INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 20, 213–231 (1995)

IMPLEMENTATION AND PERFORMANCE OF THE TIME INTEGRATION OF A 3D NUMERICAL TRANSPORT MODEL

B. P. SOMMEIJER AND J. KOK

CWI, PO Box 94079, NL-1090 GB Amsterdam, Netherlands

SUMMARY

The total solution of a three-dimensional model for computing the transport of salinity, pollutants, suspended material (such as sediment or mud), etc. in shallow seas involves many aspects, each of which has to be treated in an optimal way in order to cope with the tremendous computational task involved. In this paper we focus on one of these aspects, i.e. on the time integration, and discuss two numerical solution methods. The emphasis in this paper is on the performance of the methods when implemented on a vector/parallel, shared memory computer such as a Cray-type machine. The first method is an explicit time integrator and can straightforwardly be vectorized and parallelized. Although a stabilizing technique has been applied to this method, it still suffers from a severe time step restriction. The second method is partly implicit, resulting in much better stability characteristics; however, as a consequence of the implicitness, it requires in each step the solution of a large number of tridiagonal systems. When implemented in a standard way, the recursive nature would prevent vectorization, resulting in a very long solution time. Following a suggestion of Golub and Van Loan, this part of the algorithm has been tuned for use on the Cray C98/4256. On the basis of a large-scale test problem, performance results will be presented for various implementations.

KEY WORDS Transport models 3D advection–diffusion equations Numerical time integration Vectorization Parallel processing

1. INTRODUCTION

Mathematical modelling of the transport of salinity, pollutants, suspended material (such as sediment or mud), etc. in shallow seas involves the numerical solution of 3D partial differential equations of advection–diffusion–reaction type. The *total* solution of such a transport model is very complex and has many different aspects, e.g. the choice of the spatial discretization technique, the choice of the corresponding data structures (which has a significant influence on the performance), the treatment of the boundary conditions at the sea bed and at the free water surface, the choice of the time integration method, etc. It is clear that the tremendous computational task imposed by this kind of problem can only be tackled if *each* of these aspects is thoroughly investigated and treated by sophisticated numerical techniques in combination with the most powerful computers. In this paper we focus on the *time integration* part and discuss the implementation of some algorithms that efficiently exploit the vectorization and parallelization facilities offered by Cray-type computers. In a previous paper[1] several time integration techniques have been studied and compared with respect to the efficient solution of the 3D transport problem. The conclusion formulated in Reference 1 was that stabilized, explicit Runge–Kutta methods may be eligible candidates. However, as a disadvantage, these methods have to obey a rather restrictive condition on the time step because of stability reasons. A remedy to avoid the

CCC 0271–2091/95/030213–19

Received March 1994
Revised July 1994

stability condition is offered by the odd–even line hopscotch method. Since this method is partly implicit, a large number of uncoupled tridiagonal systems have to be solved in each step. Normally, the straightforward and numerically stable way to solve these systems does not allow one to exploit vectorization facilities owing to the recursive nature of the standard decomposition approach. Since the systems are uncoupled and of identical shape (which includes a similar storing of the matrix elements and the right-hand-side components), it is possible to vectorize (over all the tridiagonal systems) each step within the standard decomposition algorithm. This technique, which will be termed vectorization-across-tridiagonal-systems, was originally proposed by Golub and Van Loan[2] and has also been used by de Goede[3] in his 3D hydrodynamical shallow water model. This vectorization technique results in a considerably increased performance. Moreover, it can straightforwardly be combined with parallelization for use on a multi(vector)processor system; the *atexpert* utility[4] (to predict performance on a dedicated system) indicates almost optimal speed-up (more than 90% efficiency). When equipped with this linear system solver, the hopscotch method turns out to be superior to the stabilized Runge–Kutta methods.

The paper is organized as follows. In Section 2 we discuss the actual transport model that has been studied. Section 3 is devoted to the discretization of this PDE. First we briefly outline the spatial (or semi-) discretization method and the consequences for the data structures that we need, as well as the resulting vectorization capabilities. Next, and this is the major aim of the present paper, two time integration methods are described in some detail, including the motivations for their choice. In Section 4 we survey the various possibilities to solve the linear systems occurring in the hopscotch integrator and discuss several options to efficiently cope with the Jacobian matrices. Furthermore, the actual implementation of the solver is discussed. Its performance is illustrated in Section 5 by applying it to a large-scale test example. Conclusions are formulated in Section 6.

2. DEFINITION OF THE MATHEMATICAL MODEL

The mathematical model describing transport processes in three dimensions is given by the system of advection–diffusion–reaction equations[5]

$$\frac{\partial c_i}{\partial t} = -\frac{\partial}{\partial x}(uc_i) - \frac{\partial}{\partial y}(vc_i) - \frac{\partial}{\partial z}[(w - w_f)c_i] + \frac{\partial}{\partial x}\left(\varepsilon_x \frac{\partial c_i}{\partial x}\right) + \frac{\partial}{\partial y}\left(\varepsilon_y \frac{\partial c_i}{\partial y}\right) + \frac{\partial}{\partial z}\left(\varepsilon_z \frac{\partial c_i}{\partial z}\right) + g_i, \quad (1)$$

where c_i are the unknown concentrations of the contaminants. The local fluid velocities u, v, and w (in directions x, y, and z respectively) have to be provided by a hydrodynamical model; since this paper merely discusses the solution of (1), the velocity field is considered to be known in advance. The fall velocity w_f, which may be a non-linear function of the concentration, is only relevant in the case of modelling the transport of suspended material. The terms g_i describe chemical reactions, emissions from sources, etc. and therefore depend on the concentrations c_i. Thus the mutual coupling of the equations in system (1) is only due to the functions g_i. In the present paper we shall confine our considerations to the numerical modelling of a *single* transport equation and thus omit the subscript i in the sequel. The extension to systems is a subject of current research and will be discussed in a forthcoming paper. Finally, the diffusion coefficients ε_x, ε_y and ε_z are assumed to be given functions.

The physical domain in space is bounded by vertical, closed boundary planes, by the water elevation surface and by the bottom profile. On these boundaries Dirichlet, Neumann or mixed boundary conditions will be prescribed. Supplementing this with an initial condition, the concentration c can be computed in space and time.

3. DISCRETIZATION METHODS

To arrive at a fully discrete numerical approximation, we will follow the method-of-lines (MOL) approach. That is, we first transform equation (1) into a system of ordinary differential equations (ODEs) by discretizing only the spatial derivatives and leaving time continuous; then these ODEs will be integrated in time numerically. In the following subsections the two steps in this discretization process will be discussed separately.

3.1. Spatial discretization

Typically, the physical domain in space is quite irregular in both horizontal and vertical directions. One possible way to cope with this situation is to map the physical domain on a rectangular box by means of co-ordinate transformations. The advantage of this approach is that the physical boundaries can be exactly represented; however, a serious disadvantage is a much more complicated mathematical model and, additionally, the danger of introducing coefficients of large magnitude. This increases the stiffness in the resulting ODEs and excludes the possibility of using explicit time integration techniques (see Reference 1 for a discussion on this aspect).

Therefore we decided to follow the so-called 'dummy point' approach. By this we mean that the whole physical domain will be enclosed by a rectangular box. Obviously, the disadvantage is that we may introduce many artificial (meaningless) points. However, the regular grid structure allows for an efficient implementation on vector computers, which compensates for the additional computational effort.

Before applying the semi-discretization process, it is convenient to rewrite equation (1) taking into account the particular applications we have in mind. As usual, we shall only consider the *incompressible* case, i.e. we assume $u_x + v_y + w_z = 0$. Furthermore, the diffusion coefficients ε_x, ε_y, ε_z are assumed to be constant and equal to ε. Recalling that we are concentrating on a single transport equation, then simplifies to

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} - v\frac{\partial c}{\partial y} - w\frac{\partial c}{\partial z} + \frac{\partial (w_f c)}{\partial z} + \varepsilon\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial z^2} + \frac{\partial^2 c}{\partial z^2}\right) + g. \tag{2}$$

The first step in the semi-discretization process is to let the physical domain be enclosed by a rectangular box containing the grid points

$$P_{j,k,m} := (x_0 + j\Delta x, y_0 + k\Delta y, z_0 + m\Delta z), \quad j = 0, \ldots, J+1,$$
$$k = 0, \ldots, K+1, \quad m = 0, \ldots, M+1, \tag{3a}$$

where (x_0, y_0, z_0) corresponds with the south-west corner point at the bottom of the rectangular box. It will be assumed that the mesh parameters Δx, Δy and Δz are such that the physical boundaries can be approximated by a subset of grid points with sufficient accuracy. This subset of boundary grid points will be denoted by $\partial\mathbb{B}$ and divides the remaining grid points into outer and inner ones lying 'outside' and 'inside' $\partial\mathbb{B}$. These sets of points are denoted by $\mathbb{B}_{\text{out}}$ and $\mathbb{B}_{\text{in}}$ respectively. Furthermore, we assume that the enclosing box is such that no grid points of $\partial\mathbb{B}$ are on the boundary planes of the box (i.e. the indices j, k and m of the boundary points satisfy $1 \leq j \leq J, 1 \leq k \leq K$ and $1 \leq m \leq M$. Here we remark that the spatial grid does not necessarily need to be equidistant in the various spatial directions. For example, the use of a stretched grid in the vertical may be necessary to adequately resolve the concentration field in the neighbourhood of the sea bed in the case of suspended sediment. Although such a stretched grid will have an impact on the stiffness of the resulting ODE, it is not essential for the discussion that follows.

Next the spatial differential operators $\partial/\partial x, \partial/\partial y$ and $\partial/\partial z$ are replaced by (standard) *symmetric* differences. This choice is justified by the fact that the present problem does not give rise to very large

gradients in the solution (such as steep fronts or even discontinuities), which would necessitate the use of *unsymmetric* (i.e. upwind) discretizations. Since the principal aim of this paper is the time integration aspect, we do not continue the discussion on the best choice of the spatial discretization. Of course, to arrive at the total solution of the transport problem, this aspect also requires thorough investigation. We refer to the recent overview[6] in which a considerable amount of literature on the spatial discretization of advection-dominated partial differential equations is surveyed.

Furthermore, we denote the numerical approximations at $P_{j,k,m}$ to c, u, ... by capitals $C_{j,k,m}$, ... and we introduce the spatial shift operators $X_{\pm}$, $Y_{\pm}$ and $Z_{\pm}$ defined by

$$X_{j,k,m}C_{j,k,m} := C_{j\pm1,k,m}, \qquad Y_{\pm}C_{j,k,m} := C_{j,k\pm1,m}, \qquad Z_{j,k,m} := C_{j,k,m\pm1}.$$

Then on the *computational* set of grid points **S** defined by

$$\mathbb{S} := \{P_{j,k,m}: 1 \le j \le J, 1 \le k \le K, 1 \le m \le M\} \tag{3b}$$

the associated ODEs take the form

$$\begin{aligned}\frac{dC_{j,k,m}}{dt} = &-\left(\frac{1}{2\Delta x}U_{j,k,m}(X_+ - X_-) + \frac{1}{2\Delta y}V_{j,k,m}(Y_+ - Y_-) + \frac{1}{2\Delta z}W_{j,k,m}(Z_+ - Z_-)\right)C_{j,k,m}\\ &+\varepsilon\left(\frac{1}{(\Delta x)^2}(X_+ - 2 + X_-) + \frac{1}{(\Delta y)^2}(Y_+ - 2 + Y_-) + \frac{1}{(\Delta z)^2}(Z_+ - 2 + Z_-)\right)C_{j,k,m}\\ &+\frac{1}{2\Delta z}\omega(C)_{j,k,m}(Z_+ - Z_-)C_{j,k,m} + G_{j,k,m},\end{aligned} \tag{4a}$$

where $\omega(c) := w_f(c) + c\partial w_f(c)/\partial c$.

The next step is to take into account the boundary conditions. If $P_{j,k,m}$ is a (physical) boundary point where the boundary condition is of Dirichlet type, then $C_{j,k,m}$ is explicitly given, so that by means of differentiation with respect to time we obtain ODEs of the form

$$\frac{dC_{j,k,m}}{dt} = d_{j,k,m}(t), \tag{4b}$$

which should replace the ODEs occurring in (4a) at all Dirichlet-type boundary points.

If $P_{j,k,m}$ is a boundary point of non-Dirichlet type, then the corresponding ODE in (4a) asks for the concentration at one or more outer grid points. By means of the boundary conditions these auxiliary concentrations can be explicitly expressed in terms of concentrations at inner (or boundary) grid points. Especially at the free water surface and the sea bed this will give rise to rather complicated expressions involving interpolations. This aspect is certainly worth a *separate* study and is not discussed in this paper. Finally we assume 'dummy' concentrations at all points in the computational set S which are in B_{out}.

In conclusion, the equations (4) corresponding to the set of grid points **S** as defined by (3b) define a second-order-consistent semi-discretization of the initial–boundary value problem (1) of dimension $N := JKM$. Notice that only the concentrations defined by this system of ODEs corresponding to the grid points of $\mathbb{B}_{in}$ and $\partial\mathbb{B}$ are relevant.

In the analysis of time integrators for (4) it is more convenient to represent this system in the form

$$\begin{aligned}\frac{d\mathbf{C}(t)}{dt} = \mathbf{F}(t,\mathbf{C}(t)) := &A_x(t)\mathbf{C}(t) + A_y(t)\mathbf{C} + A_z(t)\mathbf{C}(t) + W_z(\mathbf{C}(t))\mathbf{C}(t) + \mathbf{G}(t,\mathbf{C}(t))\\ &+\mathbf{B}_x(t) + \mathbf{B}_y(t) + \mathbf{B}_z(t),\end{aligned} \tag{5}$$

where $\mathbf{C}(t)$ is a vector of dimension N containing all concentrations defined at the points of $\mathbb{S}$, $A_x(t)$, $A_y(t)$ and $A_z(t)$ are N-by-N tridiagonal matrices depending only on t, and $W_z(\mathbf{C})$ is an N-by-N tridiagonal

matrix which vanishes if the fall velocity w_f occurring in (1) vanishes. The N-dimensional vector $\mathbf{G}(t, \mathbf{C}(t))$ is the discrete analogue of the source function g, and the vectors $\mathbf{B}_x(t)$, $\mathbf{B}_y(t)$ and $\mathbf{B}_z(t)$ represent the inhomogeneous contributions of the boundary conditions at the vertical east and west boundaries, at the vertical north and south boundaries and at the surface and bottom boundaries respectively. Notice that, assuming a three-point coupling in the spatial discretization, (5) is the *general* form of the resulting ODEs.

Owing to the above 'dummy point' approach, the data structures in the code are extremely simple: the arrays U, V and W (containing the velocity field) and the array DCDT (containing the time derivatives of the concentrations) are defined on the computational domain $\mathbb{S}$; hence these three-times-subscripted arrays have indices running from 1 to J, from 1 to K and from 1 to M respectively. The array C (containing the concentration vector $\mathbf{C}$) is defined on the whole enclosing box, where the indices run respectively from 0 to $J + 1$, from 0 to $K + 1$ and from 0 to $M + 1$. A consequence of this approach is that the whole array C will contain values including the meaningless values corresponding to grid points lying in $\mathbb{B}_{\text{out}}$. Hence, on entering the subroutine F to calculate the time derivatives, the following steps have to be performed: (i) in the case of non-Dirichlet boundary conditions the points in $\mathbb{B}_{\text{out}}$ which are at a distance of one grid point from $\partial\mathbb{B}$ should be calculated using the boundary conditions (notice that these points may lie on the boundary of the enclosing box); (ii) calculate the time derivatives of the concentration (i.e. evaluate the right-hand side of (4a)) at *all* points belonging to $\mathbb{S}$ and store these values in the array DCDT; (iii) in the case of Dirichlet boundary points the corresponding values in DCDT have to be overwritten by the expressions defined in (4b). In this way an efficient implementation of the subroutine F on a vector machine can be obtained (see the results in Section 5).

3.2. Time integration methods

In Reference 1 several time integration techniques have been discussed and compared on the basis of their sutability to solve transport equations. It turned out that the explicit, stabilized Runge–Kutta methods and the odd–even line hopscotch method are the most promising for integrating the space-discretized transport equation (5) on Cray-type computers. In the following subsubsections these methods will be discussed in detail.

3.2.1. Stabilized Runge–Kutta methods. The q-stage Runge–Kutta method that we consider to advance the solution of (5) over one step of size Δt is defined

$$\begin{aligned} \mathbf{C}^{(0)} &= \mathbf{C}_n, \\ \mathbf{C}^{(j)} &= \mathbf{C}_n + \alpha_j \Delta t \mathbf{F}(t_n + \mu_j \Delta t, \mathbf{C}^{(j-1)}), \quad j = 1, \ldots, q, \\ \mathbf{C}_{n+1} &= \mathbf{C}^{(q)}. \end{aligned} \tag{6}$$

Here $\mathbf{C}_n$ denotes the numerical approximation to the solution of the ODE (5) at $t_n = n\,\Delta t$ and the quantities $\mathbf{C}^{(j)}$ denote intermediate approximations. In this method the free parameters α_j and μ_j will be chosen to give the method the required properties: second-order accuracy in time, as large a stability boundary as possible and minimal costs per step. The first requirement is fulfilled by setting $\alpha_q = 1$ and $\alpha_{q-1} = \mu_q = \frac{1}{2}$. The remaining α_js can be used to give the method optimal stability characteristics. Since our transport problem is usually convection-dominated, we decided to optimize the stability boundary along the imaginary axis (for a more detailed discussion on this aspect we refer to Reference 1). It is well known[7,8] that for *odd* values of q the second-order scheme (6) possesses an optimal imaginary stability boundary $\beta_{\text{imag}} = q - 1$. In Table I the corresponding α_j-values are listed for various values of q (for the sake of completeness we also give the values for β_{real}, the corresponding stability boundary along the real axis). Finally, the requirement of minimal costs is obtained by

Table I. Parameters α_j to obtain an optimal imaginary stability boundary β_{imag} for (6)

q	α_1	α_2	α_3	α_4	α_5	α_6	α_7	β_{imag}	β_{real}
4	1/4	1/3						$2\sqrt{2}$	2·78
5	1/4	1/6	3/8					4	2·59
7	1/6	1/12	2/9	4/19	19/54			6	3·00
9	1/8	1/20	5/32	2/17	17/80	5/22	11/32	8	3·31

'freezing' the t-argument in the first $q-1$ stages, i.e. $\mu_j = 0, j = 1, \ldots, q-1$. In the case of (5) a substantial part of the effort in calculating $\mathbf{F}(t, \mathbf{C})$ comes from the evaluation of the matrices $A_x(t)$, $A_y(t)$ and $A_z(t)$, the vector $\mathbf{G}(t, \mathbf{C}(t))$ and the boundary conditions. Hence, except for the first and the last stage (i.e. for $j = 0$ and $j = q$), these quantities do not need to be evaluated.

Moreover, the storage requirements are relatively modest and the structure of system (5) allows an extremely efficient implementation on vector computers.

3.2.2. Odd–even line hopscotch methods. The class of hopscotch methods[9] belongs to the class of so-called operator-splitting methods. To define the specific odd–even line hopscotch (OELH) method that we have in mind, the right-hand-side function $\mathbf{F}$ in (5) is assumed to be split into

$$\mathbf{F}(t, \mathbf{C}(t)) = \mathbf{F}_0(t, \mathbf{C}(t)) + \mathbf{F}_*(t, \mathbf{C}(t)), \tag{7}$$

where $\mathbf{F}_0$ is the vector that is obtained from $\mathbf{F}$ by replacing all components corresponding to a grid point $P_{j,k,m}$ where $j + k$ assumes an *even* value by zero. Similarly, $\mathbf{F}_*$ is obtained if all elements in $\mathbf{F}$ corresponding to a grid point for which $j + k$ is *odd* are replaced by zero. Notice that the third index, m, is not involved in this definition, which means that we apply the same splitting on each horizontal plane of the grid, or equivalently, *all* grid points lying on the *same* vertical gridline are either in $\mathbf{F}_0$ or in $\mathbf{F}_*$. This observation is crucial for the OELH algorithm. The configuration of the computational points is illustrated in Figure 1.

Now we define the two-stage, second-order splitting method

$$\begin{aligned} \mathbf{C}_{n+1/2} &= \mathbf{C}_n + \tfrac{1}{2}\Delta t \mathbf{F}_0(t_{n+1/2}, \mathbf{C}_{n+1/2}) + \tfrac{1}{2}\Delta t \mathbf{F}_*(t_n, \mathbf{C}_n), \\ \mathbf{C}_{n+1} &= \mathbf{C}_{n+1/2} + \tfrac{1}{2}\Delta t \mathbf{F}_0(t_{n+1/2}, \mathbf{C}_{n+1/2}) + \tfrac{1}{2}\Delta t \mathbf{F}_*(t_{n+1}, \mathbf{C}_{n+1}). \end{aligned} \tag{8}$$

Starting from $\mathbf{C}_n$, the numerical solution at $t = t_n$, the approximation $\mathbf{C}_{n+1}$ at the next step point is obtained by performing two stages; $\mathbf{C}_{n+1/2}$ can be considered as an intermediate approximation.

Owing to the aforementioned splitting and observing that the discrete system (4) possesses a *three*-point coupling in the horizontal direction, we see that in the first stage the '$*$-components' of $\mathbf{C}_{n+1/2}$ can be computed without evaluating $\mathbf{F}_0$. Hence the first stage can be split into an explicit forward Euler-type calculation for the '$*$-components', followed by an implicit backward Euler-type calculation for the 'o-components'. Since the '$*$-components' are already known at the intermediate time level $t_n + \Delta t/2$, each 'o-component' is only coupled in the vertical direction (see Figure 1). Since we also used a *three*-point coupling to discretize $\partial/\partial z$ and $\partial^2/\partial z^2$, this implicit part of the algorithm results in the solution of tridiagonal systems along each vertical grid line. Of course, similar arguments hold for the second stage, but now the roles of the two types of components are interchanged.

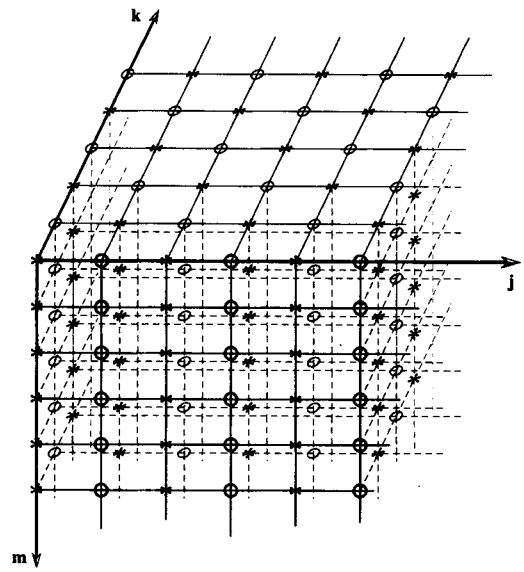

Figure 1. Configuration of the grid points to be used in the OELH method. Owing to the three-point coupling and the absence of mixed derivative terms, the calculations in the horizontal directions can be made explicit. Associated with each vertical grid line we obtain a tridiagonal system; the quintessence of the method is that all these systems are uncoupled

Using these observations, the scheme (8) can be recast in the form

$$\begin{aligned}
&\mathbf{C}_{*,n+1/2} = \mathbf{C}_{*,n} + \tfrac{1}{2}\Delta t\mathbf{F}_*(t_n, \mathbf{C}_n) = \mathbf{C}_{*,n} + (\mathbf{C}_{*,n} - \mathbf{C}_{*,n-1/2}).\\
&\mathbf{C}_{o,n+1/2} = \mathbf{C}_{0,n} + \tfrac{1}{2}\Delta t\mathbf{F}_0(t_{n+1/2}, \mathbf{C}_{n+1/2}),\\
&\mathbf{C}_{0,n+1} = \mathbf{C}_{0,n+1/2} + \tfrac{1}{2}\Delta t\mathbf{F}_0(t_{n+1/2}, \mathbf{C}_{n+1/2}) = \mathbf{C}_{0,n+1/2} + (\mathbf{C}_{0,n+1/2} - \mathbf{C}_{0,n}).\\
&\mathbf{C}_{*,n+1} = \mathbf{C}_{*,n+1/2} + \tfrac{1}{2}\Delta t\mathbf{F}_*(t_{n+1}, \mathbf{C}_{n+1}).
\end{aligned} \tag{9}$$

Observe that in the explicit part of both stages the corresponding **F**-evaluation can be saved. Apart from the very first step, these **F**-evaluations can be expressed in terms of the concentrations at a previous time level. On using this substitution, we arrive at the so-called 'fast form'. As a result, the total amount of work per step consists of solving JK uncoupled, implicit relations each of dimension M. For this purpose one usually applies a (modified) Newton process. Notice that one Newton iteration will suffice if the fall velocity term w_f is not needed in the model, since then the equation is linear in **C**. This case has been considered in the numerical experiments, so that (9) requires *one* **F**-evaluation over the whole field plus the solution of JK tridiagonal linear systems of dimension M (in Section 4 we give a detailed description how these systems can be solved efficiently on a vector computer). It is worth mentioning that the storage requirements of the OELH method are quite modest; apart from the arrays to store the velocity field (which are anyhow needed by every 'transport solver'), (9) needs two arrays to store the concentration vector **C** at the various time levels, one array to hold the time derivative (i.e. DCDT), three arrays for housing the tridiagonal Jacobian matrices and 'half' an array to gather the right-hand-side vectors in the linear equations, which amounts to 6·5 arrays (of dimension N).

The main reason for constructing this particular variant within the hopscotch family is that usually the most severe restrictions on the time step originate from teh discretization in the *vertical* direction, i.e. (see (1) and (4a)) the terms $|w|/\Delta z$ and $\varepsilon_z/(\Delta z)^2$ are usually larger than the similar expressions corresponding to the horizontal direction. Hence in explicit methods the time step will be dictated by

stability conditions induced by the vertical discretization. Since the OELH method is explicit in the horizontal but implicit (and hence unconditionally stable) in the vertical, it usually allows for time steps which are in accordance with the size that we would require for accuracy reasons.

4. SOLVING THE LINEAR SYSTEMS IN VECTOR MODE

In the previous section we have seen that the OELH method (9) requires the solution of JK tridiagonal systems of dimension M. In this section we describe the selected linear system solver which is tailor-made for the present application of a three-dimensional numerical transport model. Details of the implementation with respect to optimal vector performance are discussed as well as the possibility of exploiting several parallel processors. Furthermore, we explain some additional options that have been incorporated in the linear solver (which can be of use in the case of different time integrators) and finally we present performance results.

4.1. Algorithmic aspects

In solving the tridiagonal system $Ax = b$, we decompose the matrix A as $LD^{-1}U$, where D is a diagonal matrix and L and U are respectively lower and upper bidiagonal matrices, both with unit diagonal. During development we have compared this decomposition with the obvious alternatives LU and $L^{-1}U$, in particular with respect to the overall number of operations and possibilities for fine tuning regarding vector performance. Apart from the opportunity that $LD^{-1}U$ offers to save on the number of divisions, the merits of the three methods do not differ conspicuously. The observed slight advantage of the first method can be attributed to the circumstances that it combines best (on average) with all additional facilities that we have chosen to implement, namely to combine decompostion and solution in one subroutine and the possibility to reuse a stored decomposition. Furthermore, the implementation is suitable for all possible couplings of the unknowns which may stem from a coupling in either the x-, y- or z-direction of the physical domain. This feature is not actually exploited by the OELH method (we recall that this method is only implicit in the vertical), but it gives the linear solver the flexibility to be used in e.g. LOD-type integration methods (see Reference 1 for a discussion of such methods in the present context).

Standard solvers for large tridiagonal systems are publicly available through e.g. LAPACK[10] (which provides the subroutines SGTTRF for factorizing a tridiagonal matrix and SGTTRS for the subsequence forward/backward substitution) and the Cray Scientific Subroutine library SCILIB (SDTSOL would be a candidate). The main reason for developing our own implementation is the fact that all tridiagonal systems are *uncoupled* and, moreover, are of similar shape and have equal dimension. These properties allow an efficient implementation especially for vector processors employing the method of *vectorization-across-systems* which is described below.

We will first describe the $LD^{-1}U$ decomposition method for a *single* tridiagonal system (of dimension n). With proper subscript notation the method reads

```
D1 := 1/A1,1
for i = 2 until n
    Ui-1,i := Ai-1,i * Di-1    (Ui-1,i not needed)
    Li,i-1 := Ai,i-1 * Di-1
    Di := 1/(Ai,i - Li,i-1 * Ai-1,i)
(Solution of Ax = b: )
```

$x_1 := b_1$

for i = 2 **until** n

$x_i := b_i - L_{i,i-1} * x_{i-1}$

$x_n := D_n * x_n$

for i = n − 1 **step** − 1 **until** 1

$x_i := D_i * (x_i - A_{i,i+1} * x_{i+1})$

An obvious Fortran implementation, which receives the (co)diagonals of the input matrix A in one-dimensional arrays L(2:*n*), D(1:*n*) and U(2:*n*) delivers the decomposition results in the same three arrays and the solution in the array B that initially contained the right-hand-side vector, is given by

```
C       Input: matrix in L(2:N), D(1:N), U(2:N), right-hand side in B(1:N).
C       i-th row of L and i-th column of U and i-th diagonal element of D:
        D(1) = 1.0/D(1)
        DO I = 2, N
            L(I) = L(I) * D(I-1)
            D(I) = 1.0/(D(I) - L(I) * U(I))
        END DO
C       We now have matrix L with unit diagonal in L, D^-1 in D, U is not computed
C       We proceed with forward, scale and back solve:
        DO I = 2, N
            B(I) = B(I) - L(I) * B(I-1)
        END DO
        B(N) = D(N) * B(N)
        DO I = N-1, 1, -1
            B(I) = D(I) * (B(I) - U(I+1) * B(I+1))
        END DO
C       Solution in B.
```

For the OELH method, in which we have to solve *JK* systems of dimension *M*, a straightforward implementation would be to put the above implementation for solving a single system into a loop

```
DO IND = 1, J*K
    {the previous source code, with N replaced by the actual dimension M of each system}
    {and, since we are dealing with three-dimensional arrays, all subscripts (I) replaced by (IND,1,I)}
    {and similar substitutions for I+1 and I−1}
END DO
```

However, owing to the *recursive* nature in the inner loops, this approach will result in a poor performance on vector processors. The trick to obtain good vector code, as described by Golub and Van Loan[2] and called '*vectorization-across-tridiagonal-systems*', is to interchange the inner and outer loops. This results in a number of nested loops with the 'IND-loop' as the inner one. For example, the following loop in the single-system implementation

```
DO I = 2, N
    B(I) = B(I) - L(I) * B(I-1)
END DO
```

becomes

```
DO I = 2, N
    DO IND = 1, J*K
        B(IND, 1, I) = B(IND, 1, I) - L(IND, 1, I) * (B(IND, 1, I - 1)
    END DO
END DO
```

In fact, each step in the original Gaussian elimination process is now performed in vector mode, since the vectorizable inner loop runs over all systems. This approach works perfectly well for the present class of problems.

We remark that in the above description the index l is always the *third* index in the three-dimensional arrays; this is because the OELH method is implicit in the vertical (i.e. z-) direction, which corresponds with the third index. As a matter of fact, our tridiagonal solver is more general in the sense that it is also capable of treating systems originating from implicitness in the x- or y-direction; in those cases the index l will be the first or second index respectively in all three-dimensional arrays. In the next subsection the influence on the performance is shown. For full details on all three cases of coupling we refer to the Fortran 77 listing in the appendix to the institute report of the present paper.[11]

4.2. Implementation aspects

We will briefly touch on a number of issues that are of importance for the development of the final implementation. During the development several design decisions concerning details were taken after intensive measurement of performance differences for all of the possible branches discussed below.

(1) *Branching with respect to the direction of the coupling of unknowns.* Different implementations are used to treat the linear systems arising from a coupling in the x-, y- or z-direction of the physical domain. In these cases the order in which data storage places are accessed is completely different.

(2) *Keeping and reusing the decomposition results.* One design decision taken is to store the decomposition results in such a way that they can be reused efficiently. Therefore the results of the decomposition (and of the solution) are delivered in the storage space containing the original input. Although costs are connected with this way of storing the decomposition, the advantage is clear in the case where the decomposition is reused for a system with only a new right-hand side.

(3) *Parallel processing possibility.* The final implementation of the tridiagonal system solver is parametrized with N_{PP}, the number of parallel processors to be employed. It is extremely simple to split the complete workload into N_{PP} portions of (approximately) equal size which are completely independent. A speed-up of 4 (using all four processors of the Cray C98/4256) would be obtainable, but the splitting of long loops into four parts slightly decreases the performance on a single vector processor. This is confirmed by a couple of experiments where we obtained a parallel speed-up of 3.3–3.8.

(4) *Fine tuning.* The actual implementation receives the data of a Jacobian matrix J and a time increment Δt from which it first computes the coefficient matrices $A = I - \Delta t\, J$ that will subsequently be decomposed. Much attention has been given to combining loops that are described as different tasks into single loops whenever possible. This loop combining has been applied not only for the above-mentioned setting-up of A and its decomposition, but also for combining decomposition and forward substitution.

(5) *About vectorization-across-tridiagonal-systems.* In order to employ vectorization-aross-tridiagonal-systems, Golub and Van Loan[2] recommended that the data should be reordered such that successive computations (i.e. corresponding iterations of different tridiagonal systems) can access successive array elements. In our application this would require complete copyint of the data in a different order before proceeding with the decomposition. However, on the Cray C98/4256 such a data conversion is not at all needed, since processing is equally efficient with loops in which we jump with large but constant stride through the accessed memory.

In Table II we summarize the results of timing experiments for the tridiagonal system solver. The experiments have been carried out with $J = K = 101$ and $M = 11$, so all arrays have dimensions (101, 101, 11).

Table II. CPU times (in milliseconds) and Mflop rates on a Cray C98/4256 using one processor

	Scalar mode		Vector mode		
Coupling	CPU	Mflops	CPU	Mflops	Speed-up
In x-direction	66·9	26·7	3·58	489·1	18.7
In y direction	71·8	25·0	3·57	502·8	20·1
In z-direction	67·1	25·4	3·14	542·4	21·4

5. NUMERICAL EXPERIMENTS

To show the performance of the methods described in the previous sections, we take the example problem (see also Reference 1)

$$\frac{\partial c}{\partial t}+\frac{\partial(uc)}{\partial t}+\frac{\partial(vc)}{\partial y}+\frac{\partial(wc)}{\partial z}=\varepsilon\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)c+g(t,x,y,z), \quad 0\le t\le T, \tag{10a}$$

with Neumann conditions of the form $\partial c/\partial n = h(t,x,y,z)$ on all boundaries of the physical domain $0\le x\le L_x, 0\le y\le L_y, -L_z\le z\le 0$. In our experiments, where the emphasis is on the behaviour in time of the solution procedure, it is convenient to have the exact solution at our disposal. This allows for easy measurement of the global errors. In terms of the scaled co-ordinates $\tilde{x}:=x/L_x, \tilde{y}:=y/L_y$ and $\tilde{z}:=L_z$ the concentration c is prescribed as

$$c(t,x,y,z)=\exp\{\tilde{z}-f(t)-\gamma[(\tilde{x}-r(t))^2+(\tilde{y}-s(t))^2]\}, \tag{10b}$$

where the functions r, s and f are defined by $r(t):=[2+\cos(2/T_p)]/4, s(t):=[2+\sin(2\pi t/T_p]/4$ and $f(t):=4t/(T_b+t)$. Hence a local concentration is advected in a rotation (with period T_p) around the center of the domain. Furthermore, we see that the concentration is slightly decreasing in the vertical direction (when going downwards) and damped in time. (To get a better impression of the behaviour of this function, we refer to the pictures in Figures 2(a) and 2(b), where the solution is plotted for various points in space and time.) The inhomogenous function g and the function h defining the boundary conditions follow from this exact solution. Owing to the special (i.e. exponential) structure of (10b), the functions g and h can be written in the form $g=c\tilde{g}$ and $h=c\tilde{h}$. This form has been used in the implementation.

We take the following values for the parameters: $L_x = L\,y = 20{,}000$ m, $L_z = 100$ m, $\varepsilon = 0{\cdot}5\ \mathrm{m^2\ s^{-1}}$, $T_p = 43{,}200$ s (12 h), $T_b = 32{,}400$ s. The (dimensionless) parameter γ can be used to adjust the solution. To stress the *local* nature of the concentration, we have chosen c as large as 30. For the length of the integration interval T we have used two different values, namely $T_1 = 10{,}800$ s (3 h) and $T_2 = 432{,}000$ s (which equals 5 days). The particular value will be specified in the tables of results (at these points in time the damping effects are $\exp(-f(T_1))\approx 0\cdot 37$ and $\exp(-f(T_2))\approx 0\cdot 024$). The concentration c is assumed to be measured in $\mathrm{kg\ m^{-3}}$.

The velocity fields are prescribed by the analytical expressions

$$\begin{aligned} u(t,x,y,z) &= c_1\sin(\tilde{x}+\tilde{y})\sin(\beta\tilde{z})d(t),\\ v(t,x,y,z) &= C_2\cos(\tilde{x}+\tilde{y})\sin(\beta\tilde{z})d(t),\\ w(t,x,y,z) &= \left(-\frac{C_1}{L_x}\cos(\tilde{x}+\tilde{y})+\frac{C_2}{L_y}\sin(\tilde{x}+\tilde{y})\right)\left(-\frac{L_z}{\beta}\cos(\beta\tilde{z})\right)d(t), \end{aligned} \tag{11}$$

with $d(t)=\cos(2\pi t/T_p)$, $C_1 = 3$, $C_2 = 4$ and $\beta = 0{\cdot}05$. We remark that these fluid velocities satisfy the

relation for local mass balance, i.e.

$$\frac{\partial u}{\partial t}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}=0,$$

so that (10a) can be simplified indeed to the form (2); the fall velocity term w_f is not taken into account in our tests.

To obtain insight to what extent the length of the vectors will influence the performance of the algorithms, we have done experiments on two spatial grids: grid I is defined by $J = K = 101$ and $M = 11$, amounting to about 10^5 grid points; grid II is defined by $J = K = 201$ and $M = 21$, amounting to about $8{\cdot}5 \times 10^5$ grid points.

5.1. Algorithmic tests of the time integration methods

First we will compare the time integrators with respect to their *numerical* properties such as accuracy and stability. Therefore they are applied to problem (10), (11) defined on grid I. In Table III we show the (numerical) behaviour of the OELH method and several stabilized RK methods based on q stages (in the sequel these methods will be denoted by RK2q). As the end point of the integration we choose $T_1 = 10{,}800$. This table shows that OELH is the most stable time integration process (an asterisk denotes unstable behaviour). Moreover, it is quite accurate, since already for Δt as large as 135 the time integration error is almost negligible compared with the spatial discretization error, which is approximately equal to 0·0005 for this grid. Figures 2(a) and 2(b) show the initial concentration $c(0, x, y, z)$ according to (10b) and the numerical solution at $t = T_1$ obtained with the OELH method using $\Delta t = 135$. Furthermore, we observe that the stability of the RK methods is improved as the number of stages increases.

The next question is of course which stabilized RK method is the most efficient one, i.e. what is the optimal q-value? To answer this question, we have to take into account the costs of the various stages. As pointed out in Section 3.2.1., an RK2q method requires two 'expensive' **F**-evaluations and q-2 'cheap' **F**-evaluations per step (since the overhead in an RK method is negligible, we only count **F**-evaluations). Let σ denote the fraction that can be saved be evaluating a 'cheap' **F**. Then the total cost $E(q, \sigma)$ (in terms of full **F**-evaluations) over the whole integration interval is given by $E(q,\sigma) := N_{st}[2 + (q-2)(1-\sigma)]$, where N_{st} is the minimal number of time steps that has to be taken for stability reasons. If N_{st} is only determined by β_{imag}, then $E(q, \sigma)$ reduces to $D[q(1-\sigma) + 2\sigma)]/(q-1)$, where D is a constant depending on the length of the integration interval and on the spectral radius of $\partial\mathbf{F}/\partial\mathbf{C}$. In this case $E(q, \sigma)$ is a monotonically decreasing function of q, which suggests a large q-value (yielding increasing efficiency as $\sigma \to 1$). However, in practice the value of β_{real} is also relevant, especially if $\Delta z \to 0$ (see also the experiments described in Section 5.3). This is already noticeable from the maximally allowed time steps as listed in Table III,

Table III. Global errors for OELH and several stabilized RK methods at $T_1 = 10{,}800$

Steps	Δt	OELH	RK24	RK25	RK27	RK29
10	1080	0·0421	*	*	*	*
20	540	0·00875	*	*	*	*
40	270	0·00196	*	•	*	•
80	135	0·00049	*	*	*	0·00038
95	114	0·00049	*	*	0·00040	0·00041
125	86·4	0·00050	*	0·00043	0·00044	0·00044
160	67·5	0·00050	0·00047	0·00046	0·00046	0.00046

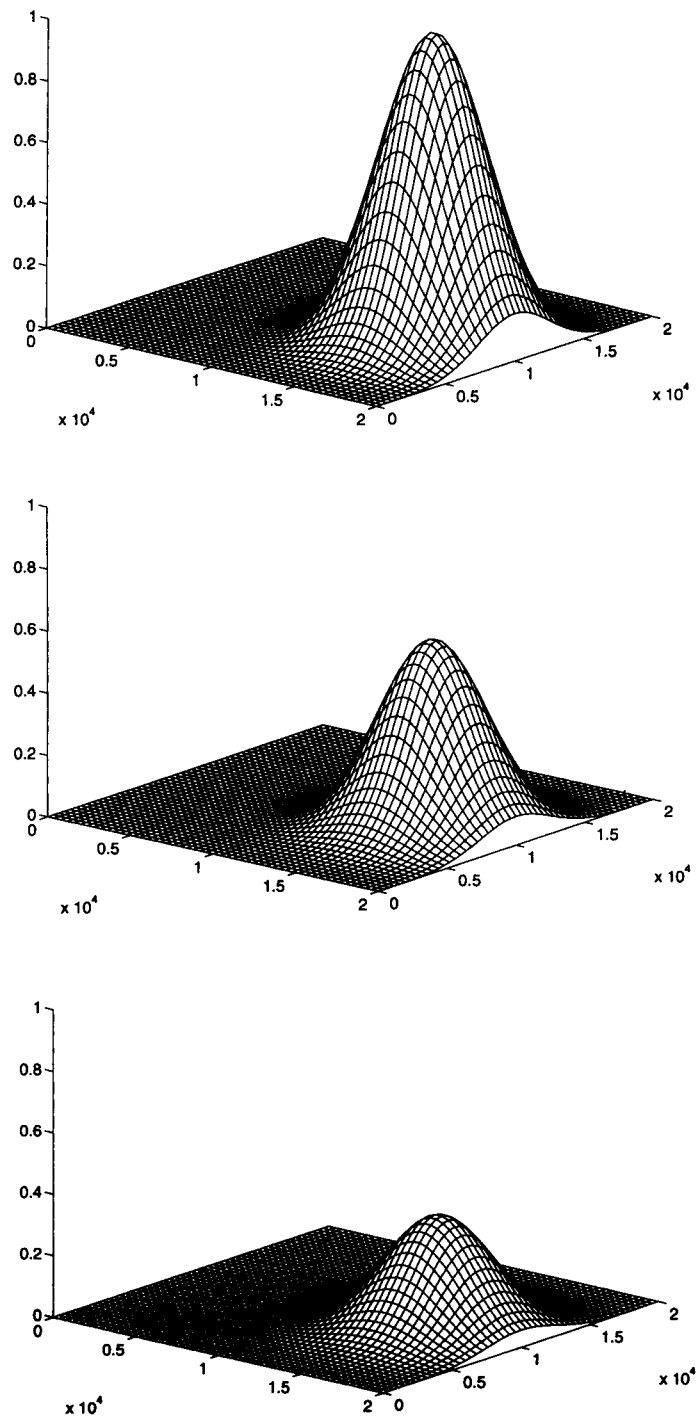

Figure 2(a). The concentration $c(t, x, y, z)$ for $t = 0$ at three horizontal planes, namely at the surface ($z = 0$), halfway down ($z = -50$ m) and at the bottom ($z = -100$ m)

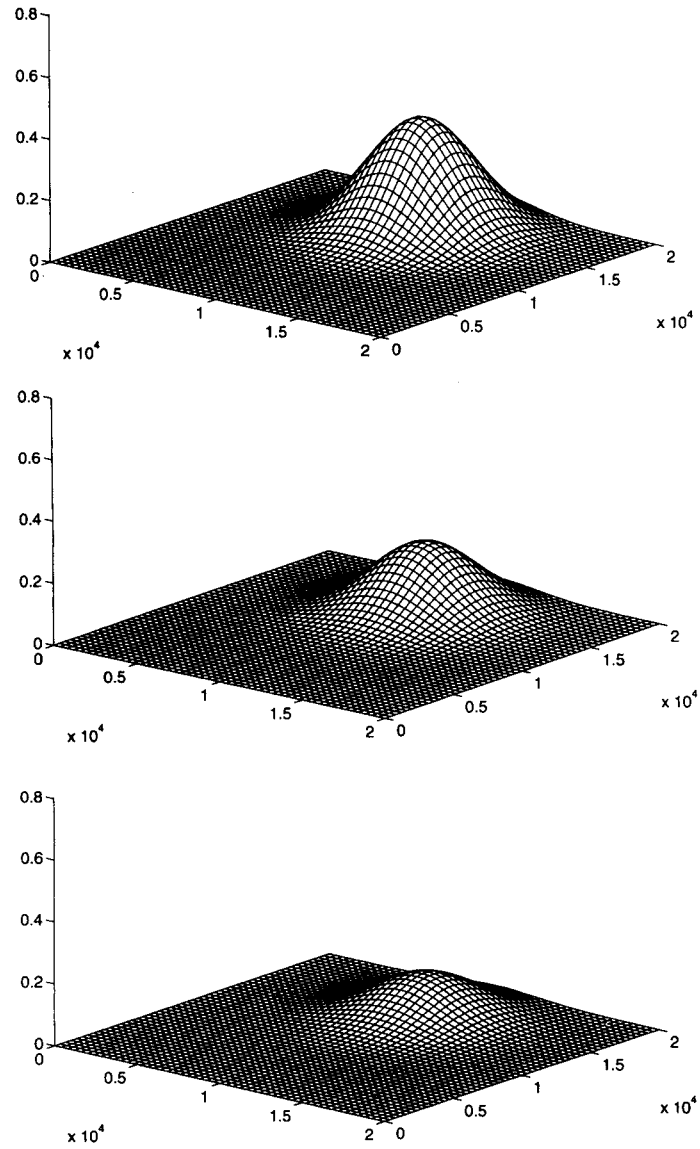

Figure 2(b). The computed concentration after 3 h ($t = T_1 = 10{,}800$) at three horizontal planes, namely at the surface ($z = 0$), halfway down ($z = -50$ m) and at the bottom ($z = -100$ m). At the surface the maximum of the concentration equals 0·37. Furthermore, we see that the solution has been advected over 90°

which show that the increase in Δt is less than might be expected on the basis of β_{image}. In conclusion, the optimal value of q depends on the ratio between $|w|/\Delta z$ and $\varepsilon_z/(\Delta z)^2$ and on the value of σ. For the problem described in this section, $q = 7$ turned out to be a good choice. Therefore in the sequel we will confine ourselves to the OELH and RK27 methods.

Next we let the methods be subjected to a more severe stability test by integrating up to $T_2 = 432{,}000$. The results, given in Table IV, reveal that both methods show a stability behaviour that is similar to their behaviour on the short integration interval.

Table IV. Global errors for OELH and RK27 at $T_2 = 432{,}000$

Steps	Δt	OELH	RK27
750	576	0·0527	*
800	540	0·00484	*
1600	270	0·00174	*
3200	135	0·00101	*
3500	123	0·00098	*
3800	114	0·00096	0·00080
4000	108	0·00094	0·00081

5.2. Performance results

In this subsection we describe the performance of both integration methods when implemented on the Cray C98/4256, which is a four-processor machine. In all experiments we used the CF77 compiling system. Performance results in scalar and vector modes are respectively obtained used **cf77-Wf" -o novector"** and **cf77 -Zv**. Moreover, we frequently used the package **perfview**[4] (using the flags **-F** and **-lperf**) to obtain the megaflop rates (i.e. 10^6 floating point operations per second) and CPU times of the various routines. All experiments have been performed on one processor. Since the clock cycle time of the Cray C98/4256 equals 4·2 ns and each processor is equipped with two vector pipes, the optimal vector speed equals 476 Mflops; in the exceptional case that a multiply and an addition can always be chained, this theoretical peak performance can be multiplied by a factor of two.

In Table V the performance results (on grid I at end point $T_1 = 10{,}800$) are given for the RK27 method using the smallest number of steps, i.e. $N_{st} = 95$.

The performance results for the OELH method are given in Table VI. Here we used $N_{st} = 80$, corresponding to $\Delta t = 135$. This time step yields a global error which is comparable with that of the RK27 method using the largest possible step.

These tables give rise to the following remarks and conclusions.

Table V. Vector performance of the main routines in RK27

Routine	Number of calls	Averge time (s)	Accumulated percentage of CPU time spent	Mflop rate
F (derivative)	665	$5{\cdot}7 \times 10^{-3}$	74·9	549
G (source term)	190	$4{\cdot}4 \times 10^{-3}$	91·6	516
RK (driver)	1	$3{\cdot}7 \times 10^{-1}$	98·8	503

Table VI. Vector performance of the main routines in OELH

Routine	Number of calls	Averge time (s)	Accumulated percentage of CPU time spent	Mflop rate
F (derivative)	161	$4{\cdot}4 \times 10^{-3}$	35·8	354
G (source term)	161	$3{\cdot}0 \times 10^{-3}$	60·1	395
TRI3P3 (tridiagonal solver)	160	$1{\cdot}6 \times 10^{-3}$	73·0	544
JACOB (evaluates Jacobian)	160	$1{\cdot}2 \times 10^{-3}$	82·7	219
IMPL (solves implicit relation)	160	$1{\cdot}1 \times 10^{-3}$	92·0	197
HS (driver)	1	$8{\cdot}5 \times 10^{-2}$	96·3	426

(1) The high megaflop rates obtained in most routines clearly indicates that both time integrators vectorize extremely well. Since these methods have attractive numerical properties as well, they are both candidates for the efficient solution of transport problems.

(2) Except for the driver, most routines in OELH are called about $2N_{st}$ times; the factor of two originates from the fact that the OELH method has two stages (notice that the fast form of OELH requires some initialization, resulting in one extra call of F and G). Observe that the average time spent in F is lower than the corresponding time for RK27. This is because OELH needs to evaluate (per call of F) only half the number of points. The stride 2 that is involved in the hopscotch algorithm reduces the vector performance. The same arguments apply to G. This stride effect is more pronounced in the routines IMPL and JACOB, since there the number of arithmetic operations is relatively low.

(3) It should be remarked that in our test problem the g-function is unrealistically expensive. This is because we have used this function to let the prescribed concentration (10b) be the exact solution. In real-life problems this function will be much cheaper. If we ignore the time that the integrators spend in the routine G, then RK27 spends 89·9% of its time in the derivative function; for OELH this change would result in 47·4% for F. For a code based on an *implicit* method this percentage is pretty high, showing that the linear algebra part is efficiently treated by the vectorization-across-tridiagonal-systems approach. It is of interest to mention that if this part were performed in the traditional (i.e. recursive and hence non-vectorizable) way, then the first three lines in Table VI would change to the following:

Routine	Number of calls	Averge time (s)	Accumulated percentage of CPU time spent	Mflop rate
TRI3P3 (tridiagonal solver)	160	$3{\cdot}5 \times 10^{-2}$	76·5	25
F (derivative)	161	$4{\cdot}4 \times 10^{-3}$	86·2	354
G (source term)	161	$3{\cdot}0 \times 10^{-3}$	92·7	395

This is in accordance with the usual experience that an implicit method spends most of its time in solving the systems.

Of course, more interesting than the vector performance of the algorithms is the actual CPU time that they need to arrive at an accurate solution. A global performance is given in Table VII, where we have given the performance in scalar mode as well. From this table we see that the methods are more than 10 times faster when run in vector mode; this factor confirms the excellent vectorization capabilities of both algorithms. Furthermore, it is clear that the OELH method is able to produce an accurate result in roughly 40% of the time needed by RK27. Although this last method shows a very high megaflop rate, its time step restriction makes it less efficient than OELH.

Table VII. Global performance of OELH and RK27 on grid I

	OELH (N_{st} = 80)		RK27 (N_{st} = 95)	
Mode	CPU (s)	Mflop rate	CPU (s)	Mflop rate
Scalar	20·8	34	69·9	39
Vector	2·0	366	5·1	538

Table VIII. Global errors (on grid II) for OELH and RK27 at t_1 = 10,800

Steps	Δt	OELH	RK27
10	1080	0·024	*
20	540	0·0053	*
40	270	0·0013	*
80	135	0·00028	*
160	67·5	0·00013	*
280	38·6	0·00012	*
290	37·2	0·00012	0·468
300	36·0	0·00012	0·00012
320	33·75	0·00012	0·00012

5.3. *Influence of the spatial grid*

Next we solve the problem (10), (11) discretized on grid II. It is to be expected that the refinement of the spatial grid will have an influence on the accuracy, stability and vector performance. For the end point of the integration interval we choose T_1. The results are listed in Table VIII.

For $\Delta t \rightarrow 0$ the error on both methods converges to 0·00012, which is the spatial discretization error on this fine grid. This value is four times smaller than the error obtained on grid I (see Table III), which is in agreement with the second-order spatial discretization that we have used.

With respect to stability we may conclude that the OELH method behaves as stably as it did on the 'coarse' grid. However, owing to the increased stiffness of the resulting ODE, the RK27 method is forced to take time steps that are reduced by a factor of three. This factor indicates that the terms originating from the diffusion part and from the advection part in the PDE are *both* of influence of the time step by a factor of four or two respectively.

Table IX. Vector performance of the main routines in RK27 (grid II, N_{st} = 300)

Routine	Number of calls	Average time	Accumulated percentage of CPU time spent	Mflop rate
F (derivative)	2100	$3{\cdot}9 \times 10^{-2}$	75·7	599
G (source term)	600	$3{\cdot}0 \times 10^{-2}$	92·4	574
RK (driver)	1	99·5	514	

Table X. Vector performance of the main routines in OELH (grid II, N_{st} = 80)

Routine	Number of calls	Average time (s)	Accumulated percentage of CPU time spent	Mflop rate
F (derivative)	161	$2{\cdot}5 \times 10^{-2}$	34·6	467
G (source term)	161	$1{\cdot}7 \times 10^{-2}$	57·7	526
TRI3P3 (tridiagonal solver)	160	$1{\cdot}2 \times 10^{-2}$	75·0	534
IMPL (solves implicit relation)	160	$5{\cdot}9 \times 10^{-3}$	83·1	289
JACOB (evaluates Jacobian)	160	$5{\cdot}5 \times 10^{-3}$	90·7	335
HS (driver)	1	$6{\cdot}3 \times 10^{-1}$	96·2	436

Table XI. Parallel performance of OELH estimated by altexpert

Processors	1	2	3	4	5	6	7	8
Speed-up	1·00	1·98	2·94	3·88	4·84	5·72	6·70	7·54
Processors	9	10	11	12	13	14	15	16
Speed-up	8·42	9·26	9·99	10·74	11·98	12·60	13·42	14·15

It is also of interest to see the influence of the grid refinement (i.e. longer vectors) on the performance. Tables IX and X give this information for RK27 (with $N_{st} = 300$) and OELH (using $N_{st} = 80$) respectively. Comparing Tables V and IX, we see that the performance of RK27 is slightly improved: the 'coarse' grid I is sufficiently fine to let this method run at a speed close to the optimal speed. The overall performance of this method increases from 538 to 587 Mflops. A comparison of Tables VI and X shows that the average Mflop rate of the OELH method is increased by approximately 25% (the overall performance changes from 366 to 465 Mflops).

5.4. Parellelization aspects

Finally we consider the prospects that the methods offer with respect to parallelization. In both codes we frequently encounter the situation that we have a three-times-nested loop of the following structure:

```
DO 10 k = 1, K (in horizontal space direction)
DO 10 m = 1, M (in vertical space direction)
DO 10 j = 1, J (in horizontal space direction)
   ...
   body of the loop
   ...
10 CONTINUE
```

A typical treatment of such loops on parallel/vector machines is to vectorize the inner loop (index j) and to parallelize the outer loop (index k), whereas the loop with index m is performed in sequential mode (or, if possible, collapsed with the j-loop).

We have used the autotasking facility of the Cray (specifying **cf177 -Zp** activates the Fortran preprocessor FPP to analyse the code) to get an indication of the speed-up that can be obtained for OELH when various vector pipes are used. To this end we employed the so-called **atexpert** utility.[4] Atexpert is capable of producing predictions of the performance on a dedicated system. The results obtained by this utility, when running OELH on the coarse grid, are given in Table XI. The speed-up factors clearly indicate that the OELH method efficiently exploits a multiprocessor machine like the Cray.

6. CONCLUDING REMARKS

In this paper we have discussed stablized Runger–Kutta (RK) methods and the odd–even line hopscotch (OELH) method. From a previous evaluation study[1] both types of methods turned out to be suitable candidates for the time integration of a three-dimensional numerical transport model. These methods have quite different numerical properties: the RK methods are explicit and hence have to obey a rather restrictive step size condition; since the OELH method is partly implicit, this method is in many practical situations not hampered by such a stability-induced step size restriction. The two methods also differ with respect to vectorization capabilities: because of the extremely simple nature of the RK methods, they are straightforwardly implemented on a vector processor machine. On the basis of large-scale test problems (about 10^5–10^6 unknowns), we measured a vector performance of over

500 Mflops on a one-processor Cray C98/4256 (having a clock cycle of 4·2 ns). At first sight the OELH method seems to be less suitable for vectorization, since it has to solve tridiagonal systems. However, since all these systems are uncoupled, they can be efficiently implemented on vector machines. Following this approach, which was initially proposed by Golub and Van Loan,[2] the overall performance of the OELH method turns out to be 366–465 Mflops. Of course, these high Mflop rates only show that the methods are well suited for vectorization; a good numerical solver should in addition have proper algorithmic characteristics, resulting in the ultimate goal of small CPU times. Comparing the two methods in this respect, we see that the OELH method is by far superior to the RK methods. The fact that OELH can take much larger time steps easily compensates for the lower vector speed. Furthermore, we have seen that OELH is also capable of exploiting the parallel facilities offered by multiprocessor architecture.

As explained in Section 3, we have chosen the so-called 'dummy point' approach, which means that the physical domain is enclosed by a rectangular box. In the test problem discussed in Section 5, this approach did not lead to dummy points, so that all floating point operations were really useful operations. In practical situations, however, where we have capricious geometries (e.g. in estuaries, coasts and bottom profiles), the situation is less ideal and the introduction of dummy points is unavoidable. In such cases we should consider an *effective* Mflop rate, which obviously has to be related to the number of *useful* (i.e. effective) floating point operations. However, this observation applies to any integration method, since this 'dummy point' approach is inherent to the spatial discretization and the choice of the data structures.

Summarizing, we conclude that the OELH method is an efficient method for the time integration of three-dimensional transport models because it combines in a suitable way the following properties: sufficient accuracy, sufficient stability, modest storage requirements, almost optimal vectorization and parallelization capabilities and low computational costs.

ACKNOWLEDGEMENTS

The authors are grateful to Professor P. J. van der Houwen for his constructive remarks.

This work was sponsored by the Stichting NAtionale Computerfaciliteiten (National Computing Facilities Foundation, NCF) who provided the authors with a grant from Cray Research University Grants Program.

REFERENCES

1. B. P. Sommeijer, P. J. van der Houwen and J. Kok, 'Time ingegration of three-dimensional numerical transport models', *Rep. NM-R9316*, CWI, Amsterdam, 1993; *Appl. Numer. Math.*, in press.
2. G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 2nd ed. 1989.
3. E. D. de Goede, 'Numerical methods for the three-dimensional shallow water equations on supercomputers', *Thesis*, University of Amsterdam, 1992.
4. *UNICOS Performance Utilities Reference Manual, SR-2040, Edition 7.0*, Cray Research, Inc.k 19XX.
5. M. Toro, L. C. van Rijn and K. Meijer, 'Three dimensional modelling of sand and mud transport in currents and waves', *Tech. Rep. H461*, Delft Hydraulics, Delft, 1989.
6. C. B. Vreugdenhil and B. Koren (eds), *Notes on Numerical Fluid Mechanics*, Vol. 45, *Numerical Methods for Advection–Diffusion Problems*, Vieweg, Braunschweig, 1993.
7. P. J. van der Houwen, *North-Holland Series in Applied Mathematics and Mechanics*, Vol. 19, *Construction of Integration Formulas for Initial Value Problems*, North-Holland, Amsterdam, 1977.
8. E. Hairer and G. Wanner, *Springer Series in Computational Mathematics*, Vol. 14, *Solving Ordinary Differential Equations II; Stiff and Differential–Algebraic Problems*, Springer, Berlin, 1989.
9. A. R. Gourlay, 'Hopscotch: a fast second order partial differential equation solver', *J. Inst. Math. Appl.*, **6**, 375–390 (1970).
10. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, PA, 1992.
11. B. P. Sommeijer and J. Kok, 'Implementation and performance of a three-dimensional numerical transport model', *Rep. NM-R9402*, CWI, Amsterdam, 1994.