Comparing Task and Data Parallel Execution Schemes for the DIIRK Method

THOMAS RAUBER GUDULA RÜNGER *

Computer Science Dep., Universität des Saarlandes, 66041 Saarbrücken, Germany

Abstract. We investigate the parallel implementation of the diagonal–implicitly iterated Runge–Kutta method, an iteration method which is appropriate for the solution of stiff systems of ordinary differential equations. We discuss different strategies for the implementation of the method on distributed memory multiprocessors, which mainly differ in the data distribution and the order of independent computations. 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.

1 Introduction

We consider numerical methods for the solution of initial value problems (IVPs) associated with systems of first order ordinary differential equations (ODEs)

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

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

A class of solution methods called *iterated Runge-Kutta* methods have been proposed for a *parallel* solution of IVPs [6, 3, 8]. 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, therefore, they are very attractive for a parallel implementation. 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 [7, 8], 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 [4], IRK methods have been parallelized for DMMs. 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 (DIIRK) method which is appropriate for the integration of stiff systems [2, 8]. We investigate parallel implementations of the DIIRK method on DMMs with an arbitrary number

* both authors are supported by DFG

of processors. We present strategies for the parallel implementation of the DIIRK method that differ in the data distributions and the order of computations. 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 independent 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.

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 for 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 Method

One time step of the DIIRK method to compute the next approximation vector $\mathbf{y}_{\kappa+1}$ consists of a fixed number m of iteration steps, each computing s stage vectors $\mathbf{v}_l^{(j)}$ for $l = 1, \ldots, s$ and $j = 1, \ldots, m$. The initial iteration vector is provided by the predictor method. Choosing a simple one–step predictor method yields the following standard algorithm **Std** for the DIIRK method:

$$\mathbf{v}_l^{(0)} = \mathbf{y}_\kappa \qquad l = 1, \ldots, s \tag{2}$$

$$\mathbf{v}_l^{(j)} = \mathbf{y}_\kappa + h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{f}(\mathbf{v}_i^{(j-1)}) + h d_{ll} \mathbf{f}(\mathbf{v}_l^{(j)}) \quad l = 1, ..s \quad j = 1, ..m \tag{3}$$

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^{s} b_l \, \mathbf{f}(\mathbf{v}_l^{(m)}) \tag{4}$$

One time step $\kappa \rightarrow \kappa+1$ according to system (2), (3), (4) is called a *macrostep*. The execution of one iteration step $j \rightarrow j+1$ of (3) is called a *corrector step*. The number m of corrector steps determines the convergence order of the method. For each corrector step j, 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 as described in [5]. Each step of the Newton method includes the computation of a Jacobi matrix by forward difference approximations and the solution of a linear equation system by the Gaussian elimination. 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 (4) of corrector step j we get the following reduced algorithm **Red**:

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

$$\mathbf{w}_l^{(j)} = h \sum_{i=1}^{s} (a_{li} - d_{li}) \mathbf{fval}_i^{(j-1)} \qquad l = 1, \ldots, s;\ j = 1, \ldots, m \tag{6}$$

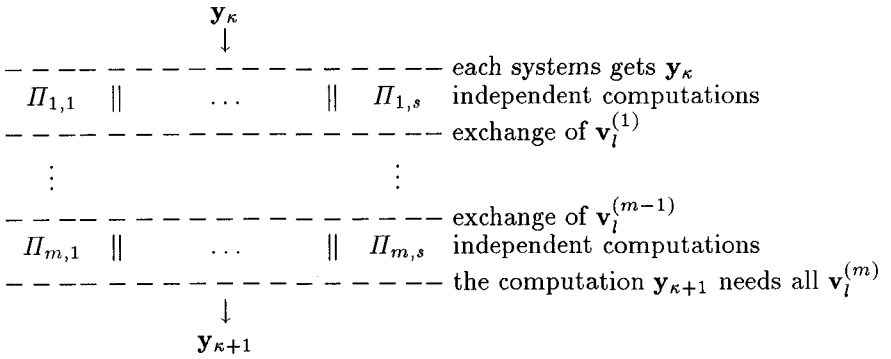
$$\mathbf{v}_l^{(j)} = \mathbf{y}_\kappa + \mathbf{w}_l^{(j)} + h d_{ll} \mathbf{f}(\mathbf{v}_l^{(j)}) \qquad l = 1, \ldots, s;\ j = 1, \ldots, m \tag{7}$$

$$\mathbf{fval}_l^{(j)} = \left(\mathbf{v}_l^{(j)} - \mathbf{y}_\kappa - w_l^{(j)}\right)/(h d_{ll}), \qquad l = 1, \ldots, s;\ j = 1, \ldots, m \tag{8}$$

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^{s} b_l \, \mathbf{fval}_l^{(m)} \tag{9}$$

3 Parallel Implementation of the DIIRK Method

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 these independent subsystems do not only decrease the computational effort but can also be exploited for a parallel implementation. We compute $v_l^{(j)}$, $l = 1, \ldots, s$, by solving the subsystems by a separate Newton iteration. Let $\Pi_{j,l}$ for $l = 1, \ldots, s$ and one corrector step j denote the subsystems of one corrector step j. $\Pi_{j,l}$ is the nonlinear system $F_{j,l}(\mathbf{z}) = 0$ with $F_{j,l}$ computed from Equation (3). 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 $\Pi_l \,||\, \Pi_r$ are independent and may be solved in parallel. The horizontal dashed lines indicate a data exchange. 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: (a) The consecutive computation scheme **Con** solves the systems $\Pi_{j,l}$, $l = 1, \ldots, s$, in consecutive order by *all* available processors. (b) The group computation scheme **Grp** solves the systems $\Pi_{j,l}$, $l = 1, \ldots, s$, in parallel by independent, disjoint *groups* of processors. The combination of the two parallel algorithms **Std** and **Red** with the computation schemes **Con** and **Grp** results in four implementations: **ConStd**, **ConRed**, **GrpStd**, and **GrpRed**. In the following subsections we describe these parallel implementations in more detail.

```
/* predictor */
forall q ∈ P do
  for l = 1,...,s do
    initialize v_l^(0) according to Equation (2);
/* corrector */
for j = 1,...,m do
  for l = 1,...,s do
    solve F_{j,l} = 0 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};}
```

Fig. 1. Parallel macrostep of the DIIRK version **ConStd**.

3.1 Consecutive parallel algorithm – Con

For the Newton iteration, a *row cyclic distribution* of the Jacobian is appropriate. The iteration vector for the Newton method is held replicated on all processors. This distribution results in a good load balance for the Gaussian elimination and avoids unnecessary communication overhead. 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 of the Newton method is executed by each processor for all components to make the new iteration vector available on all processors [5].

ConStd: Figure 1 shows a pseudocode program for one macrostep of the DIIRK method executed on a DMM with p processors $P = \{q_1, \ldots, q_p\}$. In the predictor step, each processor initializes the entire vector $\mathbf{v}_l^{(0)}$ according to (2). To do this, the approximation vector $\mathbf{y}_\kappa$ must be replicated on all processors. In the corrector step, 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 of the Newton method results in a replication of $v_l^{(j)}$ without additional communication. In the update step, 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 processor computes $\lceil n/p \rceil$ *contiguous* elements of $\mathbf{y}_{\kappa+1}$, i.e., a block distribution is used.

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

```
forall q ∈ P do
  for l = 1,...,s do {
    initialize v_l^(0) according to Equation (2);
    initialize ⌈n/p⌉ components of fval_l^(0) cyclically according to (5);}
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 in parallel by all processors;
      compute ⌈n/p⌉ components of fval_l^(j) cyclically according to (8);}
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}; }
```

Fig. 2. Parallel macrostep of the DIIRK version **ConRed**.

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. For the predictor step, the vectors $\mathbf{fval}_l^{(j)}$, $l = 1, \ldots, s$, are initialized *cyclically* according to (5). For the corrector step, the vectors $\mathbf{fval}_l^{(j)}$, $l = 1, \ldots, s$, are computed *cyclically* according to (8). 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 (8). For the update step, 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.

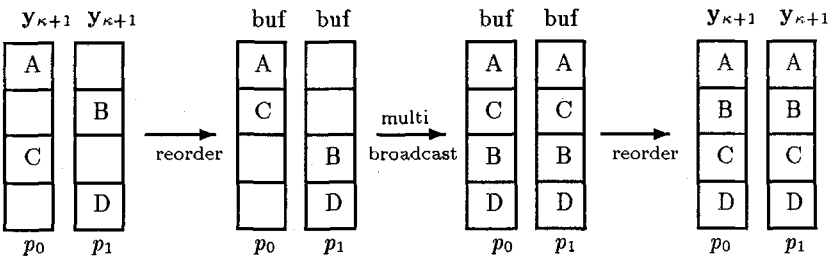


Fig. 3. Collecting the distributed pieces of $\mathbf{y}_{\kappa+1}$ to avoid multiple multi-broadcast operations. The first reorder step transforms the cyclic distribution of $\mathbf{y}_{\kappa+1}$ into a block distribution of a buffer array buf. The multi-broadcast operation makes all components available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

```
forall l ∈ {1,...,s} do
  forall q ∈ G_l do {
    initialize v_l^(0) according to Equation (2);
    first processor in group: broadcast v_l^(0) to other groups; }
for j = 1,...,m do
  forall l ∈ {1,...,s} do
    forall q ∈ G_l do
      solve F_{j,l} = 0 in parallel by all g_l processors in group G_l;
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}; }
```

Fig. 4. Parallel macrostep of the DIIRK version **GrpStd**.

3.2 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 $\mathbf{v}_l^{(j)}$, $l \in \{1, \ldots, s\}$.

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 of size g_l. Processor $q \in G_l$ with group index i_q, $i_q = 0, \ldots, g_l - 1$, is responsible for the computation of rows $rows(q) = \{i | i \equiv i_q \bmod g_l, 0 \leq i \leq g_l\}$. Group G_l executes 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.

GrpStd The group implementation leads to the pseudocode program in Figure 4. Again, in the predictor step, each processor initializes the entire vector $\mathbf{v}_l^{(0)}$ according to (2). To do this, the approximation vector $\mathbf{y}_\kappa$ must be replicated on *all* processors. 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. The group partitioning is used only 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.

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. In the predictor step, group G_l initializes vector $\mathbf{fval}_l^{(j)}$ cyclically according to (5). In the corrector step, group G_l computes

```
forall l ∈ {1,...,s} do
  forall q ∈ G_l do {
    initialize v_l^(0) according to (2);
    first processor in group: broadcast v_l^(0) to other groups;
    initialize ⌈n/g_l⌉ components of fval_l^(0) cyclically according to (5);
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 in parallel by all g_l processors in group G_l;
      compute ⌈n/g_l⌉ components of fval_l^(j) cyclically according to (8);}
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; }
```

Fig. 5. Parallel macrostep of the DIIRK version **GrpRed**.

vector $\mathbf{fval}_l^{(j)}$ cyclically according to (8). 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. 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.

4 Numerical Experiments

For the numerical experiments with parallel DIIRK methods on an Intel iPSC/860 we use a 3–stage Radau method [1] of order $p = 5$ as corrector and 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: (a) $\mathbf{f}$ has fixed evaluation costs that are independent of the system size (sparse function). (b) 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 time-dependent PDEs results in a function $\mathbf{f}$ with a constant computational effort [1]. A function $\mathbf{f}$ with system size depending evaluation costs arises when solving time-dependent PDEs with Fourier–Galerkin methods.

Figures 6 and 7 show the measured runtimes in seconds and speedup values for one macrostep of the DIIRK method for $p = 16$ processors. 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. The experiments show that it is not obvious which parallel implementation should be preferred. But several observations concerning the runtime and speedup values can be made.

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

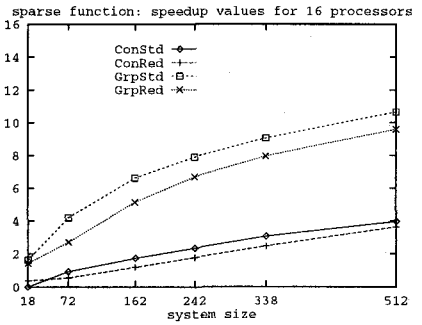


Fig. 6. Runtimes in seconds and speedup values for sparse right hand sides **f**.

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

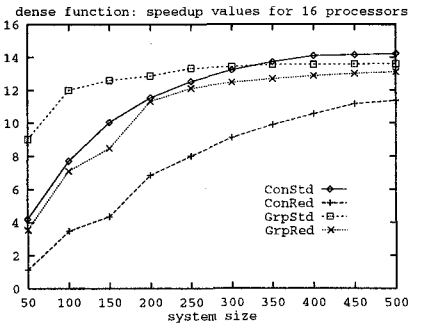


Fig. 7. Runtimes in seconds and speedup values for dense right hand sides **f**.

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 and by 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 numbers 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** and 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, therefore, use less communication time.

Efficiency: The efficiency of the implementations mainly depends on the application but also on the number of processors. The application of dense functions results 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 mainly 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 ODE 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 especially be observed for sparse functions. 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 numbers of processors. This is the case 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 numbers 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. 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 Figure 6. 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

We have presented four parallel implementations of the DIIRK method which result from a combination of different computation schemes with different algorithms. The result of the experiments confirm that the performance of these implementations strongly depend on the application and the number of processors available. For dense functions and large systems, a consecutive algorithm results in smaller execution times than a 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 than for the consecutive implementation because the communication overhead is smaller. 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. E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems.* Springer–Verlag, Berlin, 1993.
2. E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II.* Springer, 1991.
3. A. Iserles and S.P. Nørsett. On the Theory of Parallel Runge–Kutta Methods. *IMA Journal of Numerical Analysis*, 10:463–488, 1990.
4. T. Rauber and G. Rünger. Parallel Iterated Runge–Kutta Methods and Applications. *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.
5. T. Rauber and G. Rünger. Performance Analysis for a Parallel Newton Method. *to appear in: International Journal of High Speed Computing*, 1996.
6. 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.
7. 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.
8. 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.