Parallel-iterated Runge–Kutta methods for stiff ordinary differential equations

B.P. Sommeijer

Centre for Mathematics and Computer Science, Amsterdam, Netherlands

Abstract

Sommeijer, B.P., Parallel-iterated Runge–Kutta methods for stiff ordinary differential equations, Journal of Computational and Applied Mathematics 45 (1993) 151–168.

For the numerical integration of a stiff ordinary differential equation, fully implicit Runge–Kutta methods offer nice properties, like a high classical order and high stage order as well as an excellent stability behaviour. However, such methods need the solution of a set of highly coupled equations for the stage values and this is a considerable computational task. This paper discusses an iteration scheme to tackle this problem. By means of a suitable choice of the iteration parameters, the implicit relations for the stage values, as they occur in each iteration, can be uncoupled so that they can be solved in parallel. The resulting scheme can be cast into the class of Diagonally Implicit Runge–Kutta (DIRK) methods and, similar to these methods, requires only one LU factorization per step (per processor). The stability as well as the computational efficiency of the process strongly depends on the particular choice of the iteration parameters and on the number of iterations performed. We discuss several choices to obtain good stability and fast convergence. Based on these approaches, we wrote two codes possessing local error control and stepsize variation. We have implemented both codes on an ALLIANT FX/4 machine (four parallel vector processors and shared memory) and measured their speedup factors for a number of test problems. Furthermore, the performance of these codes is compared with the performance of the best stiff ODE codes for sequential computers, like SIMPLE, LSODE and RADAU5.

Keywords: Parallelism; stiffness; diagonally implicit Runge–Kutta methods; stability; convergence.

1. Introduction

Due to the never-ending demand for more speed in scientific computation, the available computer power of new architectures has tremendously increased during the last decades. This is mainly obtained by new hardware design and by a prodigious progress in micro-electronics.

However, this *hardware advancement* is not sufficient to meet the requirements as they occur in large-scale problems. The main problem in effectively exploiting this huge potential of computer power is the fact that there is very *little software* available for these machines. In order to be efficient, this software should be based on algorithms that are well tuned to the new architectures.

Since many numerical algorithms were designed for the traditional sequential computers, the existing methods are not necessarily the best. This is particularly true in the field of numerical methods for ordinary differential equations. Therefore, it is highly desirable to (re)consider these algorithms and, eventually, replace them with more suitable candidates.

Herewith, we arrive at the major aim of this paper: *the construction of new algorithms*, specifically designed for a wide class of new architectures, thus making an attempt to decrease the arrears of software with respect to hardware.

In this paper we will concentrate on numerical methods for the initial-value problem (IVP) for the ordinary differential equation (ODE), written in the autonomous form

$$y'(t) = f(y(t)), \quad 0 \leqslant t \leqslant T, \quad y \in \mathbb{R}^N, \quad f: \mathbb{R}^N \to \mathbb{R}^N. \tag{1.1}$$

In particular, we shall discuss the construction of algorithms for (1.1) that are suitable in a *parallel* environment. Although parallel computers are available now for quite a few years, it is remarkable that this area received only marginal attention and in fact is still in its infancy. A possible explanation may be that the integration of an IVP by a step-by-step process is sequentially in nature and thus offers little scope to exploit parallelism.

Nevertheless, there are some avenues: at first, there is the rather obvious way to distribute the various components of the system of ODEs amongst the available processors. This is especially effective in *explicit* methods, since they frequently need the evaluation of the right-hand side function or a given vector y, so that the components of f can be evaluated independently of one another. Following the terminology of [11], this is called *parallelism across the problem*. A more interesting approach, called *parallelism across the method*, is to employ the parallelism inherently available within the method. Concurrent evaluations of the entire function f for various values of its argument and the simultaneous solution of various (nonlinear) systems of equations are examples of parallelism across the method. Remark that this form of parallelism is also effective in case of a scalar ODE (i.e., $N = 1$ in (1.1)), whereas parallelism across the problem aims at large N-values. Also notice that both approaches can be combined because they are more or less "orthogonal". Still another approach, which could be termed *parallelism across the time*, is followed in [2]. Contrary to the step-by-step idea, a number of steps is performed simultaneously, yielding numerical approximations in many points on the t-axis in parallel. In fact, these methods belong to the class of *waveform relaxation* methods. These methods show a significant speedup provided that the number of steps is (very) large. In the present paper we will confine ourselves to parallelism across the method.

Unfortunately, many existing algorithms that perform well on a sequential computer can take hardly profit from a parallel configuration. This feature necessitates us to construct *new* methods, specifically designed for parallel execution. In doing so, it was in many cases unavoidable to introduce some redundancy in the total volume of computational arithmetic. As a consequence, it is overambitious to expect a speedup (compared with a good sequential solver) in the solution time with a factor s, if s processors are available.

In many of the methods considered in this paper, a *small* number (typically in the range from 2 to 6) of concurrent subtasks of *considerable* computational complexity can be distinguished. Consequently, (i) these methods are aiming at so-called "coarse-grain" parallelism, and (ii) communication and synchronization overhead will be small compared with CPU-time.

2. Runge–Kutta methods

The general Runge–Kutta (RK) method to proceed the numerical solution of (1.1) from t_n over a step h is given by

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i f(Y_i),$$ (2.1a)

$$Y_i = y_n + h \sum_{j=1}^{s} a_{ij} f(Y_j), \quad i = 1, \ldots, s.$$ (2.1b)

Here, $y_n \approx y(t_n)$, a_{ij}, b_i are the coefficients defining the RK method and s is called the number of stages. The quantities Y_i, the stage-values, can be considered as intermediate approximations to the solution y. An RK method is said to be *explicit* iff $a_{ij} = 0$, $j \geq i$. Otherwise, it is called an *implicit* RK (IRK) method. For the algorithms described in this paper, our starting point will always be an IRK method.

A nice feature of IRK methods is that a high order of accuracy can be combined with excellent stability properties [6]. Well-known examples of such IRKs are the Gauss–Legendre methods (order $2s$ and A-stable) and the Radau IIA methods (order $2s - 1$ and L-stable). A serious disadvantage, however, is the high cost of solving the algebraic equations defining the stage-values Y_i. Since the Y_i are coupled in general, this is a system of dimension sN, thus involving $O((sN)^3)$ arithmetic operations. This compares unfavourably with ODE solvers based on linear multistep (LM) methods, where a system of dimension N has to be solved in each step. This is the main reason that IRK methods did not receive great popularity to serve as the basis for efficient, production-oriented software. In the literature, several remedies have been proposed to reduce the amount of linear algebra per step, like Diagonally Implicit RK (DIRK) methods [1,7,8,20] and Singly Implicit RK (SIRK) methods [3,5]. However, both approaches have their own disadvantages and did not succeed in completely superseding the LM-type of methods. Another possibility to realize the excellent prospects that IRK methods offer is the use of parallel processors. In [26–28], we analyzed several parallel methods. In the next sections we will summarize the main characteristics.

Motivated by our starting point that parallelism across the method should also be effective for scalar ODEs, we will assume throughout that (1.1) is a *scalar* equation. This has the notational advantage that we can avoid tensor products in our formulation. However, the extension to systems of ODEs, and therefore to nonautonomous equations, is straightforward.

In describing the parallel methods, it will be convenient to use a compact notation for the RK method (2.1). Introducing $A = (a_{ij})$, $b = (b_i)$, $Y = (Y_i)$ and $e = (1, \ldots, 1)^T$, all of dimension s, a succinct notation of the RK method reads

$$y_{n+1} = y_n + hb^T f(Y),$$ (2.2a)

$$Y = y_n e + hAf(Y),$$

(2.2b)

where $f(v) := (f(v_j))$, for a given vector $v = (v_j)$.

3. Diagonal iteration

The main problem in the application of an IRK is the solution of (2.2b) for the stage vector Y; once this vector has been obtained, (2.2a) is straightforward. A direct treatment to solve (2.2b) (i.e., applying some form of modified Newton iteration) offers little scope to exploit parallelism, except for the linear algebra part; this aspect is not discussed here, since it is "orthogonal" to the subject of this paper, i.e., the parallel calculation of the stage vector Y. To that purpose, we introduce the iteration process

$$Y^{(j)} - hDf(Y^{(j)}) = y_n e + h[A - D]f(Y^{(j-1)}), \quad j = 1,\ldots,m.$$

(3.1a)

Here, D is a *diagonal* matrix. This is crucial, since now, given an iterate $Y^{(j-1)}$, each individual component $Y_i^{(j)}$ of the unknown iterate $Y^{(j)}$ has to be solved from an implicit relation of the form

$$Y_i^{(j)} - hd_i f(Y_i^{(j)}) - \Sigma_i = 0, \quad i = 1,\ldots,s,$$

(3.1b)

where Σ_i is the ith component of the right-hand side vector in (3.1a) and d_i is the ith diagonal entry of the matrix D. Clearly, all Σ_i depend on $Y^{(j-1)}$, but can be computed straightforwardly (even in parallel). The bulk of the computational effort involves the solution of the s equations for the components $Y_i^{(j)}$, $i = 1,\ldots,s$. However, given the Σ_i, the equations (3.1b) are *uncoupled* and can be solved in parallel. Hence, assuming that we have s processors available, each iteration in (3.1a) requires *effectively* the solution of only one implicit relation of the form (3.1b). This is especially advantageous in case of (large) systems of ODEs, because then each iteration in (3.1a) requires effectively the solution of a system of dimension N, the ODE dimension. As a consequence, the total iteration process has the effect that the solution of one system of dimension sN has been transformed into the solution of a sequence of m systems, all of dimension N. Moreover, since D is the same in all iterations, the (parallel) LU decompositions of the matrices $I - hd_i \, \partial f/\partial y$ can be restricted to the first iteration. Summing up, the total computational complexity of the iteration process is $O(N^3 + MmN^2)$, whereas a direct treatment requires $O(s^3N^3 + Ms^2N^2)$, with M the number of (modified) Newton iterations required. Since typical s-values range from 2 to 6 and because the required number of iterations m is quite modest (as we shall see in Sections 3.1 and 3.2), we now arrive at a manageable level of arithmetic. Notice that this approach is quite similar to that of a DIRK method, where also only one LU decomposition of a matrix of dimension N is required per step. In [1,8,20], A-stable DIRKs are analyzed of order p with $p - 1$ implicit stages, $2 \leqslant p \leqslant 4$. Cooper and Sayfy [7] constructed A-stable DIRKs with five implicit stages. They present a method of order 5 and could increase the order to 6 by adding one explicit stage. We are not aware of A-stable DIRKs of higher order. However, the parallel approach allows for A-stable methods of as high an order as 10 (excluding 9) (cf. Section 3.1) or even arbitrary high order (cf. Section 3.2).

A further advantage of the parallel methods is that the stage order [9] can be made higher than that of a classical DIRK method. We postpone the discussion of this aspect until Section 3.2 and first finish the discussion of the iteration scheme (3.1a).

To start the iteration (3.1a), we need the initial approximation $Y^{(0)}$. One of the possibilities to choose this vector is given by

$$Y^{(0)} - hBf(Y^{(0)}) = y_n e + hCf(y_n e).$$ (3.1c)

Here, the matrix B will be chosen either zero or of diagonal form in order to exploit parallelism (in the same way as described for (3.1a)); C is an arbitrary full matrix. Particular choices of these matrices will be discussed in the Sections 3.1 and 3.2. In the sequel, the initial approximation $Y^{(0)}$ will be referred to as the *predictor*.

If m iterations have been performed with (3.1a), then the new approximation at t_{n+1} is defined by (cf. (2.2a))

$$y_{n+1} := y_n + hb^T f(Y^{(m)}).$$ (3.2a)

Once an underlying IRK has been selected (henceforth called the *corrector*), the freedom left in the iteration process (3.1) consists of the matrices B, C and D, and the number of iterations m. With respect to the matrix D we have considered several possibilities: first of all, there is the simplest choice, which sets D equal to the zero matrix. In this case we obtain an *explicit* iteration process and, consequently, the resulting scheme is only suitable for *nonstiff* equations. This approach has received relatively much attention in the literature (see, e.g., [4,16,17,19,21,25]). Choosing the "trivial" predictor $Y^{(0)} = y_n e$, the order behaviour of the resulting algorithm can be formulated as in the following theorem (see also [16–18]).

Theorem 3.1. *The method {(3.1a) with $D = O$, (3.1c) with $B = C = O$, (3.2a)} is of order* $\min\{p^*, m + 1\}$, *where p^* is the order of the corrector (2.2).*

Notice that this method is itself an explicit RK method with $s(m + 1)$ stages. However, since $Y^{(1)} = y_n e + hAf(y_n)e$, we see that the first s stages all require the same f-evaluation and hence can be collapsed into one stage. As a result, the method defined in Theorem 3.1 can be considered as an explicit RK method with $sm + 1$ stages. On a parallel machine, however, the *effective* number of stages equals only $m + 1$ (provided that s processors are available). This means that if the number of iterations $m \leqslant p^* - 1$, then we have obtained an explicit RK method where the number of effective stages equals the order. This is an optimal result [16] and compares favourably with the situation for classical (uniprocessor) explicit RK methods, where the number of stages increases faster than linearly if we want a high order. Furthermore, we observe that the number s of required processors is minimal with respect to the order, if the generating RK method is of Gauss–Legendre type, since these methods have the highest possible order with respect to the number of stages. We do not continue the discussion of the case $D = O$, since this paper aims at *stiff* problems, leading us to *implicit* methods, i.e., to the case $D \neq O$.

Before specifying particular choices of D, we first want to discuss an aspect of the corrector which is relevant with respect to stiffness. In integrating stiff ODEs, a favourable property of the method is that it is "stiffly accurate". This notion has been introduced in [23] and means that the RK method satisfies $b^T = e_s^T A$, with e_s the sth unit vector. Hence, b^T equals the last row of A, or equivalently, the last component of the stage vector Y is an approximation to the

solution at the new step point t_{n+1}. Therefore, in case of a stiffly accurate corrector, (3.2a) will be replaced by

$$y_{n+1} := e_s^T Y^{(m)}. \tag{3.2b}$$

Another aspect worth mentioning is, that — for nonstiffly accurate correctors — the final evaluation $f(Y^{(m)})$ in (3.2a) has a bad influence on the stability of the iterated scheme [24]. This can be avoided by the following modification (see [13]): suppose that the corrector would have been solved, i.e., Y satisfies (2.2b). Then (assuming that A is nonsingular), we have

$$hf(Y) = A^{-1}[Y - y_n e]. \tag{3.3}$$

Replacing Y in this relation by $Y^{(m)}$ (that is, assuming that (3.3) is reasonably satisfied by $Y^{(m)}$) and substitution into (3.2a) leads to

$$y_{n+1} := y_n + b^T A^{-1}[Y^{(m)} - y_n e]. \tag{3.2c}$$

In case of a nonstiffly accurate corrector, the use of (3.2c) instead of (3.2a) has two consequences for the resulting method: the stability is improved by using (3.2c), since we avoid the final evaluation $f(Y^{(m)})$; on the other hand, (3.2a) is more accurate (see also the Remark following Theorem 3.2). For a stiffly accurate corrector, however, (3.2a) and (3.2c) are equivalent.

Now, we return to the discussion of the matrix D; we distinguish two cases.

(i) D is such that after a *prescribed* number of iterations the resulting method has good stability properties. This option was followed in [28] and will be outlined in Section 3.1. In this approach the order of the resulting method equals the order of the corrector and the number of iterations m is minimal to reach this order.

(ii) Another option is to *solve* the corrector and to choose D in such a way that we obtain fast convergence in the iteration process (3.1a). This strategy has been followed in [26,27] and will be the subject of Section 3.2.

In the following sections these cases will be briefly discussed; henceforth, the above Parallel Diagonally-Iterated RK methods will be denoted by PDIRK methods.

3.1. Diagonal iteration with a prescribed number of iterations

Here, we consider methods for which the number of iterations m will be fixed. As we shall see, this number is dictated by the orders of the corrector and of the predictor. To clarify this strategy, we quote a theorem from [28].

Theorem 3.2. *Let p^* be the order of the underlying corrector (2.2). Then the order p of the resulting PDIRK method {(3.1), (3.2a), (3.2b)} is given by*

$$\min\{p^*, m + r\}, \qquad \text{for all matrices } B, C \text{ and } D,$$

$$\min\{p^*, m + 1 + r\}, \qquad \text{if } (C + B)e = Ae,$$

$$\min\{p^*, m + 2 + r\}, \qquad \text{if, in addition, } BAe = A^2 e,$$

where r takes the value 1 if y_{n+1} is defined by (3.2a) (i.e., the nonstiffly accurate case) and $r = 0$ if y_{n+1} is defined by (3.2b) (the stiffly accurate case).

Furthermore, if the corrector is stiffly accurate, then the corresponding PDIRK method has the same property.

Remark. For the nonstiffly accurate case, we observe that if we change to the modification (3.2c), then r should be set to 0 in Theorem 3.2. Since in [26,28], the nonstiffly accurate methods were analyzed on the basis of (3.2a), we will confine ourselves in this overview to this choice.

Based on this theorem, we adopted in [28] the strategy to choose m such that $p = p^*$. This means that we stop iterating as soon as the order has reached the order of the corrector, since a continuation of the iteration process would not increase the order of the PDIRK method. Furthermore, in [28], we only considered correctors of Gauss–Legendre type and of Radau IIA type.

With respect to the choice of the predictor, it turned out that employing the C-matrix did not yield particular advantage; so, here we only present results for $C = O$. For the matrix B we choose either $B = O$ or $B = D$. Although B and D may be different diagonal matrices, the choice $B = D$ has the computational advantage that the LU decompositions of $I - d_i h \, \partial f / \partial y$, which are needed during the iteration (3.1a), can also be used in solving (3.1c) for $Y^{(0)}$.

The diagonal matrix D is chosen such that the resulting PDIRK method has optimal stability characteristics. Here, we distinguish two approaches: matrices D with constant and with varying diagonal entries. These variants will be discussed in the following subsections, respectively.

3.1.1. D-matrices of the form dI

In this relatively simple case we could perform a rather thorough stability analysis, using the so-called "E-polynomials" (see, e.g., [6]). In this connection we also mention the work of Wolfbrandt [29], who investigated similar stability functions. A few classes of unconditionally stable methods are listed in Table 3.1. The values of d can be found in [28].

We recall that the Gauss and Radau IIA methods are good choices to serve as a corrector, since these IRKs have a high order with respect to the number of processors required (i.e., these methods need a minimal number of stages). It is however interesting to remark that any RK method can be used as a corrector, even an *explicit* one, although in that case we have the unconventional situation that an explicit corrector is iterated by means of an implicit iteration process. For example, the PDIRK scheme resulting from iterating a pth-order explicit RK method ($p \leqslant 6$ or $p = 8$) using exactly p iterations and $B = D = dI$ can be made L-stable by choosing the appropriate d-value. However, the number of processors equals the number of stages of the explicit RK method and thus is at least p.

Table 3.1
Unconditionally stable PDIRK methods with $D = dI$

Corrector	Matrices B and D	Attainable order p	Number of effective stages	Stability
Gauss	$B = O, D = dI$	$p \leqslant 4, p = 6$	$p - 1$	A-stable
Gauss	$B = D = dI$	$p \leqslant 6, p = 8$	p	L-stable
Radau IIA	$B = O, D = dI$	$p \leqslant 6, p = 8$	p	L-stable
Radau IIA	$B = D = dI$	$p \leqslant 8, p = 10$	$p + 1$	L-stable

3.1.2. Nonconstant D-matrices

If we omit the restriction of constant elements in D, then we can save one iteration and still obtain the same order as in the previous subsection, simply by setting $B = D = \text{diag}(Ae)$ (cf. Theorem 3.2). The stability function for this case is rather complicated, so that the stability region of the methods had to be determined numerically. Some of these methods turned out to be only $A(\alpha)$-stable, however with α close to $90°$. In Table 3.2, we collect a few methods with good stability properties.

3.1.3. Numerical example on the ALLIANT FX / 4

Here, we will show the performance of an L-stable PDIRK scheme with $B = D = dI$ (cf. Table 3.1) when running on a parallel computer. Based on the four-stage Radau IIA method, we perform seven iterations to arrive at order 7. Hence, including the calculation of the predictor, this PDIRK scheme requires, effectively, eight stages per step. This method is L-stable for a range of d-values, from which we selected $d = 0.169\ 024\ 637\ 9$. This special d-value has the effect that the degree of the denominator in the (rational) stability function is *two* larger than the degree of the numerator, which causes extra damping at infinity.

We equipped this method with a provisional strategy for error control and stepsize selection. Since the PDIRK approach inherently provides a whole set of *embedded reference solutions* of lower order, a simple way to obtain an estimate for the local truncation error is given by $\| e_s^T Y^{(m)} - e_s^T Y^{(j)} \|$ for some $j < m$. Notice that this estimate does not require additional computations, since $Y^{(j)}$ is anyhow needed to proceed the iteration process. In our code we set $j = m - 1$ (recall that $s = 4$ and $m = 7$). For further details concerning the strategy, we refer to [28].

We implemented this scheme on the ALLIANT FX/4 computer (four parallel processors and shared memory) and applied it to several test problems. The goal of these tests is twofold: (i) we want to investigate to what extent the theoretical parallelization can be realized in practice; in other words, how close we can approach the ideal speedup factor on this four-processor machine; and (ii) we want to compare the performance of the code PDIRK with that of a good sequential solver. To that purpose we selected the reliable code SIMPLE of [22] which is based on an A-stable DIRK method. Its (fixed) order is 3, which is rather low. Moreover, it has stage order 1. Since many problems are more efficiently integrated if high-order formulas are available, we also included in our tests the code LSODE of [14]. This BDF-based code has formulas up to order 5 available, from which only those of first and second order are A-stable. Hence, LSODE is less robust as a *general* stiff solver, but, on the other hand, is generally accepted as a good sequential solver and enjoys considerable usage over a long period.

Table 3.2
PDIRK methods with a nonconstant D-matrix

Corrector	Attainable order p	Number of effective stages	Stability
Gauss/Radau IIA	$p \leqslant 5$	$p - 1$	Strongly A-stable
Gauss/Radau IIA	$p = 6, 7$	$p - 1$	$A(\alpha)$-stable, $\alpha > 83°$
Radau IIA	$p = 3, 5, 7$	p	$L(\alpha)$-stable, $\alpha > 89°$

Table 3.3
Performance of the codes SIMPLE, LSODE and PDIRK for problem (3.4)

Method	(Effective) number of implicit relations/step	TOL	Δ	T_1	T_4
SIMPLE	3	10^{-4}	6.5	0.63	0.85
		10^{-5}	7.8	1.38	$> T_1$
		10^{-6}	9.5	3.67	$> T_1$
LSODE	1	10^{-5}	7.4	0.35	$> T_1$
		10^{-7}	8.6	0.80	$> T_1$
		10^{-9}	10.3	1.71	$> T_1$
PDIRK	8	10^{2}	8.5	0.51	0.19
		10^{0}	11.1	1.08	0.37

One of the test problems described in [28] is the set of reaction rate equations:

$$\frac{dy_1}{dt} = -0.04\ y_1 + 10^4 y_2 y_3, \qquad y_1(0) = 1,$$

$$\frac{dy_2}{dt} = 0.04\ y_1 - 10^4 y_2 y_3 - 3.\cdot 10^7 (y_2)^2, \quad y_2(0) = 0, \qquad 0 \leqslant t \leqslant 10^8, \qquad (3.4)$$

$$\frac{dy_3}{dt} = 3\cdot 10^7 (y_2)^2, \qquad y_3(0) = 0.$$

This problem is also used in [14,22] to illustrate the codes SIMPLE and LSODE. Initially, the solution changes rapidly and small stepsizes are necessary; gradually, the problem reaches a steady state and an efficient integration requires the stepsize to be increased significantly. In a typical situation, we observed stepsizes in the range $[10^{-3}, 10^6]$. For several values of TOL (the local error bound), the results of the various codes are given in Table 3.3. Here, T_1 and T_4 denote the CPU-time (in seconds) when the program is run on one and four processors, respectively. The accuracy is measured by means of Δ, which is defined by writing the maximum norm of the global (absolute) error in the endpoint in the form $10^{-\Delta}$.

From this experiment we can conclude the following.

(i) Concerning the parallelization of the PDIRK code we observe a speedup (defined by T_1/T_4) with a factor ≈ 2.8. The main reason for not obtaining the ideal factor 4 is that the various processors needed a different number of Newton iterations to solve their "own" implicit relations. We counted the total number of Newton iterations (over the whole integration interval) for each individual processor and observed that the two extreme values differ by about 20%.

(ii) The scalar codes SIMPLE and LSODE run faster on one processor than on four. Apparently, the parallelization overhead degrades the performance. Moreover, the dimension of the ODE (3.4) is too small to take any advantage from the vectorization capabilities of the ALLIANT.

(iii) When compared with PDIRK, we see that SIMPLE needs much more time in the high-accuracy range. This is obviously due to its low order. LSODE is more efficient in this range but, when compared to PDIRK, its CPU-time is approximately four times as large to obtain 8.5 digits (absolute) precision.

(iv) Finally, we observe that the value for TOL used by PDIRK is several orders of magnitude larger than the value used by either SIMPLE or LSODE to achieve the same global error. This can be explained as follows. Due to its high order, the local truncation error of PDIRK is usually relatively small. Therefore, if crude tolerances are used, the error control mechanism signals that a large stepsize can be used in order to balance the estimated and the requested local error. On the other hand, the Newton process imposes a limitation on the stepsize. In our implementation, the Newton processes to solve for (the components of) $Y^{(0)}$ are given the value y_n as initial iterate. Unfortunately, for large values of h (as suggested by the error estimator) this initial iterate is not always inside the contraction domain for the Newton process, resulting in an adequate reduction of the stepsize. As a consequence, this high-order scheme, using a small(er) stepsize, will produce a local error which is much smaller than requested. In conclusion, for this test problem, the restriction on the stepsize imposed by the Newton process is more stringent than that imposed by the local error control, unless very small values for TOL are used. We have also integrated some *linear* ODEs (for which the convergence problems are not relevant, of course) and observed a relation between TOL and the global error similar to that of SIMPLE and LSODE.

3.2. Diagonal iteration until convergence

PDIRK methods with a fixed number of iterations, as considered in the previous subsection, are in fact special DIRK methods. It is well known [9] that DIRK methods possess a low so-called *stage order* (viz., 1) which, in general, drastically reduces the accuracy. As a matter of fact, in many stiff problems the actually observed order equals the stage order (or, sometimes the stage order + 1). As a consequence of this so-called *order-reduction* phenomenon, the relevance of methods with a high algebraic (i.e., classical) order and a low stage order is questionable. Therefore, apart from the "fixed-m-strategy" we also considered the approach where the corrector is iterated *until convergence*. This implies that we can rely on all the characteristics of the corrector, like stability and accuracy behaviour and, in particular, the stage order. For example, s-stage IRK methods of Gauss and Radau type both have stage order s. In addition, they have a very high algebraic order (superconvergence) but, as observed above, this property seems to be of minor importance in many stiff problems. Therefore, we also considered (A-stable) Newton–Cotes and Lagrange type IRKs (cf. [26]); in these (collocation) methods the superconvergence is exchanged for an increase by one of the stage order. This is obtained by adding one explicit stage to the s implicit stages. The time needed for this extra explicit stage is quite negligible compared with the time involved in solving the implicit stages. Thus, we arrive at correctors with algebraic order = stage order = $s + 1$, which are suitable for parallel iteration on an s-processor machine.

Having decided to *solve* the corrector, we can now consider (3.1a) as an *iteration* process, where "iteration" has the classical meaning. This leads us automatically to a criterion for choosing the matrix D: this matrix should be such that we have *fast convergence* in (3.1a).

In [26] it was shown that the iteration error $Y - Y^{(j)}$, in first approximation, satisfies the recursion

$$Y - Y^{(j)} = Z(z)\left[Y - Y^{(j-1)}\right], \quad j = 1,\dots,m, \quad z := h\lambda, \tag{3.5a}$$

where the iteration matrix Z is defined by

$$Z(z) := zD[I - zD]^{-1}[D^{-1}A - I].$$ (3.5b)

Here, λ denotes an approximation to the derivative $\partial f/\partial y$ and should be understood to run through the spectrum of the Jacobian matrix in case of systems of ODEs. The convergence behaviour of (3.1a) is determined by the iteration matrix Z and we have the matrix D at our disposal to obtain fast convergence.

The main difficulty in choosing D is that Z depends on z, i.e., on the problem. Therefore, we cannot expect to find a uniformly "best" D-matrix. Since we are aiming at the integration of stiff equations, we considered the influence of Z on the eigenvectors of $\partial f/\partial y$ corresponding to eigenvalues of large modulus. For $|z| \to \infty$, Z behaves as $I - D^{-1}A$. Thus a strong damping of these eigenvectors leads us to the minimization of the spectral radius of $I - D^{-1}A$. Observe that the "nonstiff" eigenvectors (corresponding to small values of $|z|$) are already damped since Z behaves as $z[A - D]$ for $|z| \to 0$. With this approach we obtained fast convergence on a broad collection of test examples (cf. [26]). However, we do not claim that this choice of D is the best possible. For example, a more sophisticated strategy might be the minimization of (some norm of) $Z(z)$ over the whole, or the "stiff part" of the left half-plane.

Another possibility could be to minimize the principal stiff error constants in the resulting PDIRK method; this option has been studied in [27]. Several other options to choose D were discussed in [26]. Many of these have been used in numerical tests, but it turned out that the behaviour of the strategy based on the minimization of the spectral radius ρ of $I - D^{-1}A$ could not be improved.

Based on this approach, methods have been constructed for $s = 2, 3, 4$ (cf. [26]). Only for $s = 2$ it is possible to determine D analytically such that $\rho(I - D^{-1}A) = 0$. For the larger values of s, the D-matrices had to be calculated numerically. The ρ-values found increase with s and are (for the several correctors) in the range $(0.004, 0.01)$ if $s = 3$ and in the range $(0.02, 0.1)$ for $s = 4$.

In [26], we also made a mutual comparison of several *stiffly accurate* correctors. It turned out that, in general, the Radau IIA and the Lagrange based methods are superior to the Lobatto IIIA and Newton–Cotes based methods. This is probably due to the fact that the first methods have damping at infinity, whereas the latter type of methods are only weakly stable at infinity. Furthermore, we also considered the *nonstiffly accurate* method based on the Gauss corrector. This method showed poor stability behaviour, but as mentioned before, this can be improved upon if we use (3.2c) instead of (3.2a). Evidently, the final iteration error $Y - Y^{(m)}$ depends on the initial error $Y - Y^{(0)}$ (see (3.5a)). In [26], only the "trivial" predictor $Y^{(0)} = y_n e$ (i.e., $B = C = O$ in (3.1c)) has been used. In [27], also the implicit variant is considered ($B = D$, $C = O$), as well as predictors that use information from the previous step. This is a natural way to increase the accuracy of the predictor, since all methods are based on the collocation principle. This implies that the stage vector $Y^{(m)}$ calculated in the preceding step defines a collocation polynomial which can be extrapolated to the present step. Needless to say that, in general, the increased accuracy of $Y^{(0)}$ results in fewer iterations.

Apart from the convergence behaviour we also studied the stability of the iterated methods for several, but fixed values of m. It can be shown that the stability functions of the methods based on the Lobatto IIIA and Newton–Cotes correctors (using $Y^{(0)} = y_n e$) are only A-acceptable in the limit, i.e., for $m \to \infty$ (cf. [26, Table 4.1]). For the Radau IIA and Lagrange based

methods the stability function of the PDIRK method is already A-acceptable for modest values of m. These m-values for various choices of $Y^{(0)}$ can be found in [27, Table 2]. However, for many problems the requirement of A-stability is unnecessarily strong and can be weakened to $A(\alpha)$-stability with α sufficiently large. It turns out that the angles α corresponding to the Radau IIA and Lagrange correctors are close to 90° after only a few iterations, especially if $Y^{(0)}$ is defined implicitly $(B \neq O)$; these α-values can be found in [27, Table 3].

3.2.1. Numerical example on the ALLIANT FX / 4

Similar to the "fixed-m-strategy" discussed in Section 3.1, we have implemented a PDIRK method based on the "minimal-spectral-radius-strategy". For the corrector, we selected the four-stage Radau IIA method of order 7; the predictor $Y^{(0)}$ is obtained from extrapolation of the collocation polynomial calculated in the preceding step. This pair is recommended in [27] as the most efficient combination.

For the error control, we calculate a reference solution of the form

$$y_{\text{ref}} = \alpha y_n + \beta_0 h f(y_n) + \sum_{i=1}^{4} \beta_i Y_i(m),$$

(3.6)

where the $Y_i^{(m)}$ are the final iterates, and the coefficients in (3.6) are determined to make this reference solution fourth-order accurate. This requirement leaves one coefficient free, say β_0, which is set to 0.1. Following an idea of Shampine, the local truncation error is estimated by

$$\text{local error} \approx \left\| \left(I - d_4 h \frac{\partial f}{\partial y} \right)^{-1} (y_{\text{ref}} - y_{n+1}) \right\|,$$

(3.7)

where d_4 is the last entry of the diagonal matrix D. The premultiplication with $(I - d_4 h \, \partial f / \partial_y)^{-1}$ in (3.7) serves to obtain a bounded estimate if $h\lambda \to \infty$ for problems of the form $y' = \lambda y$ (see also [13, p.134]). We remark that the LU factorization of $I - d_4 h \, \partial f / \partial y$ does not require additional computation, since this factorization is available from the iteration process (cf. (3.1) and the discussion following this formula). Hence, the computation of the local error is cheap.

The resulting code is termed PSODE. In contrast to the code PDIRK, where we used a fixed number of iterations, PSODE is equipped with a strategy to terminate the iteration (3.1a). The stopping criterion for this iteration is related to TOL and a test on the rate of convergence is performed in each iteration. If this test predicts that it is unlikely that convergence will be obtained within ITER_{max} (in PSODE set to 10) iterations, then the process is interrupted and restarted with a smaller stepsize.

There is however another difference between the two codes, which is of greater impact. In PDIRK, each iterate $Y^{(j)}$ in (3.1a) is solved up to machine precision by a modified Newton process (similar to the approach followed in conventional DIRK methods). Especially for small j-values, this is a waste of effort, since $\| Y - Y^{(j)} \|$ will be relatively large at the start of the iteration process. Moreover (and this is the essential difference between PDIRK methods and conventional DIRKs), $Y^{(j)}$ is no longer needed once we have calculated $Y^{(j+1)}$, because both are approximations to the exact solution at the same points.

Therefore, each implicit relation in PSODE is "solved" by just *one* (modified) Newton iteration. As a result, the number of iterations in (3.1a) to solve (2.2b) will increase, but it

turned out that the overall process is more efficient. An additional advantage of this new approach is that we now obtain a perfect load-balancing, since all processors perform exactly the same number of Newton iterations. Hence, a degradation of the performance as observed for the code PDIRK is avoided.

We have implemented the code PSODE on the ALLIANT FX/4 and applied it to a number of test problems. Again, the aim of these tests is to measure the speedup of the code on this parallel machine and, additionally, to compare its performance with a good sequential solver. Since for many problems LSODE is an efficient stiff solver on sequential machines, this code is again an obvious reference method. Furthermore, we selected the recent (sequential) code RADAU5 of [13]. This choice is motivated by the observation that it solves a Radau IIA method (viz., the three-point fifth-order one); this starting point is quite similar to that of PSODE, although the approach to obtain the Radau-solution is completely different. Since the DIRK-based code SIMPLE is of a different nature and also because of its inefficient behaviour in the high-accuracy range, we decided to cancel this code as a reference method.

In comparing the parallel code PSODE with the two sequential codes, we do *not* take into account effects originating from a possible "parallelization over the loops". By this we mean that a long loop is cut into s smaller parts which are then assigned to the s processors. In the Introduction, this effect is termed "parallelism across the problem" and can in fact be used by any ODE solver. Here we merely want to test *intrinsic* parallelism (called "parallelism across the method"). In order to exclude the effects of "parallelism across the problem", LSODE and RADAU5 are run on a single processor. In fact, the amount of intrinsic parallelism offered by LSODE and RADAU5 is very modest (see also the Remark at the end of this section).

Of course, if one is interested in "parallelism across the problem", then the sequential codes could be implemented on an s-processor machine. However, in that case a fair comparison would require assigning $4s$ processors to PSODE, since in each of the four concurrent subtasks of PSODE, the "parallelism across the problem" can equally well be exploited.

Summarizing, we may say that PSODE needs four times the number of processors given to a sequential code, simply because it possesses a four-fold amount of intrinsic parallelism. The large number of processors utilized by PSODE reflects the current tendency in parallel computing, since modern architectures — and certainly those entering the market in the coming years — have an "almost unlimited" number of processors (*massive parallelism*).

Another aspect which is of utmost importance for the performance of a stiff code is the amount of linear algebra per step, which in turn strongly depends on the dimension of the ODE. Prior to the specification of our test problem, we will briefly discuss the characteristics of the various codes with respect to this aspect.

A common feature of the three codes is that they need from time to time an LU decomposition of the matrix involved in their respective iteration processes to solve the nonlinear relations. Since the factorization of a general N-dimensional matrix requires approximately $\frac{2}{3}N^3$ arithmetic operations, this will dominate the total costs of the integration for *large-scale problems*. Here we may think of complicated problems from circuit analysis or semi-discretized (higher-dimensional) partial differential equations. In such applications, systems of ODEs with several thousands of equations are quite usual. In this connection we remark that both LSODE and PSODE deal with matrices of dimension N. Hence, it is to be expected that their mutual comparison is only marginally influenced if N increases and all other aspects are left unchanged.

Matters are different for the code RADAU5, since it has to deal with matrices of dimension $3N$. By exploiting the special structures in these matrices, Hairer and Wanner [13] are able to reduce the total work of the LU decomposition to $\frac{10}{3}N^3$ operations, thus gaining a factor 5 compared with a direct treatment, which would have required $\frac{2}{3}(3N)^3$ operations. However, this number $\frac{10}{3}N^3$ compares unfavourably with the number $\frac{2}{3}N^3$ (associated with LSODE and PSODE), and causes a serious drawback for RADAU5 when applied to large-scale problems.

To get some insight in the performance of the codes, we have applied them to a small test problem originating from circuit analysis. It was first described in [15] and extensively discussed in [10], [12, p.112]. This (stiff) system describes a ring modulator, which mixes a low-frequency and a high-frequency signal. The modulated signal is then used as input for an amplifier. The resulting system of fifteen ODEs is defined by

$$y_1' = C^{-1}\left[y_8 - 0.5\ y_{10} + 0.5\ y_{11} + y_{14} - \frac{y_1}{R}\right],$$

$$y_2' = C^{-1}\left[y_9 - 0.5\ y_{12} + 0.5\ y_{13} + y_{15} - \frac{y_2}{R}\right],$$

$$\begin{array}{ll}
y_3' = C_s^{-1}[y_{10} - g(z_1) + g(z_4)], & y_4' = C_s^{-1}[-y_{11} + g(z_2) - g(z_3)], \\[4pt]
y_5' = C_s^{-1}[y_{12} + g(z_1) - g(z_3)], & y_6' = C_s^{-1}[-y_{13} - g(z_2) + g(z_4)], \\[4pt]
y_7' = C_p^{-1}\left[-\dfrac{y_7}{R_i} + g(z_1) + g(z_2) - g(z_3) - g(z_4)\right], & y_8' = -L_h^{-1}y_1,
\end{array}$$

$$\begin{array}{ll}
y_9' = -L_h^{-1}y_2, & y_{10}' = L_s^{-1}[0.5\ y_1 - y_3 - 17.3\ y_{10}], \\[4pt]
y_{11}' = L_s^{-1}[-0.5\ y_1 + y_4 - 17.3\ y_{11}], & y_{12}' = L_s^{-1}[0.5\ y_2 - y_5 - 17.3\ y_{12}], \\[4pt]
y_{13}' = L_s^{-1}[-0.5\ y_2 + y_6 - 17.3\ y_{13}], & y_{14}' = L_t^{-1}[-y_1 + e_1(t) - 86.3\ y_{14}], \\[4pt]
y_{15}' = L_t^{-1}[-y_2 - 636.3\ y_{15}],
\end{array}$$

<div align="right">(3.8a)</div>

where

$$\begin{array}{ll}
z_1 := y_3 - y_5 - y_7 - e_2(t), & z_2 := -y_4 + y_6 - y_7 - e_2(t), \\[4pt]
z_3 := y_4 + y_5 + y_7 + e_2(t), & z_4 := -y_3 - y_6 + y_7 + e_2(t),
\end{array}$$

<div align="right">(3.8b)</div>

and the function g, which models the characteristics of the diodes, is defined by

$$g(z) := 40.672\,864\,02 \cdot 10^{-9}[\exp(17.749\,333\,2 \cdot z) - 1].$$

<div align="right">(3.8c)</div>

The signals e_1 and e_2 are defined by

$$e_1(t) := 0.5\ \sin(2 \cdot 10^3 \pi t), \qquad e_2(t) := 2\ \sin(2 \cdot 10^4 \pi t).$$

<div align="right">(3.8d)</div>

The technical parameters have been given the values $C = 16 \cdot 10^{-9}$, $R = 25\,000$, $C_p = 10^{-8}$, $R_i = 50$, $L_h = 4.45$, $L_s = 0.0005$ and $L_t = 0.002$, resulting in a heavily oscillating solution. Not yet fixed is the value of the capacity C_s. In our test, we give it the value 10^{-9}, which seems technically meaningful. It is reported [12] that small C_s-values cause serious difficulties. In the limit, i.e., on setting $C_s \equiv 0$, we end up with a differential-algebraic system. The integration interval in our test is $[0, 10^{-3}]$; the initial values are given by $y_i(0) = 0$, $i = 1,\ldots,15$. For several

Table 3.4
Performance of the codes RADAU5, LSODE and PSODE for the circuit problem (3.8)

Method	TOL	Nsteps	$\bar{m}$	Δ_{rel}	T_1	T_4
RADAU5	10^{-2}	1275	9.0	1.1	33.1	
	10^{-3}	2277	7.6	2.6	48.6	
	10^{-4}	3922	6.7	3.8	72.4	
	10^{-5}	6761	6.1	4.9	110.9	
LSODE	10^{-3}	7054	1.5	1.4	33.6	
	10^{-4}	9772	1.4	2.8	44.1	
	10^{-5}	13 266	1.4	2.9	57.7	
	10^{-6}	17 887	1.3	3.8	74.7	
	10^{-7}	23 310	1.3	4.5	93.1	
	10^{-8}	30 253	1.2	4.9	114.3	
PSODE	10^{-2}	1185	7.3	1.4	80.0	21.4
	10^{-3}	1561	7.3	3.1	104.5	27.8
	10^{-4}	2272	7.1	4.1	146.4	39.6
	10^{-5}	3437	6.9	5.2	212.1	57.7

values of the local error bound TOL the results obtained by the codes RADAU5, LSODE and PSODE are collected in Table 3.4. Again, T_1 and T_4 denote the CPU-time (in seconds) when the program is run on one and four processors, respectively. Recall that we restrict the timings for the sequential codes to T_1. The accuracy is measured by means of Δ_{rel}, which is defined by writing the maximum norm of the global (*relative*) error in the endpoint in the form $10^{-\Delta_{rel}}$. Furthermore, Nsteps denotes the number of (successful) integration steps and $\bar{m}$ stands for the average number of (effective) f-evaluations per step.

These results give rise to the following conclusions.

(i) We see that the speedup factor for PSODE (obviously defined by T_1/T_4) is approximately 3.7, which is pretty close to the "ideal" factor 4 on this machine. This factor rapidly converges to 4 if the dimension of the problem increases.

(ii) Furthermore, we observe a remarkable similarity between RADAU5 and PSODE: both codes need approximately seven f-evaluations per step; moreover, to produce the same accuracy, the required number of steps is of the same order of magnitude (for the more stringent values of TOL, the difference in the number of steps increases, which is probably due to the higher order of PSODE). There is however a striking difference between the two Radau-based codes and LSODE; this code is very cheap per step, but needs much more integration steps to produce the same accuracy. For example, to obtain a relative accuracy of about five digits, PSODE needs ≈ 3400 steps, RADAU5 twice as many, whereas for LSODE this number is nine times as large. Taking into account the computational effort per step of the various codes, the comparison with PSODE yields a double amount of time both for LSODE and RADAU5. Approximately the same ratios are observed in the low-accuracy range (say, $\Delta_{rel} = 3$).

As mentioned before, this example is only a model problem describing a small (part of an) electrical circuit, and is still far away from a real-life application. However, even for this small system of ODEs, the performance of (this provisional version of) PSODE is already superior by

a factor 2 to that of the (well-established) codes LSODE and RADAU5. Summarizing, we can say the following.

–The PSODE-approach is much more promising to serve as the basis for an efficient, "all-purpose" stiff solver than the LSODE-approach. This is due to the improved mathematical qualities, viz., the high order in combination with A-stability.

–In comparison with RADAU5, PSODE has the advantage that in large-scale problems, the (dominating) LU factorizations require a factor 5 less computational effort. In this connection we remark that a few preliminary experiments with a problem of dimension 75 reveal that the overall gain of PSODE is already more than a factor 4.

For really large-scale problems we expect that the speedup factor will be in the range 6–8, depending on the required accuracy. This number is composed of the asymptotic factor 5 coming from the algebra part and the remaining factor 1.2–1.6 originating from the higher order of PSODE.

Remark. It should be mentioned that RADAU5 offers a possibility to exploit a small amount of intrinsic parallelism. In using two processors, the total number of arithmetic operations to perform the LU decomposition can be reduced from $\frac{10}{3}N^3$ to $\frac{8}{3}N^3$. We refrained from adapting the code RADAU5 in order to exploit this feature.

4. Concluding remarks

In this paper we proposed an iterative approach to solve the stage vector equations occurring in a fully implicit Runge–Kutta method. By a suitable choice of the iteration parameters, the resulting scheme can be cast into the class of A-stable Diagonally Implicit Runge-Kutta (DIRK) methods. However, the new schemes can be given a much higher order than the classical DIRKs available in the literature. The iterated methods have the special feature that many of the implicit relations can be solved in *parallel*, which offers a great computational advantage. Moreover, because of the "DIRK-nature" of the new schemes, they require only one LU factorization (of a matrix with the ODE dimension) per step (per processor).

In this paper we discussed two different iteration stategies and, for both strategies, optimal iteration parameters are derived. For both approaches, a variable stepsize code has been implemented on an ALLIANT FX/4 computer. On the basis of two test problems, the speedup factors have been measured; these factors, which depend on the dimension of the ODE, vary between 2.5 and 3.7, which is pretty close to the "ideal" factor 4 on this machine.

Furthermore, the performance of the codes is compared with that of the best sequential stiff ODE codes: SIMPLE, LSODE and RADAU5. For a relatively simple ODE of small dimension, the parallel code is slightly more efficient than LSODE and much more efficient than SIMPLE. For a more difficult problem of a larger dimension (viz., fifteen ODEs), the parallel code needs considerably less time than the sequential codes.

Finally, it should be remarked that the parallel codes are still in a research phase and need a further tuning of their parameters on the basis of extensive testing. Furthermore, we plan to extend the codes with the facility to treat ODEs of the form $My'(t) = f(y(t))$, where M is a matrix which may be singular, resulting in a differential-algebraic system.

References

[1] R. Alexander, Diagonally implicit Runge–Kutta methods for stiff ODEs, *SIAM J. Numer. Anal.* **14** (1977) 1006–1021.

[2] A. Bellen, R. Vermiglio and M. Zennaro, Parallel ODE-solvers with stepsize control, *J. Comput. Appl. Math.* **31** (2) (1990) 277–293.

[3] K. Burrage, A special family of Runge–Kutta methods for solving stiff differential equations, *BIT* **18** (1978) 22–41.

[4] K. Burrage, The error behaviour of a general class of predictor–corrector methods, *Appl. Numer. Math.* **8** (3) (1991) 201–216.

[5] J.C. Butcher, A transformed implicit Runge–Kutta method, *J. Assoc. Comput. Mach.* **26** (1979) 731–738.

[6] J.C. Butcher, *The Numerical Analysis of Ordinary Differential Equations, Runge–Kutta and General Linear Methods* (Wiley, New York, 1987).

[7] G.J. Cooper and A. Sayfy, Semiexplicit A-stable Runge–Kutta methods, *Math. Comp.* **33** (1979) 541–556.

[8] M. Crouzeix, Sur l'approximation des équations différentielles opérationelles linéaires par des méthodes de Runge–Kutta, Ph.D. Thesis, Univ. Paris, 1975.

[9] K. Dekker and J.G. Verwer, *Stability of Runge–Kutta Methods for Stiff Nonlinear Differential Equations*, CWI Monographs 2 (North-Holland, Amsterdam, 1984).

[10] G. Denk and P. Rentrop, Mathematical models in electric circuit simulation and their numerical treatment, in: K. Strehmel, Ed., *Numerical Treatment of Differential Equations*, Teubner-Texte Math. **121** (Teubner, Leipzig, 1989) 305–316.

[11] C.W. Gear, Parallel methods for ordinary differential equations, *Calcolo* **25** (1988) 1–20.

[12] E. Hairer, C. Lubich and M. Roche, *The Numerical Solution of Differential-Algebraic Systems by Runge–Kutta Methods*, Lecture Notes in Math. **1409** (Springer, Berlin, 1989).

[13] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations, II: Stiff and Differential-Algebraic Problems*, Springer Ser. Comput. Math. **14** (Springer, Berlin, 1991).

[14] A.C. Hindmarsh, LSODE and LSODI, two new initial value ordinary differential equation solvers, *ACM / SIGNUM Newsl.* **15** (4) (1980) 10–11.

[15] E.H. Horneber, Analyse nichtlinearer RLCÜ-Netzwerke mit Hilfe der gemischten Potentialfunction mit einer systematischen Darstellung der Analyse nichtlinearer dynamische Netzwerke, Dissertation, Fachbereich Elektrotechnik, Univ. Kaiserslautern, 1976.

[16] A. Iserles and S.P. Nørsett, On the theory of parallel Runge–Kutta methods, *IMA J. Numer. Anal.* **10** (1991) 463–488.

[17] K.R. Jackson and S.P. Nørsett, The potential for parallelism in Runge–Kutta methods, Part I: RK formulas in standard form, Technical Report No. 239/90, Dept. Comput. Sci., Univ. Toronto, 1990.

[18] K.R. Jackson and S.P. Nørsett, The potential for parallelism in Runge–Kutta methods, Part II: RK predictor-corrector formulas, in preparation.

[19] I. Lie, Some aspects of parallel Runge–Kutta methods, Report No. 3/87, Dept. Math., Univ. Trondheim, 1987.

[20] S.P. Nørsett, Semi-explicit Runge–Kutta methods, Report Math. Comput. No. 6/74, Dept. Math., Univ. Trondheim, 1974.

[21] S.P. Nørsett and H.H. Simonsen, Aspects of parallel Runge–Kutta methods, in: A. Bellen, C.W. Gear and E. Russo, Eds., *Numerical Methods for Ordinary Differential Equations*, Proc. L'Aquila Conf., 1987, Lecture Notes in Math. **1386** (Springer, Berlin, 1989) 103–117.

[22] S.P. Nørsett and P.G. Thomsen, Embedded SDIRK-methods of basic order three, *BIT* **24** (1984) 634–646.

[23] A. Prothero and A. Robinson, On the stability and accuracy of one-step methods for solving stiff systems of ordinary differential equations, *Math. Comp.* **28** (1974) 145–162.

[24] L.F. Shampine, Implementations of implicit formulas for the solution of ODEs, *SIAM J. Sci. Statist. Comput.* **1** (1980) 103–118.

[25] P.J. van der Houwen and B.P. Sommeijer, Parallel iteration of high-order Runge–Kutta methods with stepsize control, *J. Comput. Appl. Math.* **29** (1) (1990) 111–127.

[26] P.J. van der Houwen and B.P. Sommeijer, Iterated Runge–Kutta methods on parallel computers, *SIAM J. Sci. Statist. Comput.* **12** (1991) 1000–1028.

[27] P.J. van der Houwen and B.P. Sommeijer, Analysis of parallel diagonally implicit iteration of Runge–Kutta methods, *Appl. Numer. Math.* **11** (1–3) (1993) 169–188.

[28] P.J. van der Houwen, B.P. Sommeijer and W. Couzy, Embedded diagonally implicit Runge–Kutta algorithms on parallel computers, *Math. Comp.* **58** (1992) 135–159.

[29] A. Wolfbrandt, A study of Rosenbrock processes with respect to order conditions and stiff stability, Ph.D. Thesis, Chalmers Univ. Technology, Göteborg, 1977.