, 1984.

[7] A.R. Curtis, M.J.D. Powell and J.K. Reid, On the estimation of sparse Jacobian matrices, *J. Inst. Math. Appl.* **13** (1974) 117–119.

[8] J.J. Dongarra, C.B. Moler, J.R. Bunch and G.W. Stewart, *LINPACK Users' Guide* (SIAM, Philadelphia, PA, 1979).

[9] J.A. George, M.T. Heath and J.W.-H. Liu, Parallel Cholesky factorization on a shared-memory multiprocessor, *Linear Algebra Appl.* **77** (1986) 165–187.

[10] J.A. George and J.W.-H. Liu, The design of a user interface for a sparse matrix package, *ACM Trans. Math. Software* **5** (1979) 134–162.

[11] A.C. Hindmarsh, Toward a systematized collection of ODE solvers, in: *Proceedings, Vol. 1, 10th IMACS Congress on System Simulation and Scientific Computation*, Montreal (IMACS, New Brunswick, NJ, 1982) 427–432.

[12] H.F. Jordan, Experience with pipelined multiple instruction streams, *Proc. IEEE* **72** (1984) 113–123.

[13] D.K. Kahaner, C. Moler and S. Nash, *Numerical Methods and Software* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

[14] E. Ng, S. Thompson and P.G. Tuttle, Experiments with method of lines solvers on a shared memory parallel computer, in: *Advances in Computer Methods for Partial Differential Equations VII, Proc. 7th IMACS Internat. Symp. on Computer Methods for Partial Differential Equations*, Bethlehem, PA (IMACS, New Brunswick, NJ, 1987) 161–166.

[15] J.C. Pirkle and W.E. Schiesser, DSS/2: a transportable FORTRAN 77 code for systems of ordinary and one, two, and three-dimensional partial differential equations, in: *Proceedings of the 1987 Summer Simulation Conference*, Montreal, July 1987.

[16] W.E. Schiesser, *The Numerical Method of Lines Integration of Partial Differential Equations* (Academic Press, New York, 1991).

[17] S. Thompson, Remarks on the choice of a stiff ordinary differential equation solver, *Comput. Math. Appl.* **12A** (11) (1986) 1125–1141.

[18] S. Thompson, The effect of pivoting in sparse ordinary differential equation solvers, *Comput. Math. Appl.* **12A** (12) (1986) 1183–1191.

[19] S. Thompson and P.G. Tuttle, The solution of several representative stiff problems in an industrial environment: the evolution of an O.D.E. solver, in: R.C. Aiken, Ed., *Stiff Computation* (Oxford Univ. Press, New York, 1985) 180–193.

[20] S. Thompson and P.G. Tuttle, Benchmark fluid flow problems for continuous simulation languages, *Comput. Math. Appl.* **12A** (3) (1986) 345–351.