Diagonal–Implicitly Iterated Runge–Kutta Methods on Distributed Memory Multiprocessors

Thomas Rauber Gudula Rünger *
Computer Science Department
Universität des Saarlandes
Postfach 151150
66041 Saarbrücken, Germany
+49 – 681–302–4130
FAX 49–681–302–4290
{rauber,ruenger}@cs.uni–sb.de

Abstract

We investigate the parallel implementation of the diagonal–implicitly iterated Runge–Kutta (DIIRK) method, an iteration method based on a predictor–corrector scheme. This method is appropriate for the solution of stiff systems of ordinary differential equations (ODEs) and provides embedded formulae to control the stepsize. We discuss different strategies for the implementation of the DIIRK method on distributed memory multiprocessors which mainly differ in the order of independent computations and the data distribution. In particular, we consider a *consecutive implementation* that executes the steps of each corrector iteration in sequential order and distributes the resulting equation systems among all available processors, and a *group implementation* that executes the steps in parallel by independent groups of processors. The performance of these implementations depends on the right hand side of the ODE system: For sparse functions, the group implementation is superior and achieves medium range speedup values. For dense functions, the consecutive implementation is better and achieves good speedup values.

1 Introduction

Nonlinear differential equations occur in many simulations in the natural sciences and in engineering. The numerical solution of differential equations requires a lot of computational power which may be provided by parallel machines. Parallel processing requires adequate parallel methods for the solution of differential equations. Thus, a broad area of research deals with a parallel redesign of numerical methods exploiting the inherent degree of parallelism or investigates the parallel implementation of specific simulations.

We restrict our attention to numerical methods for initial value problems (IVPs) associated with systems of first order ordinary differential equations (ODEs)

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{f}(t, \mathbf{y}(t)), \qquad \mathbf{y}(t_0) = \mathbf{y}_0, \qquad t_0 \leq t \leq t_{end} \tag{1}$$

*both authors are supported by DFG, SFB 124 and GK

and the numerical approximation of the solution $\mathbf{y} : \mathbb{R} \rightarrow \mathbb{R}^n$ on distributed memory multiprocessors. The right hand side of system (1) is a nonlinear function $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$.

Several numerical methods for the solution of partial differential equations (PDEs) have been adapted to parallel machines (see e.g. [11] [25]) but relatively little has been done about solution methods for IVPs of ODEs [1] [14] [15]. Those methods are difficult to parallelize due to their sequential nature. But the investigation of parallel methods for this class of problems is important because several extremely time consuming situations arise in practice:

- Very large systems of ODEs arise when discretizing the space dimension of time–dependent PDEs. An ODE (in the dimension of time) appears as a necessary intermediate step in the numerical solution of nonlinear time–dependent PDEs [19] [20]. The size of the ODE system depends on the discretization in space.
- The evaluation of the right hand side $\mathbf{f}$ of system (1) may be very time–consuming, e.g. if $\mathbf{f}$ is a function that depends on most of the components of its argument vector. Such functions arize when solving parabolic or hyperbolic nonlinear PDEs with Galerkin or Fourier methods [3].
- The problem may have to be solved over a very large period of time $[t_0, t_{end}]$.
- The solution of a nonlinear stiff IVP requires a solution method with good stability properties [7]. Usually those methods include the solution of a large number of implicit systems which is very expensive.

A class of solution methods called *iterated Runge-Kutta* methods have been proposed for a *parallel* solution of IVPs [18] [8] [21] [24]. Iterated Runge-Kutta methods are predictor–corrector (PC) methods based on implicit Runge–Kutta (RK) correctors, i.e. the corrector steps represent an iteration of the (implicit) basic RK–method. These methods have a large degree of inherent parallelism and are therefore very attractive for a parallel implementation. Another advantage of all iterated RK methods is that embedded solutions are provided which allow to control the stepsizes without further computational effort.

The stability properties of iterated RK methods depend on the way the corrector is iterated. A functional iteration (fixed point iteration) of an implicit RK corrector results in the IRK method. In [21] and [23] IRK methods were proposed for a parallel implementation on shared memory machines with a small number s of processors (s is the number of stages of the corrector RK–method). In [16] IRK methods has been parallelized for distributed memory machines. But because of their relatively limited region of stability those methods are only suitable for nonstiff ODEs.

In this paper, we consider the diagonal–implicitly iterated Runge-Kutta method (DIIRK) that requires the solution of a nonlinear system of equations in each iteration step. The method belongs to the class of block structured diagonal implicit Runge–Kutta (DIRK) methods which is appropriate for the integration of stiff systems [7]. A shared memory implementation of the DIIRK method is discussed in [24] for a small number s of processors. [9] discusses the linear algebra of the problem. We investigate parallel implementations of the DIIRK method on parallel machines with a distributed memory architecture and an arbitrary number of processors.

We present strategies for the parallel implementation of the DIIRK method that differ in the order of computations and in the data distributions. The algorithms take into account special properties of the DIIRK method, e.g. the stepsize control with embedded solutions and a reduction of the number of function evaluations by precomputations in the preceding corrector

iteration. In particular, we consider a *consecutive* implementation and a *group* implementation. The consecutive implementation breaks down each corrector step into dependent pieces and computes them in sequential order by distributing the resulting equation systems among all available processors. The group implementation executes the pieces in parallel by independent groups of processors.

The implementations are expressed in a *coarse grain compute/communicate* scheme. Computation phases are described in an *SPMD (single-program multiple data)* style where similar computations are executed on different portions of the problem data which are distributed among the available processors. The communication phases are described by *communication primitives* that reflect the typical data exchange of numerical problems. Both computation steps and communication primitives have the problem size and the number of processors as parameters.

We have implemented the different parallel variants of the DIIRK method on an Intel iPSC/860. The experiments take into account different numbers of processors, different dimensions of the systems and different computational effort of the right hand side $\mathbf{f}$ of the ODE system. The experiments show that the performance of the implementations depends strongly on the function $\mathbf{f}$: For sparse functions, the group implementation is much better and reaches medium range speedup values. For dense functions, the consecutive implementation is superior and reaches good speedup values.

The remaining part of this article is organized as follows: Section 2 describes the diagonal-implicitly iterated Runge-Kutta method and some characteristic properties of the DIIRK method. Section 3 develops different parallel implementations. Section 4 presents the numerical experiments on an Intel iPSC/860.

2 Diagonal–Implicitly Iterated Runge–Kutta Methods

2.1 Runge–Kutta methods

Runge–Kutta (RK) methods are one–step solution methods for IVPs of ODEs [10] [6] [7]. One step of an implicit RK method computes the next iteration vector $\mathbf{y}_{\kappa+1}$ according to the formula

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^{s} b_l \mathbf{f}(\mathbf{v}_l) \tag{2}$$

where the vectors $\mathbf{v}_l$, $l = 1, \ldots, s$, are defined by the $s \cdot n$ dimensional fully implicit system:

$$\mathbf{v}_l = \mathbf{y}_\kappa + h \sum_{i=1}^{s} a_{li} \mathbf{f}(\mathbf{v}_i) \qquad l = 1, \ldots, s \tag{3}$$

The s–dimensional vectors $\mathbf{b} = (b_1, \ldots, b_s)$ and $\mathbf{c} = (c_1, \ldots, c_s)$ and the $s \times s$ matrix $\mathbf{A} = (a_{li})$ describe the basic RK–method. The number s is called the *stage* of the RK method and h is the stepsize. The formulae (2) and (3) are given in the in stage–value notation which is appropriate for the development of correctors for the DIIRK method. (We use the convention to set vectors, e.g. $\mathbf{y}_\kappa, \mathbf{v}_l$, in bold type.)

The iteration vector $\mathbf{y}_\kappa$ represents the approximation of the solution $\mathbf{y}$ at time $(t_0 + \kappa h)$, i.e. $\mathbf{y}_\kappa = \tilde{\mathbf{y}}(t_0 + \kappa h)$, when using the pure implicit RK method for the solution of a system of ODEs. The computation of $\mathbf{y}_{\kappa+1}$ is called a *time step*.

A predictor–corrector (PC) method performs one time step by executing a number of intermediate steps. First a PC method determines an approximation with the predictor method. This initial approximation is improved in a fixed number of corrector steps where each corrector step starts with the output of the preceding corrector step. For the DIIRK method we use an s–stage, implicit, one–step RK method as corrector.

2.2 DIIRK with Fixed Number of Iterations

Iterated Runge–Kutta A PC method based on the RK method (2), (3) as corrector results in an IRK method [22] which is suitable for the solution of nonstiff systems of ODEs. For the construction of the DIIRK method we introduce a diagonal matrix $\mathbf{D}$ of dimension $s \times s$ into the equation (3). This results in the system

$$\mathbf{v}_l = \mathbf{y}_\kappa + h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i) - h d_{ll} \mathbf{f}(\mathbf{v}_l), \qquad l = 1, \ldots, s \tag{4}$$

One time step of the DIIRK methods consists of a fixed number m of iteration steps of equation (4). The initial iteration vector is provided by the predictor method. We choose a simple one–step predictor method (see [24]), the last–step–value predictor. This yields the following standard computation scheme for the DIIRK method:

$$\mathbf{v}_l^{(0)} = \mathbf{y}_\kappa \qquad l = 1, \ldots, s \tag{5}$$

$$\boxed{\textbf{Std}} \quad \mathbf{v}_l^{(j)} = \mathbf{y}_\kappa + h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) + h d_{ll} \mathbf{f}(\mathbf{v}_l^{(j)}) \quad l = 1, ..s \;\; j = 1, ..m \tag{6}$$

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^{s} b_l \, \mathbf{f}(\mathbf{v}_l^{(m)}) \tag{7}$$

One time step $\kappa \to \kappa + 1$ according to system (5), (6), (7) is called a *macrostep* of the DIIRK method. The execution of one iteration step $j \to j + 1$ of equation (6) is called a *corrector step*. The number m of corrector steps determines the convergence order of the method. The convergence order of the DIIRK method is $p^* = min(p, m + 1)$ where p is the order of the used implicit RK–method [21].

Considering all m corrector iterations of one time step, the DIIRK method is equivalent to a diagonal–implicitly RK method with block structure. This can be illustrated by the Butcher array of the method [24]:

corrector–iterations						
$j = 0$	$\mathbf{O}$					
$j = 1$	$\mathbf{A - D}$	$\mathbf{D}$				
$j = 2$	$\mathbf{O}$	$\mathbf{A - D}$	$\mathbf{D}$			
$j = 3$	$\mathbf{O}$	$\mathbf{O}$	$\mathbf{A - D}$	$\mathbf{D}$		
$\vdots$	$\vdots$	$\ddots$	$\ddots$		$\ddots$	
$j = m$	$\mathbf{O}$	$\cdots$		$\mathbf{O}$	$\mathbf{A - D}$	$\mathbf{D}$
	$\mathbf{0}^T$	$\cdots$			$\mathbf{0}^T$	$\mathbf{b}^T$

where $\mathbf{A}$ is the matrix of the RK corrector, $\mathbf{D}$ is the diagonal matrix, $\mathbf{O}$ is the $s \times s$ matrix containing 0 in every item and $\mathbf{0}^T$ is the s dimensional vector containing 0.

2.3 Implementation of the DIIRK method

For each corrector step j of a DIIRK method an implicit nonlinear system of equations has to be solved in order to get the vectors $\mathbf{v}_1^{(j)}, \ldots \mathbf{v}_s^{(j)}$. This is done by the Newton method [26].

Newton method The Newton method determines the root $z \in I\!R^n$ of a function $F : I\!R^n \to I\!R^n$ by computing a sequence of approximations $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \mathbf{z}^{(3)}, \ldots$ according to the iteration scheme:

$$\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} - DF(\mathbf{z}^{(k)})^{-1} F(\mathbf{z}^{(k)}), \qquad i = 0, 1, 2, \ldots \quad .$$

$\mathbf{z}^{(0)}$ is an initial approximation and $DF(\mathbf{z}^{(k)})$ denotes the Jacobi matrix of F at $\mathbf{z}^{(k)}$, i.e. the matrix $DF(\mathbf{z}^{(k)}) = \left(\frac{\partial F_i}{\partial \mathbf{z}_j}\right)_{i,j=1,\ldots,n} (\mathbf{z}^{(k)})$. Each iteration step k of the Newton method consists of three phases:

- the computation of the entries $\frac{\partial F_i}{\partial z_j}(\mathbf{z}^k)$ of the Jacobi matrix by a forward difference approximation

$$\frac{F_i(\mathbf{z}^{(k)} + r_j \mathbf{e}_j) - F_i(\mathbf{z}^{(k)})}{r_j} \tag{8}$$

  where $\mathbf{e}_j \in I\!R^n$ is the jth column of the unit matrix $\mathbf{I}_n$ and $r_j \in I\!R$ is a suitable interval.

- the solution of the linear system of equations $DF(\mathbf{z}^{(k)})\mathbf{y}^{(k)} = -F(\mathbf{z}^{(k)})$ with the Gaussian elimination method. An iterative method cannot be employed in the general case, because we do not necessarily have $\rho \leq 1$, for the spectral radius ρ of $DF(z^{(k)})$, i.e. the iterative methods might not converge. Also the conjugate gradient method cannot be used, because this requires $DF(z^{(k)})$ to be positive definite.

- the approximation $z^{(k)}$ is updated by $\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} + \mathbf{y}^{(k)}$.

The Newton iteration stops if the error is small enough, i.e. if $||\mathbf{y}^{(k)}|| \leq \epsilon \frac{1-L}{L}$ where L with $0 \leq L < 1$ is the *Lipschitz constant* of F and ϵ is a predefined accuracy.

Implementation of the DIIRK method The execution of the DIIRK method according to the computation scheme **Std** results in the following computational structure:

MACROSTEP–loop

- predictor computation according to equation (5)
- correctorstep–loop according to equation (6)
  - Newtonstep–loop
    - computation of the Jacobi matrix according to equation (8)
    - Gaussian elimination with pivoting, forward elimination and backward substitution
- update of the iteration vector according to equation (7)
- computation of the next stepsize according to equation (19)

The presented DIIRK method possesses several properties which we exploit for a fast implementation. Those properties include:

- An automatic stepsize control is possible without additional computational effort, because the iterations of the RK method in the corrector steps provide embedded solutions [7], see Section 2.5.

- Each of the m systems (6) of size $s \cdot n$ actually consists of s decoupled subsystems of size n each of them responsible for the computation of one vector $\mathbf{v}_l^{(j)}$, $l = 1, \ldots, s$. Because the solution of a nonlinear system of size n has computational complexity of order n^3 [13], the solution of the decoupled systems requires computational effort of order sn^3 instead of $(sn)^3$.
  For the computation of $\mathbf{v}_l^{(j)}$ we have to solve the system $F_{j,l} = 0$ with $F_{j,l} : I\!R^n \to I\!R^n$ defined as follows

$$F_{j,l}(\mathbf{z}) = \mathbf{z} - \mathbf{y}_\kappa - h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) - h d_{ll} \mathbf{f}(\mathbf{z}) \tag{9}$$

- The Jacobi matrix needed in each Newton step for solving a nonlinear implicit equation has a special shape containing the Jacobi matrix of the function $\mathbf{f}$ which is the right hand side of the ODE to be solved:

$$\frac{\partial (F_{j,l})_i}{\partial z_k} = \delta_{ik} - h d_{ll} \frac{\partial f_i}{\partial z_k}, \quad j = 1, \ldots, s.$$

- In each corrector step the number of function evaluations of $\mathbf{f}$ can be reduced when performing some precomputations in the previous corrector step, see Section 2.4. The precomputed function values can also be used for the update step (7) and the stepsize control such that both can be implemented in such a way that no further function evaluations are necessary.

In the following subsections we describe the reduction of the number of function evaluations and the stepsize control mechanism of the DIIRK method in more detail.

2.4 Reduced Number of Function Evaluations

The number of function evaluations in the corrector step $j+1$ for the computation of $v_l^{(j+1)}$, $l = 1, \ldots s$, can be reduced by exploiting the corrector step j. By a reformulation of equation (7) of corrector step j we get:

$$\mathbf{f}(\mathbf{v}_l^{(j)}) = \left(\mathbf{v}_l^{(j)} - \mathbf{y}_\kappa - h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) \right) / (h d_{ll}), \qquad l = 1, \ldots, s \tag{10}$$

which represents an alternative way for computing $\mathbf{f}$ at vector $\mathbf{v}_l^{(j)}$. All vectors used on the right hand side of equation (10) are known from corrector step j, i.e. we can compute the values of $\mathbf{fval}_l^{(j)} = \mathbf{f}(\mathbf{v}_l^{(j)})$ for $l = 1, \ldots, s$ immediately when the corrector step j is finished. Instead of (9), we now have

$$F_{j,l}(\mathbf{z}) = \mathbf{z} - \mathbf{y}_\kappa - h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{fval}_i^{(j-1)} - h d_{ll} \mathbf{f}(\mathbf{z}) \tag{11}$$

The computation of $\mathbf{f}(\mathbf{v}_l^{(j)})$ according to formula (10) not only saves computation time but also avoids an increase of the approximation error that arizes when applying $\mathbf{f}$ to $\mathbf{v}_l^{(j)}$. The solutions $\mathbf{v}_l^{(j)}$, $l = 1, \ldots, s$, of corrector step j are not the exact solutions but only good approximations determined by the Newton iteration, i.e.

$$\mathbf{v}_l^{(j)} \approx \mathbf{y}_\kappa + h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) + d_{ll} \mathbf{f}(\mathbf{v}_l^{(j)}) \qquad l = 1, \ldots, s \tag{12}$$

An application of the function $\mathbf{f}$ to this approximation in the next corrector step may increase the approximation error.

The computation scheme for one macrostep of the DIIRK method with the reduced number of function evaluations has the form:

$$\mathbf{fval}_l^{(0)} = \mathbf{f}(\mathbf{y}_\kappa) \qquad l = 1, \ldots, s \tag{13}$$

$$\mathbf{w}_l^{(j)} = h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{fval}_i^{(j-1)} \qquad l = 1, \ldots, s;\ j = 1, \ldots, m \tag{14}$$

Red

$$\mathbf{v}_l^{(j)} = \mathbf{y}_\kappa + \mathbf{w}_l^{(j)} + h d_{ll} \mathbf{f}(\mathbf{v}_l^{(j)}) \qquad l = 1, \ldots, s;\ j = 1, \ldots, m \tag{15}$$

$$\mathbf{fval}_l^{(j)} = \left(\mathbf{v}_l^{(j)} - \mathbf{y}_\kappa - w_l^{(j)} \right) / (h d_{ll}), \qquad l = 1, \ldots, s;\ j = 1, \ldots, m \tag{16}$$

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^{s} b_l \, \mathbf{fval}_l^{(m)} \tag{17}$$

2.5 Stepsize Control

For the solution of system (1) in the interval $t_0 \le t \le t_{end}$, several macrosteps are performed to approximate the solution $\mathbf{y}$ at the points $t_0, t_1, t_2, \ldots, t_{end}$ with $t_{\kappa+1} = t_\kappa + h_\kappa$. In order to achieve a good solution and to maintain a fast computation time, the stepsizes $h_0, h_1 \ldots$ have to be chosen as large as possible while guaranteeing small approximation errors.

For the problem of chosing appropriate stepsizes, [5] proposes an automatic stepsize control using two different approximations $\mathbf{y}_{\kappa+1}$ and $\tilde{\mathbf{y}}_{\kappa+1}$ for the solution $\mathbf{y}(t_{\kappa+1})$ computed with the same stepsize h. The error between those two approximations

$$error = ||\mathbf{y}_{\kappa+1} - \tilde{\mathbf{y}}_{\kappa+1}|| \tag{18}$$

and the upper bound for the solution in the interval $[t_\kappa, t_{\kappa+1}]$

$$bound = max(|\mathbf{y}_\kappa|, |\mathbf{y}_{\kappa+1}|)$$

are used to compute a new stepsize

$$h_{new} = h * min(6, max(\frac{1}{3}, 0.9 * \left(\frac{bound}{error} \right)^{1/(ord+1)})) \tag{19}$$

where ord is the minimal convergence order of the approximation methods used. The approximation vector $\mathbf{y}_{\kappa+1}$ is accepted if $error \le bound$. In this case h_{new} is used to compute $\mathbf{y}_{\kappa+2}$. Otherwise, the computation of $\mathbf{y}_{\kappa+1}$ is rejected and is repeated with stepsize h_{new}.

The DIIRK method provides several approximation solution when using the vectors $\mathbf{v}_l^{(j)}$, $l = 1, \ldots, s$, for one j, $j < m$, and equation (7)

$$\mathbf{y}^{(j)} = \mathbf{y}_\kappa + h \sum_{l=1}^{s} b_l \, \mathbf{f}(\mathbf{v}_l^{(j)}).$$

The solutions $\mathbf{y}^{(j)}$ represent embedded solutions of successively increasing order $\min(p, j+1)$ where p is the order of the basic implicit RK–method [21], [5], [12]. Usually, solutions $\mathbf{y}^{(j)}$ are

used such that the order of $\mathbf{y}_{\kappa+1}$ and $\mathbf{y}^{(j)}$ differ by 1. Therefore we choose $j = m-1$. The error is computed according to the formula:

$$\begin{aligned} error &= ||\mathbf{y}_{\kappa+1} - \mathbf{y}^{(m-1)}|| \\ &= |h| * || \sum_{l=1}^{s} b_l * \left(\mathbf{f}(\mathbf{v}_l^{(m)}) - \mathbf{f}(\mathbf{v}_l^{(m-1)}) \right) || \end{aligned} \tag{20}$$

We can also use the precomputed function evaluations for the error computation:

$$error = |h| * || \sum_{l=1}^{s} b_l * (\mathbf{fval}_l^{(m)} - \mathbf{fval}_l^{(m-1)}) || \tag{21}$$

3 Parallel implementation of the DIIRK Method

Parallel implementations of the DIIRK method are formulated in a parallel programming model that is suitable for DMMs. The processors communicate through an interconnecting network that consists of direct communication links joining certain pairs of processors. The communication is executed by explicit message passing statements.

The algorithms are expressed in a coarse grain compute/communicate scheme. The computations are performed according to the SPMD model, i.e. similar computations are executed on different portions of problem data. The distribution of the problem data among the available processors is an important part of the design of the implementation. In order to avoid data redistribution when combining different parts of the program, one has to ensure a similar distribution structure for these parts.

The data exchange is performed in a synchronous communication phase. A communication phase is expressed by one of the following communication primitives which have efficient implementations on almost all interconnection networks [2].

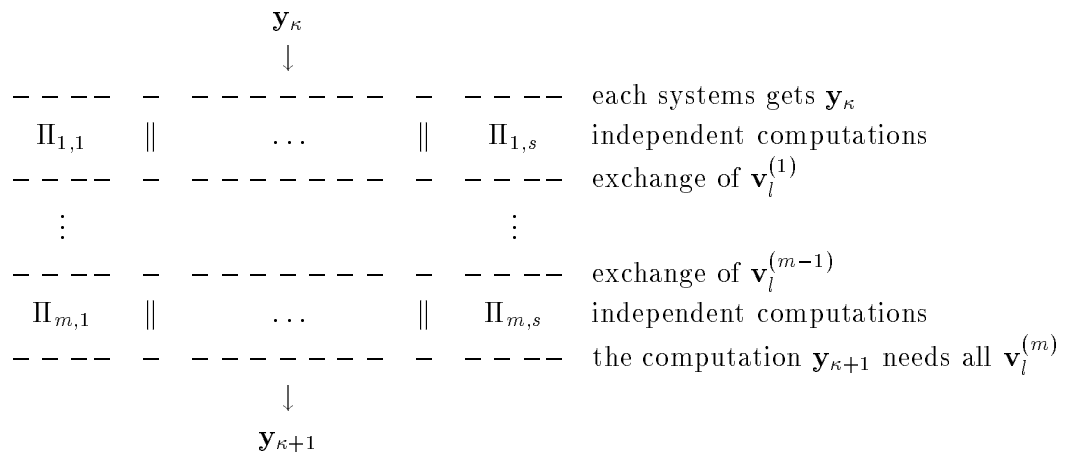
- **Single Node to Single Node:** One processor sends a message to a single other processor.
- **Single Node Broadcast and Single Node Accumulation:** A single node broadcast sends the same message from a given processor to every other processor. For a single node accumulation, a given processor receives a message from every other processor. The messages are combined by a reduction operation at each intermediate processor.
- **Single Node Scatter and Single Node Gather:** A single node scatter sends a separate message from a single processor to every other processor. The dual problem, called single node gather, collects a separate message at a given processor from every other processor without performing a reduction operation.
- **Multinode Broadcast and Multinode Accumulation:** A multinode broadcast executes a single node broadcast simultaneously for all processors. A multinode accumulation executes a single node accumulation at each processor.
- **Total Exchange:** A total exchange sends an individual message from every processor to every other processor.

3.1 Parallel algorithms

As mentioned before, in each corrector step we have to solve s independent, nonlinear subsystems each of size n instead of one system of size $s \cdot n$. The existence of independent subsystems not only

decreases the computational effort but can also be exploited for a parallel implementation. One possibility would be to exploit the special structure of the system in each step of the Newton iteration solving corrector step j. Instead, we compute the $v_i^{(j)}$, $i = 1, \ldots, s$, by solving the subsystems by a separate Newton iteration.

Let $\Pi_{j,l}$ for $l = 1, \ldots, s$ and $j = 1, \ldots, n$ denote the subsystems of one corrector step j. $\Pi_{j,l}$ is the nonlinear system $F_{j,l}(\mathbf{z}) = 0$ with $F_{j,l}$ according to equation (9). The following figure illustrates the order in which the systems $\Pi_{j,l}$ have to be solved:



The symbol ‖ indicates that Π_l and Π_r of $\Pi_l \parallel \Pi_r$ are independent and may be solved in parallel. The horizontal dashed lines indicate a data exchange that is necessary for the numerical correctness of the method.

[21] and [24] propose to compute the independent subsystems $\Pi_{j,l}$, $l = 1, \ldots, s$ in parallel on an s–processors shared memory machine (s is the number of stages and also the number of independent systems) where each processor is responsible for the solution of one system. For a general DMM with a given number p of processors, the fastest parallel implementation is not straightforward. In the following, we present two possible computation schemes for the solution of the subsystems $\Pi_{j,l}$, $l = 1, \ldots, s$, of a single corrector step:

Con The systems $\Pi_{j,l}$, $l = 1, \ldots, s$, are solved in consecutive order by *all* available processors.

Grp The systems $\Pi_{j,l}$, $l = 1, \ldots, s$, are solved in parallel by independent *groups* of processors.

The combination of the two parallel algorithms with the computation schemata **Std** and **Red**. results in four implementations:

parallel implementations	standard system **Std**	reduced system **Red**
consecutive **Con**	ConStd	ConRed
group **Grp**	GrpStd	GrpRed

In the following subsections we describe these parallel implementations in more detail and, especially, concentrate on data distribution and data exchange.

3.2 Consecutive parallel algorithm – Con

The main part of the computational work arise in the steps of the Newton iterations for solving the systems $\Pi_{j,l}$. Therefore, we choose a data distribution that ensures a good load balance in each step of the Newton iteration and avoids data distribution between them.

Data distribution Each Newton step consists mainly of the computation of the Jacobian and the application of a Gaussian elimination. For both computations a *row cyclic distribution* of the Jacobian is appropriate, i.e. the rows of the Jacobi matrices $DF_{j,l}(z^{(k)})$ are distributed such that processor q owns the rows $\{j|j \equiv q \bmod p\}$ and also the corresponding components of the right hand side $F_{j,l}(\mathbf{z}^{(k)})$. This distribution results in a good load balance for the Gaussian elimination and avoids unnecessary communication overhead.

The *square grid distribution* where all entries of the matrix are distributed cyclically for rows and columns is considered to imply an optimal load balancing when parallelizing the Gaussian elimination in isolation [4]. But in the case that the Gaussian elimination is part of the DIIRK method the row cyclic distribution results in a better global execution time for the DIIRK.

From this row cyclic distribution we can conclude the distribution of the iteration vector $z^{(k)}$ and the vector $y^{(k)}$, (see Section 2.3):

- The computation of the Jacobian may require the complete iteration vector $\mathbf{z}^{(k)}$ for the computation of each entry because the evaluation of each component f_i of the function $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ requires the complete argument vector and not only a cyclic parts of it. Therefore, the iteration vector $\mathbf{z}^{(k)}$ must be held *replicated* on all processors.
- The Gaussian elimination uses a single–node accumulation operation with a maximum reduction to determine the pivot row. The pivot row is sent to the other processors by a single–broadcast operation. The backward substitution uses a single–broadcast operation to make the computed components of the result vector available to the other processors. This means that the Gaussian elimination delivers the result vector $\mathbf{y}^{(k)}$ such that it is *replicated* to all processors.
- The update step $\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} + \mathbf{y}^{(k)}$ of the Newton method is executed by each processor for all components to make $\mathbf{z}^{(k+1)}$ available on all processors. (Details of the implementation can be found in [13]).

ConStd Figure 1 shows a pseudocode program for one macrostep of the DIIRK method executed on a DMM with p processors $P = \{q_1, \ldots, q_p\}$.

The chosen data distribution implies the following data distribution and communication.

Predictor: Each processor initializes the entire vector $\mathbf{v}_l^{(0)}$ according to equation (5). To do this, the approximation vector $\mathbf{y}_\kappa$ must be replicated on all processors.
An alternative would be that each processor initializes $\lceil n/p \rceil$ components of $\mathbf{v}_l^{(0)}$. Then a multi–broadcast operation is necessary to make $\mathbf{v}_l^{(0)}$ available to all processors. Runtime tests show that this alternative takes more time than the replicated computation because the multi–broadcast operation is more expensive than the replicated initialization, especially for larger number of processors.

Corrector: The nested loops of the corrector step are performed in consecutive order. The execution of each Newton step is distributed among all processors. The replication of the result vector $z^{(k)}$ of the Newton method results in a replication of $v_l^{(j)}$ without additional communication.

Update: The subsequent iteration vector $\mathbf{y}_{\kappa+1}$ is computed in a distributed way, i.e. each processor computes $\lceil n/p \rceil$ elements of the solution vector. To guarantee the replicated distribution for the next macrostep, the distributed pieces of $\mathbf{y}_{\kappa+1}$ are then collected by a multi–broadcast operation. To collect $\mathbf{y}_{\kappa+1}$ by a *single* multi–broadcast operation, each

```
/* predictor */
forall q ∈ P do
   for l = 1, ..., s do
      initialize v_l^(0) according to equation (5);
/* corrector */
for j = 1, ..., m do
   for l = 1, ..., s do
      solve Π_{j,l} in parallel by all processors;
/* update */
forall q ∈ P do {
   compute ⌈n/p⌉ contiguous components of y_{κ+1} = y_κ + h Σ_{l=1}^{s} b_l f(v_l^(m));
   broadcast ⌈n/p⌉ components of y_{κ+1};}
execute stepsize control in parallel;
```

Figure 1: Parallel macrostep of the DIIRK version **ConStd** for processors $P = \{q_1, \ldots, q_p\}$.

processor computes $\lceil n/p \rceil$ *contiguous* elements of $\mathbf{y}_{\kappa+1}$, i.e. a block distribution is used. This causes no problems because the corrector step delivers the vectors $\mathbf{v}_l^{(j)}$ replicated.

Stepsize control: The stepsize control is executed in a distributed way as described in [16]. In equations (20) and (21) the maximum norm is used. The value of *bound* is computed by determining the local maximum of each processor in parallel and by collecting the local maxima with a single–node accumulation operation with maximum reduction. The value of *error* is determined according to equation (20) by computing $\mathbf{f}(\mathbf{v}_l^{(m)}) - \mathbf{f}(\mathbf{v}_l^{(m-1)})$ in a distributed way and by collecting the result values with a single–node accumulation operation with maximum reduction.

ConRed In the implementation using the reduced computational scheme **Red**, the Newton method still computes the vectors $v_l^{(j)}$. But because the corrector step j needs the values $\mathbf{fval}_l^{(j-1)}$ as input instead of $v_l^{(j-1)}$, the distribution and communication of $\mathbf{fval}_l^{(j-1)}$ has to be considered.

The iteration steps of the Newton method for the computation of $\mathbf{v}_l^{(j)}$ now use the vector $\mathbf{fval}_l^{(j-1)}$ for the computation of the Jacobian. The cyclic distribution of the vectors $\mathbf{fval}_l^{(j)}$ corresponds to the cyclic computation of the Jacobian. Figure 2 shows the resulting pseudocode program.

Predictor: The vectors $\mathbf{fval}_l^{(j)}$, $l = 1, \ldots, s$, are initialized *cyclically* according to equation (13).

Corrector: The vectors $\mathbf{fval}_l^{(j)}$, $l = 1, \ldots, s$, are computed *cyclically* according to equation (16). No additional data exchange is necessary. The vectors $\mathbf{w}_l^{(j)}$ can be implemented as a single array that is overwritten after its use in equation (16).

Update: The next iteration vector $\mathbf{y}_{\kappa+1}$ is computed cyclically because the function vectors $\mathbf{fval}_l^{(j)}$ are available cyclically. To collect $\mathbf{y}_{\kappa+1}$ after its computation by a single multi–broadcast operation, each processor has to store its locally computed components in a contiguous buffer before the data exchange, see Figure 3. After the multi–broadcast the elements have to be moved to their correct positions.

Runtime tests show that the additional overhead for the buffer operations is larger than the saving in the computation time of the update step. Therefore the update step does

```
/* predictor */
forall q ∈ P do
   for l = 1,...,s do {
      initialize v_l^(0) according to equation (5);
      initialize ⌈n/p⌉ components of fval_l^(0) cyclically according to (13);}
/* corrector */
for j = 1,...,m do
   for l = 1,...,s do
      forall q ∈ P do {
         compute ⌈n/p⌉ components of w_l^(j) = h Σ_{i=1}^{s} (a_li − d_li) fval_i^(j−1) cyclically;
         solve F_{j,l} = 0 with F_{j,l} according to (11) in parallel by all processors;
         compute ⌈n/p⌉ components of fval_l^(j) cyclically according to (16);}
   /* update */
   forall q ∈ P do {
      compute ⌈n/p⌉ contiguous components of y_{κ+1} = y_κ + h Σ_{l=1}^{s} b_l f(v_l^(m));
      broadcast ⌈n/p⌉ components of y_{κ+1}; }
   execute stepsize control in parallel;
```

Figure 2: Parallel macrostep of the DIIRK version **ConRed** for processors $P = \{q_1, \ldots, q_p\}$.

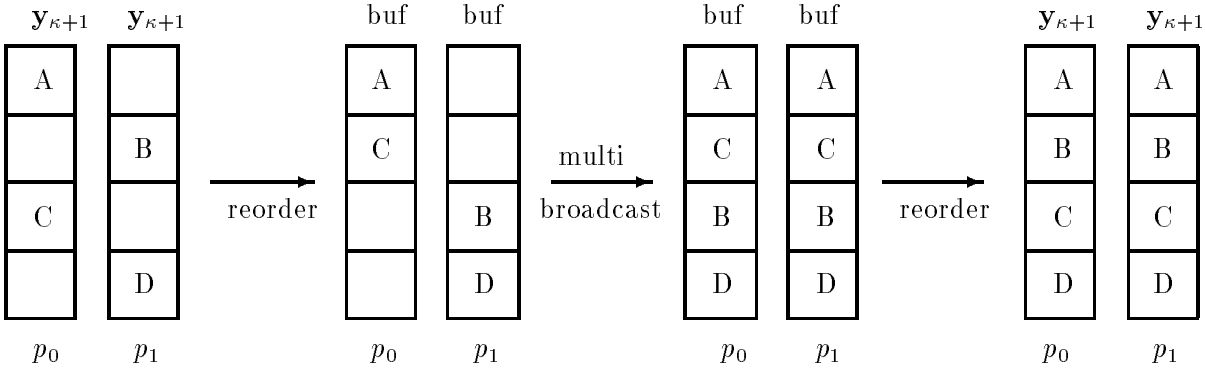


Figure 3: Collecting the distributed pieces of $\mathbf{y}_{\kappa+1}$ to avoid multiple multi–broadcast operations. The first reorder step transforms the cyclic distribution of $\mathbf{y}_{\kappa+1}$ into a block distribution of a buffer array `buf`. The multi–broadcast operation makes all components available on all processors. The second reorder step rearranges the correct order of the components.

not use the precomputed function values. Instead, each processor computes a contiguous part of the vector $\mathbf{y}_{\kappa+1}$ and executes the necessary function evaluations. (The buffering technique is successfully used in the group implementation.)

Stepsize control: The vector $\mathbf{fval}_l^{(j)}$ can also be used in the stepsize control to compute the value of error, see equation (21).

3.3 Group Parallel Computation –Grp

For the group implementation, the subsystems $\Pi_{j,l}$, $l = 1, \ldots, s$, are solved in parallel by disjoint groups of processors. We assume that the number of available processors is greater than the number of stages, i.e. $p \geq s$. The set of processors is divided into s groups $G_1, \ldots, G_s$. Group G_l contains about the same number $g_l = \lceil p/s \rceil$ or $g_l = \lfloor p/s \rfloor$ of processors. In each corrector

iteration step $j = 1, \ldots, m$ group G_l is responsible for the computation of one subvector $\mathbf{v}_l^{(j)}$, $l \in \{1, \ldots, s\}$.

Data distribution Again, the Gaussian elimination determines the data distribution of the entire macrostep. To get a good load balance, we use a group cyclic distribution, i.e. the rows of the Jacobian $DF_{j,l}$, $j = 1, \ldots, m$, are distributed cyclically among the processors of group G_l. Processor $q \in G_l$ with group index i_q, $i_q = 0, \ldots, g_i - 1$, is responsible for the computation of rows

$$rows(q) = \{i | i \equiv i_q \bmod g_i, 0 \leq i \leq g_i\}.$$

Group G_l execute the Newton iteration for the computation of $\mathbf{v}_l^{(j)}$ independently from all other groups.

The Gaussian elimination now uses communication operations that operate on groups of processors. A single–node group–accumulation is used to determine the pivot row. A group broadcast operation is used to send the pivot row to the other processors of the group. A group broadcast operations is also used in the backward elimination phase to make the computed components of the result vector available to the other processors of the group.

GrpStd The group implementation leads to the pseudocode program in Figure 4.

```
/* predictor */
forall l ∈ {1, ..., s} do
   forall q ∈ G_l do {
      initialize v_l^(0) according to equation (5);
      first processor in group: broadcast v_l^(0) to other groups; }
/* corrector */
for j = 1, ..., m do
   forall l ∈ {1, ..., s} do
      forall q ∈ G_l do
         solve F_{j,l} = 0 with F_{j,l} according to (11) in parallel by all g_l processors in group G_l;
/* update */
forall q ∈ P do {
   compute ⌈n/p⌉ contiguous components of y_{κ+1} = y_κ + h Σ_{l=1}^{s} b_l f(v_l^(m));
   broadcast ⌈n/p⌉ components of y_{κ+1}; }
execute stepsize control in parallel;
```

Figure 4: Parallel macrostep of the DIIRK version **GrpStd** for processors $P = \{q_1, \ldots, q_p\}$.

Predictor: Again, each processor initializes the entire vector $\mathbf{v}_l^{(0)}$ according to equation (5). To do this, the approximation vector $\mathbf{y}_\kappa$ must be replicated on *all* processors.

Corrector: After corrector step j, the computed vector $\mathbf{v}_l^{(j)}$, $l = 1, \ldots, s$, must be distributed to the processors of all groups because they are used in the corrector step $j + 1$ for the evaluation of $F_{j+1,l}$. This is realized by a broadcast operation executed by the first processor of each group.

Update and stepsize control: The group partitioning is only used for the computation of the corrector steps. To execute the update step in a distributed way, about the same number of components $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ of $\mathbf{y}_{\kappa+1}$ is assigned to each processor. These have to be contiguous elements to collect the different parts by a single multi–broadcast operation. The stepsize control is executed in the same way as for the consecutive computation.

GrpRed When using the reduced computation system we again have to make sure that the particular components of $\mathbf{fval}_l^{(j)}$ are available. Figure 5 shows the resulting pseudocode program.

```
/* predictor */
forall l ∈ {1,...,s} do
   forall q ∈ G_l do {
      initialize v_l^(0) according to equation (5);
      first processor in group: broadcast v_l^(0) to other groups;
      initialize ⌈n/g_l⌉ components of fval_l^(0) cyclically according to (13);
/* corrector */
for j = 1,...,m do {
   forall l ∈ {1,...,s} do
      forall q ∈ G_l do {
         compute ⌈n/g_l⌉ components of w_l^(j) = h Σ_{i=1}^{s} (a_li − d_li) fval_i^(j−1) cyclically;
         solve F_{j,l} = 0 with F_{j,l} according to (11) in parallel by all g_l processors in group G_l;
         compute ⌈n/g_l⌉ components of fval_l^(j) cyclically according to (16);}
/* update */
forall q ∈ P do {
   compute ⌈n/p⌉ components of y_{κ+1} = y_κ + h Σ_{l=1}^{s} b_l fval_l^(m) cyclically;
   broadcast ⌈n/p⌉ components of y_{κ+1} with buffer technique; }
execute stepsize control in parallel;
```

Figure 5: Parallel macrostep of the DIIRK version **GrpRed** for processors $P = \{q_1, \ldots, q_p\}$.

Predictor: Group G_l initializes vector $\mathbf{fval}_l^{(j)}$ cyclically according to equation (13).

Corrector: Group G_l computes vector $\mathbf{fval}_l^{(j)}$ cyclically according to equation (16).

After the computation of $\mathbf{fval}_l^{(j)}$, this vector must be made available to the processors of the other groups because they need it for the next corrector step. In particular, processor q needs the values $\mathbf{fval}_l^{(j)}[i]$ with $i \in \mathit{rows}(q)$ for $l = 1, \ldots, s$. Because the different groups may contain different number of processors, it is best to make the entire vector $\mathbf{fval}_l^{(j)}$ available to the processors of the other groups. This is realized by a two step communication. First, $\mathbf{fval}_l^{(j)}$ is made available to all processors of group G_l and then $\mathbf{fval}_l^{(j)}$ is distributed to the the processors of the other groups, see Figure 6. The first step can be executed by a *single* group–multi–broadcast operation, if we apply the buffer technique shown in Figure 3. The second step is realized by a broadcast operation that is executed by the first processor of each group.

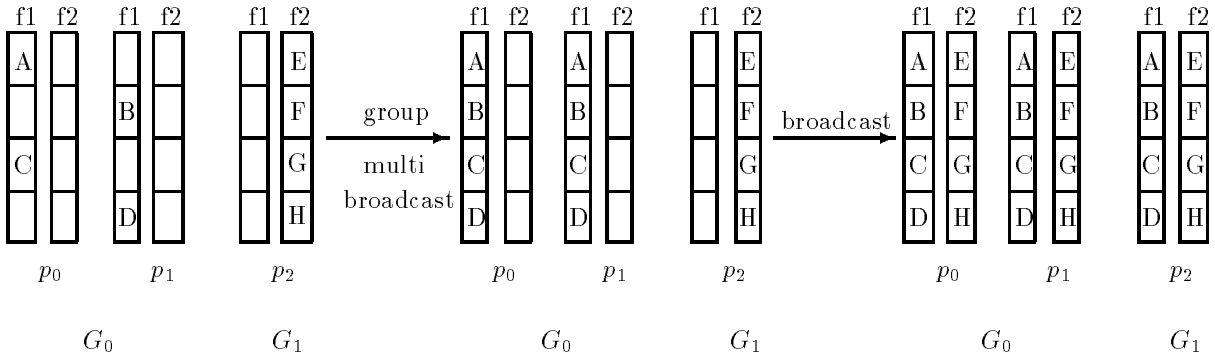


Figure 6: Making $\mathbf{fval}_l^{(j)}$ available to all processors. The figure shows two groups $G_0 = \{p_0, p_1\}$ and $G_1 = \{p_2\}$ and uses a RK–method with $s = 2$ stages. $\mathbf{fval}_l^{(j)}$ is represented as fl, $l = 1, 2$.

4 Numerical Experiments

For the implementation of parallel DIIRK methods on an Intel iPSC/860 we use a 3–stage Radau method [5] of order $p = 5$ as corrector and the simple last–step–value predictor from equation (5). Because of $p^* = min(p, m + 1)$, we execute 4 corrector iterations.

All four implementations are applied to two classes of ODEs that differ in the amount of computational work of the right hand side $\mathbf{f}$ of the ODE. We distinguish two typical cases:

- $\mathbf{f}$ has fixed evaluation costs that are independent of the system size (sparse function).
- The evaluation costs of $\mathbf{f}$ depend linearly on the system size (dense function).

Both cases may occur when solving systems of differential equations with implicit methods: The discretization of the spatial derivatives of a two–dimensional reaction-diffusion equation (Brusselator with diffusion) results in a function $\mathbf{f}$ with a constant computational effort [5]. The standard discretization of the spatial derivatives on an uniform grid with mesh size $1/(N-1)$ leads to an ODE system of dimension $n = 2N^2$. A function $\mathbf{f}$ with system size depending evaluation costs arises when solving nonlinear partial differential equations with Fourier–Galerkin methods, see e.g. [17].

Figures 7, 8, and 9 show the measured runtimes and speedup values for sparse functions for $p = 4$, $p = 8$, and $p = 16$ processors. Figures 10, 11, and 12 show the results for the case that the evaluation costs of $\mathbf{f}$ depend linearly on the system size.

The global execution times of one macrostep are denoted by t_{ConStd} t_{ConRed}, t_{GrpStd} and t_{GrpRed}. They include the runtimes for the predictor, the corrector, the update step and the stepsize control. The Newton iteration stops if the error is smaller than 10^{-4}. The precomputed function values in the implementations ConRed and GrpRed are used for the computation of the Jacobian, for the update step, and for the stepsize control. The given speedup values are obtained by comparing the parallel global execution times with the global execution time of a sequential program that is running on a single processor.

4.1 Observations and Interpretations

The experiments with different parallel algorithm (consecutive order or groups), different computation schemata (standard or reduced function evaluations), different classes of the right hand

n	t_{ConStd}	t_{ConRed}	t_{GrpStd}	t_{GrpRed}
18	1.02	0.91	0.38	0.28
72	9.49	7.54	6.20	3.43
162	31.19	30.40	42.28	23.51
242	95.23	75.65	100.51	72.83
338	204.40	166.44	231.04	178.42
512	581.84	495.20	738.66	611.54

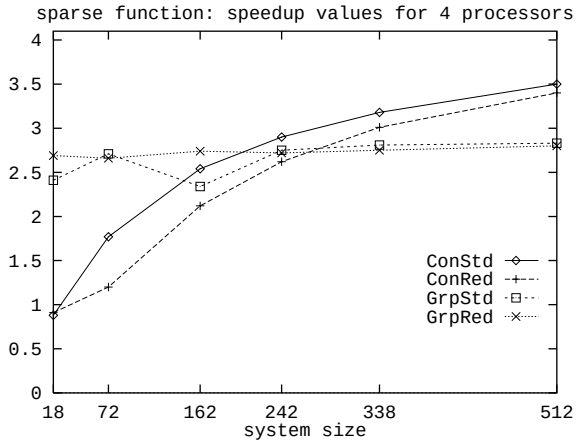


Figure 7: Measured running times in seconds and speedup values for one macrostep of the DIIRK method for $p = 4$ processors. Sparse right hand side $\mathbf{f}$.

n	t_{ConStd}	t_{ConRed}	t_{GrpStd}	t_{GrpRed}
18	1.35	1.15	0.47	0.40
72	10.86	9.46	4.30	2.90
162	37.63	33.45	20.22	14.33
242	88.96	73.99	52.81	39.87
338	176.46	145.99	116.84	93.43
512	433.53	382.65	367.83	307.29

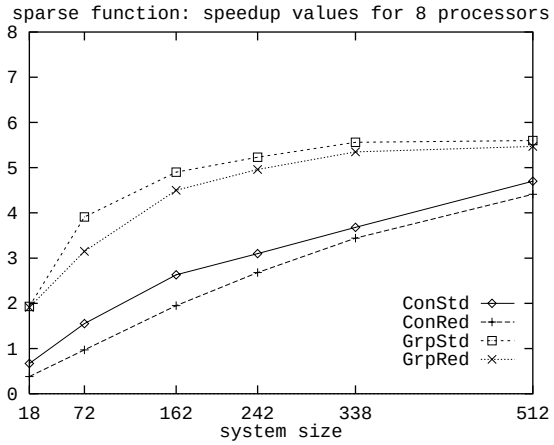


Figure 8: Measured running times in seconds and speedup values for one macrostep of the DIIRK method for $p = 8$ processors. Sparse right hand side $\mathbf{f}$.

n	t_{ConStd}	t_{ConRed}	t_{GrpStd}	t_{GrpRed}
18	3.26	2.04	0.55	0.53
72	18.09	16.75	4.02	3.39
162	57.28	54.29	14.97	12.54
242	118.18	111.80	34.94	29.54
338	212.80	203.20	71.75	62.81
512	491.95	464.70	167.56	141.71

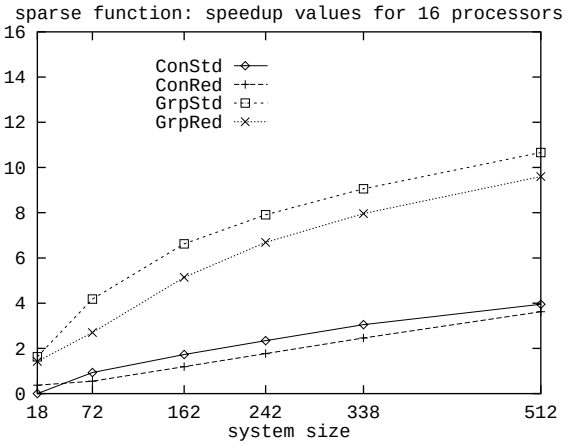


Figure 9: Measured running times in seconds and speedup values for one macrostep of the DIIRK method for $p = 16$ processors. Sparse right hand side $\mathbf{f}$.

n	t_{ConStd}	t_{ConRed}	t_{GrpStd}	t_{GrpRed}
50	6.46	2.52	7.58	8.71
100	52.30	16.55	73.43	40.29
150	171.59	50.25	244.54	132.17
200	394.68	112.14	575.97	309.42
250	789.92	219.06	1124.82	599.62
300	1347.41	369.87	1937.40	1031.44
350	2146.98	583.58	3075.28	1631.20
400	3183.93	860.28	4592.96	2429.02
450	4523.34	1222.14	6538.10	1726.31
500	6208.66	1613.38	8972.88	2365.86

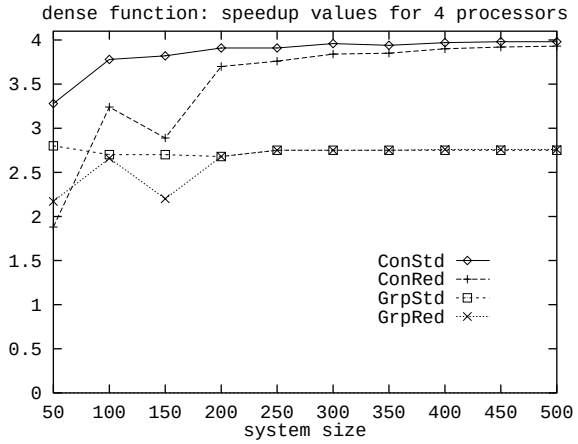


Figure 10: Measured running times in seconds and speedup values for one macrostep of the DIIRK method for $p = 4$ processors. Dense right hand side $\mathbf{f}$.

n	t_{ConStd}	t_{ConRed}	t_{GrpStd}	t_{GrpRed}
50	4.85	5.65	3.63	1.27
100	31.64	13.33	31.85	9.56
150	94.20	33.78	103.58	29.40
200	210.40	69.54	242.84	67.17
250	419.45	130.40	477.83	128.89
300	707.36	213.21	811.74	219.78
350	1106.08	325.44	1289.18	346.64
400	1633.43	472.85	1924.58	519.95
450	2343.37	667.83	2233.81	729.18
500	3187.78	1232.35	3752.58	998.15

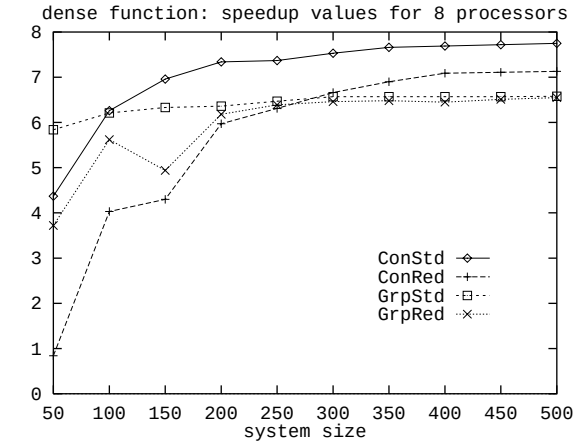


Figure 11: Measured running times in seconds and speedup values for one macrostep of the DIIRK method for $p = 8$ processors. Dense right hand side $\mathbf{f}$.

n	t_{ConStd}	t_{ConRed}	t_{GrpStd}	t_{GrpRed}
50	5.34	4.16	2.35	1.34
100	25.65	15.41	16.51	7.55
150	65.17	33.33	52.02	17.11
200	133.86	60.57	119.91	36.70
250	246.92	102.73	231.84	68.01
300	403.12	155.93	397.07	113.97
350	617.32	226.92	624.57	177.01
400	897.98	317.57	933.85	260.46
450	1290.81	438.12	1327.06	366.60
500	1737.39	575.21	1816.62	498.46

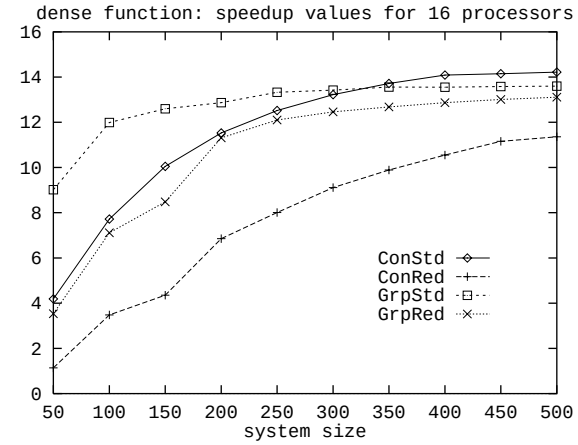


Figure 12: Measured running times in seconds and speedup values for one macrostep of the DIIRK method for $p = 16$ processors. Dense right hand side $\mathbf{f}$.

side **f** and different numbers of processors show that it is not obvious what parallel implementation should be preferred. But several observations concerning the runtime and speedup values can be made on the experiments.

Standard/reduced computation scheme Although the reduced scheme causes more communication in a parallel implementation, the global execution time is considerably reduced if the precomputed function values are used. Depending on the system size and the number of processors, the precomputation of the function values reduces the global execution time by

- 10–20% for sparse functions
- 40–75% for dense functions

The effect is especially large for dense functions, because the global execution time is dominated by the computation time of the Jacobian.
Using the precomputed function values for the stepsize control and the update step of the DIIRK method has only a very limited effect on the global execution time because these operations are only executed once for each macrostep.

The speedup values for the variants using scheme **Red** are always smaller than for the associated standard version because the contribution of the computational work to the global execution time is reduced.

Consecutive/group parallel algorithm The runtimes of the group implementation **Grp** are getting better with increasing number of processor compared with the consecutive implementation **Con**. The effect varies for dense/sparse function with the system size, i.e.

- for sparse functions and large system sizes **Grp** is much better than **Con**
- for dense functions **Grp** is only better than **Con** if the reduced variant is considered.

The group implementation **Grp** has a smaller communication overhead than the consecutive implementation **Con** because the group broadcast operations only involve the processors of the same group and use therefore less communication time.

Efficiency The efficiency $speedup/p$ of the four implementations mainly depend on the application but also on the number of processor. The application of dense function result in good speedup values while the speedup values for sparse functions are not satisfactory. A loss of efficiency be can be observed in both cases.

- For the consecutive implementation **Con** the loss of efficiency is mostly caused by communication overhead, not by a load imbalance. The load imbalance is small, if the system size is large compared to the number of processors. In this case, the equations of the system can be distributed quite evenly among the processors. The communication overhead is increasing with the number of processors because the costs of the broadcast operations is increasing. This can be especially observed for sparse functions, see Figure 8 and 9.
- For the group implementation **Grp** the loss of efficiency is caused by communication overhead and load imbalance. The impact of the load imbalance is large for small numbers of processors if the groups contain different number of processors. This is the case in Figures 7 and 10 for $p = 4$ processors and $s = 3$. Here, groups G_1 and G_2 contain one processor each and group G_3 contains two processors.

Sparse functions The runtime and speedup values of the four implementations vary with increasing number of processors. For $p = 4$ we have runtimes

$$t_{ConRed} < t_{ConStd} < t_{GrpRed} < t_{GrpStd}$$

which change to

$$t_{GrpRed} < t_{GrpStd} << t_{ConRed} < t_{ConStd}$$

for $p = 16$. Only for $p = 4$ processors, the consecutive implementation is slightly better than the group implementation because of the large load imbalance of the latter one, see Figure 7. For larger numbers of processors, the group implementation reaches global execution times that are much better than for the consecutive implementation.

The consecutive implementation **Con** only reaches limited speedup values that are not increasing with the number of processors, see Figures 8 and 9. This is caused by a large communication overhead increasing with the number of processors. The communication overhead is caused by the Gaussian elimination dominating the computation of the Jacobian. For larger number of processors, the group implementation **Grp** reaches speedup values that are much better than for the consecutive implementation. The reason for this lies in the smaller communication overhead for the Gaussian elimination and in the fact that the load imbalance is getting smaller for increasing number of processors.

Dense functions For larger system sizes, the parallel implementations using system **Red** have runtimes which are considerably smaller than the runtimes of the standard scheme **Std**, i.e. $t_{Red} << t_{Str}$. The consecutive implementation **Con** has always smaller global execution times than the group implementation **Grp**, i.e. $t_{ConStd} < t_{GrpStd}$. In this case, the load imbalance of the group implementation has a larger impact than the communication overhead of the consecutive implementation. The communication overhead is decreasing with increasing system sizes because the computation of the Jacobian is dominating. Only for small systems and larger number of processors, the additional communication overhead of the consecutive implementation is larger than the load imbalance of the group implementation.

But the global execution times for the reduced versions change with increasing numbers of processor. For $p = 4$ we have runtimes

$$t_{ConRed} < t_{GrpRed} << t_{ConStd} < t_{GrpStd}$$

which change to

$$t_{GrpRed} < t_{ConRed} << t_{ConStd} < t_{GrpStd}$$

for $p = 16$.

The speedup values for the consecutive implementations **Con** are better than for the group implementations **Grp** but the difference decreases with increasing numbers of processors.

5 Conclusions

Although IVPs for ODEs are widely considered to be inherently sequential or at best to have a small degree of parallelism, there exist algorithms for solving systems of ODEs with a large potential of parallelism. In this article, we considered the diagonal–implicitly iterated Runge–Kutta methods and have shown that they can be successfully implemented on distributed memory multiprocessors.

We have presented four parallel implementations of the DIIRK method which have been realized on the Intel iPSC/860. The parallel implementations are representative approaches that result from a combination of different computation scheme (the original version **Std** and an improved version *Red*) and different parallel algorithms **Con** and **Grp** specifying the order of computation and the data distribution.

The parallel implementations were applied to two classes of ODEs that differ in the computational amount for computing the right hand side **f** of the ODE. It was not obvious what parallel version would be the best for a specific ODE. The result of the experiments confirm that the performance of these implementations strongly depend on the application and the number of available processor.

For dense functions and large systems, the consecutive algorithm results in smaller execution times than the group algorithm. For small systems, the group implementation is slightly better than the consecutive implementation. Both implementation reach good speedup values.
For sparse functions, the group implementation has smaller execution times because the communication overhead is smaller than for the consecutive implementation. The speedup values of the consecutive implementation are only satisfactory for $p = 4$ processors whereas the group implementation reaches medium range speedup values also for larger numbers of processors.

References

[1] A. Bellen, R. Vermiglio, and M. Zennaro. Parallel ODE–Solvers with Stepsize Control. *Journal of Computational and Applied Mathematics*, 31:277–293, 1990.

[2] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computing.* Prentice Hall, New York, NY, 1988.

[3] R. Dautray and J.-L. Lions. *Mathematical Analysis and Numerical Methods for Science and Technology*, volume 4. Springer, 1990.

[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[5] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems.* Number 8 in Springer Series in Computational Mathematics. Springer–Verlag, Berlin, 1987.

[6] E. Hairer, S.P. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems.* Springer Series in Computational Mathematics. Springer–Verlag, Berlin, 1993.

[7] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential–Algebraic Problems.* Number 14 in Springer Series in Computational Mathematics. Springer–Verlag, Berlin, 1991.

[8] A. Iserles and S.P. Nørsett. On the Theory of Parallel Runge–Kutta Methods. *IMA Journal of Numerical Analysis*, 10:463–488, 1990.

[9] K. Potma and W. Hoffmann. Boosting the Performance of the Linear Algebra part in an ODE solver for shared Memory Systems. *Future Generation Computer Systems*, 10(2):315–319, 1994.

[10] L.F. Shampine and H.A. Watts and S.M. Davenport. Solving Nonstiff Ordinary Differential Equations – the State of the Art. *SIAM Review*, 18(3):376–411, 1976.

[11] J.M. Ortega and R.G. Voigt. Solution of Partial Differential Equations on Parallel Computers. *SIAM Review*, 27:149–240, 1985.

[12] P.J. Prince and J.R. Dormand. High order embedded Runge-Kutta formulae. *J. Comp. Appl. Math.*, 7(1):67–75, 1981.

[13] T. Rauber and G. Rünger. Solving ODEs on Distributed Memory Multiprocessors. Technical Report 19–93, University Saarbrücken, September 1993.

[14] T. Rauber and G. Rünger. Hypercube Implementation and Performance Analysis for Extrapolation Methods. In *Proceedings of the CONPAR'94*, pages 265–276, Linz, Austria, 1994.

[15] T. Rauber and G. Rünger. Load Balancing for Extrapolation Methods on Distributed Memory Multiprocessors. In *Proceedings of the PARLE'94*, pages 277–288, Athen, Greece, 1994.

[16] T. Rauber and G. Rünger. Iterated Runge–Kutta Methods on Distributed Memory Multiprocessors. In *3nd Euromicro Workshop on Parallel and Distributed Processing*, 1995.

[17] G. Rünger. Über ein Schrödinger-Poisson-System. PhD Thesis, Köln, 1989.

[18] S. P. Nørsett and H. H. Simonsen. Aspects of Parallel Runge–Kutta methods. In *Numerical Methods for Ordinary Differential Equations*, volume 1386 of *Lecture Notes in Mathematics*, pages 103–117, 1989.

[19] H.W. Tam. *Parallel Methods for the Numerical Solution of Ordinary Differential Equations*. Report No. UIUCDCS–R–89–1516, University of Illinois at Urbana–Champaign, Department of Computer Science, 1989.

[20] H.W. Tam. One–Stage Parallel Methods for the Numerical Solution of Ordinary Differential Equations. *SIAM Journal on Scientific and Statistical Computing*, 13(5):1039–1061, 1992.

[21] P.J. van der Houwen and B.P. Sommeijer. Parallel Iteration of high–order Runge–Kutta Methods with stepsize control. *Journal of Computational and Applied Mathematics*, 29:111–127, 1990.

[22] P.J. van der Houwen and B.P. Sommeijer. Parallel ODE Solvers. In *International Conference on Supercomputing*, pages 71–81, 1990.

[23] P.J. van der Houwen and B.P. Sommeijer. Iterated Runge–Kutta Methods on Parallel Computers. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1000–1028, 1991.

[24] P.J. van der Houwen, B.P. Sommeijer, and W. Couzy. Embedded Diagonally Implicit Runge–Kutta Algorithms on Parallel Computers. *Mathematics of Computation*, 58(197):135–159, January 1992.

[25] S. Vandewalle, R. van Driessche, and R. Piessens. The Parallel Performance of Standard Parabolic Marching Schemes. *International Journal of High Speed Computing*, 3(1):1–29, 1991.

[26] W. Liniger and R.A. Willoughby. Eficient Integration Methods for Stiff Systems of Ordinary Differential Equations. *SIAM Journal of Applied Analysis*, 7:47–66, 1970.