Parallel Execution of Embedded and Iterated Runge-Kutta Methods

THOMAS RAUBER
Institut für Informatik
Universität Halle-Wittenberg
Kurt-Mothes-Str.1
06120 Halle (Saale), Germany
rauber@informatik.uni-halle.de
Tel. (49)(345)5524711
FAX. (49)(345)5527009

GUDULA RÜNGER
Institut für Informatik
Universität Leipzig
Postfach 920
04009 Leipzig, Germany
ruenger@cs.uni–sb.de
Tel. (49)(341)9732207

Abstract

In this paper, we consider the parallel solution of nonstiff ordinary differential equations with two different classes of Runge-Kutta (RK) methods providing embedded solutions: classical embedded RK methods and iterated RK methods which were constructed especially for parallel execution. For embedded Runge-Kutta methods, mainly the potential system parallelism is exploited. Iterated RK methods provide an additional source of parallelism in the form of independent function evaluations, but they usually require a higher number of function evaluations. We emphasize on the parallel execution time of those methods.

1 Introduction

Many computationally intensive scientific simulations can be solved efficiently by the use of parallel computers if appropriate parallel algorithms are available. In this paper, we consider the parallel solution for systems of non-stiff ordinary differential equations (ODEs) of the form

$$y'(x) = f(x, y(x)) \text{ with } y(x_0) = y_0 \tag{1}$$

with explicit Runge-Kutta (RK) methods providing embedded solutions.

Modern RK methods include an error control and stepsize selection mechanism so that the accuracy of the resulting discrete solution is consistent with a predefined error tolerance [4, 6]. To save evaluations of the right-hand side f, *embedded* RK formulas are used that allow the computation of two approximations of different order with the same function evaluations. Popular embedded RK methods are the Fehlberg methods [5] or the Dormand&Prince methods [8]. Both approaches have been used to construct embedded RK formulas of different orders. The number of function evaluations usually increases faster than the order of approximations. Variants of these methods are used in many software libraries like DVERK or RKSUITE [2].

Parallel solution methods for ODEs (based on one-step methods) have to concentrate on the parallel execution of one time step, as the approximations to successive points in time have to be computed one after another. Usually potential *system parallelism* is exploited, i.e., the computation of the components of the iteration vectors is distributed among the processors. New parallel methods have been proposed to increase the potential parallelism within one time step. An example are iterated Runge-Kutta methods [11] which also provide embedded solutions. Iterated Runge-Kutta methods require more function evaluations than the best sequential methods, but they provide more parallelism in compensation [10]. The additional potential parallelism is *method* parallelism caused by independent subcomputations within each time step. Combining method parallelism with splitting the evaluation of the function f across several processors results in solution methods with good scalability properties, i.e., large *relative* speedup values can be achieved, especially if the evaluation of function f is costly. A coverage of parallel methods for solving ODE system is given [3].

We use a generalized SPMD (single-program, multiple-data) programming model in which system and/or method parallelism can be exploited. System parallelism is realized by distributing the computation of different components of a system (e.g. the computation of components of solution vector) among the processors, which all perform the same computations on different data. Load balance can be achieved by distributing equal portions of data to the processors. Method parallelism is realized by assigning different subcomputations to different processors. Since potential method parallelism is usually small, independent subcomputations are performed by entire subsets of processors (often called *groups* of processors), each group working in SPMD style. Altogether this results in group-SPMD computations of independently working groups of processors. The algorithms are formulated in a message passing style for parallel machines with distributed memory organization, communication is realized by single-transfer operation and collective communication operations in which all processors of a specific group take part. For shared memory implementations, synchronization and locking of variables has to be taken into account for a correct implementation. Our parallel implementations of the RK methods exploit system parallelism in the case of embedded RK methods, and a mixture of system and method parallelism in the case of iterated RK methods.

We compare the parallel execution of embedded RK methods with iterated RK methods. A runtime analysis shows that the resulting parallel methods differ not only in their computational effort but also in their communication behavior. For parallel embedded methods each processor typically has to send less data to more processors than for the corresponding iterated RK methods. The investigation shows that the best choice of a parallel method depends on the relationship between the size and type of the ODE system and the latency and bandwidth of the parallel machine. For large ODE systems and/or costly function evaluations, or for a small number of processors, parallel embedded methods are faster, whereas for sparse and small ODE systems iterated RK methods are sometimes better.

In the next section, we descibes the computation schemes of embedded RK methods and a parallel implementation. In Section 3, we describe iterated RK methods and give the parallel computation schemes. In Section 4, we analyze their parallel performance and present experimental runtime results. Section 5 concludes.

2 Embedded Runge-Kutta Methods

The following short description of embedded and iterated RK methods concentrates on the computation scheme of the methods and the inherent potential parallelism. In each time step, RK methods with embedded solutions compute not only the approximation y_κ for the solution function of Equation (1) but also a second approximation $\hat{y}_\kappa$ which uses the same function values but combines them differently.

$$\begin{aligned} k_l &= f(x_\kappa + c_l h_\kappa, y_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} k_i), \quad l = 1, \ldots, s & (2) \\ y_{\kappa+1} &= y_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l k_l & (3) \\ \hat{y}_{\kappa+1} &= y_\kappa + h_\kappa \cdot \sum_{l=1}^{s} \hat{b}_l k_l \end{aligned}$$

The order r of approximation $y_{\kappa+1}$ and the order $\hat{r}$ of approximation $\hat{y}_{\kappa+1}$ usually differ by 1, i.e., $r = \hat{r} + 1$. The difference between the two approximations $y_{\kappa+1}$ and $\hat{y}_{\kappa+1}$ gives an asymptotical estimate for the local error in the lower order approximation and can be used for stepsize control [4]. The current step is accepted, if a suitable weighted norm of the local error estimate lies within a predefined tolerance level. Although the estimate of the local error is in the lower order approximation, the more accurate approximation is usually used to advance the integration (local extrapolation). Several embedded RK methods have been derived including the methods of Dormand & Prince (e.g., DOPRI5 of order 5(4) or DOPRI8 of order 8(7)) or Verner's methods [6].

Potential parallelism arises in embedded RK methods when systems of ODEs are considered. Each function evaluation for the computation of a stage vector k_l can be distributed among the processors. Depending on the coefficient matrix (a_{li}), there might also be a

```
forall processors $q$ do in parallel {
    for  $l = 1, \dots, s$ do sequentially {
        compute $\lceil n/p \rceil$ el. of $\tilde{\mu}(l) = y_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} k_i$;
        make $\tilde{\mu}(l)$ available to all processors;
        compute $\lceil n/p \rceil$ el. of $k_l = f(x_{\kappa+1} + c_l \cdot h, \tilde{\mu}(l))$;
    }
    compute $\lceil n/p \rceil$ el. of $y_{\kappa+1} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l k_l$;
    compute $\lceil n/p \rceil$ el. of $\hat{y}_{\kappa+1} = y_\kappa + h_\kappa \sum_{l=1}^{s} \hat{b}_l k_l$;
    decide on acceptance of $y_{\kappa+1}$
    compute new stepsize $h_{\kappa+1}$
}
```

Figure 1: Parallel execution of one time step of an embedded RK method.

potential for independent computation of different function evaluations for k_{l_1} and k_{l_2}, which we do not consider here.

For a sequential execution, the number of f evaluations required gives a reasonable measure of the costs of a timestep. For a parallel execution, the costs of communication and synchronization have also to be considered.

Figure 1 shows a parallel implementation of one time step of an embedded RK method which exploits system parallelism in an SPMD style. For an ODE system with n equations, a solution vector with n components has to be computed in each time step. For an s-stage method, the parallel algorithm computes the s stage vectors $k_1, \dots, k_s$ one after another. For each stage vector, every processor computes an equal number of components of the argument vector $\tilde{\mu}(l)$. After the computation of $\tilde{\mu}(l)$, the components of $\tilde{\mu}(l)$ is made available to all processors. Then, each processor computes an equal number of components of the stage vector k_l by evaluating the corresponding components of the function f. Note that the components of $\tilde{\mu}(l)$ have to be made available to all processors before the function evaluations because the evaluation of a component of f by a specific processor q might need other components of $\tilde{\mu}(l)$ besides the ones q has computed locally.

After the distributed computation of all stage vectors, each processor computes an equal number of components of the vectors $y_{\kappa+1}$ and $\hat{y}_{\kappa+1}$ which are both approximations (of different order) to the solution function. The maximum norm $|y_{\kappa+1} - \hat{y}_{\kappa+1}|$ is used to decide on the acceptance of $y_{\kappa+1}$ as the next approximation and to compute a new stepsize $h_{\kappa+1}$.

2.1 Shared-memory implementation

We consider a simple model of a shared-memory machine (SMM), see Figure 2, that describes such a machine as a collection of identical processors each of which has access to any location in a shared global memory. The connection between the processors and

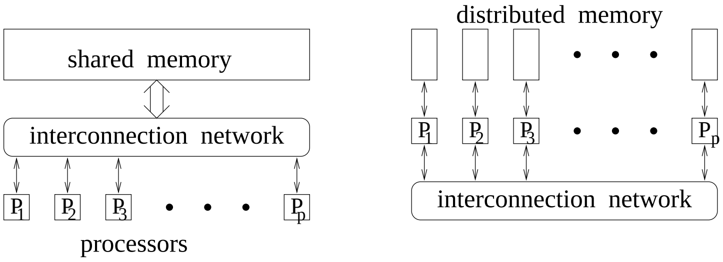

Figure 2: Schematic representation of shared memory and distributed memory architectures.

the global memory is established by an interconnection network that guarantees a uniform memory-access time (UMA machine). Concurrent write-write or read-write accesses to the same memory cell have to be protected by a locking mechanism. The overhead costs for a shared memory implementation are caused by synchronization, memory accesses and avoiding read and write conflicts.

For embedded RK methods, no concurrent accesses to the argument vectors $\tilde{\mu}(l)$ are necessary, if the computations of the different argument vectors are separated by synchronization operations from their use in the successive computation of k_l. Similarly, concurrent accesses to the stage vectors can be avoided, if a synchronization operation is performed before their use in the following time step. Therefore, no memory locks have to be used. For the computation of the maximum norm, a global maximum has to be determined after each processor has computed its local maximum. If no hardware support in the form of multi-prefix operation is provided [1], this has to be done in $\log p$ steps in a tree-like accumulation process. Altogether, $2 \cdot s$ synchronization operations and one accumulation operation are necessary in each time step, resulting in the following runtime costs $T_E(n,p)$ for computation and $C_E(n,p)$ for shared memory overhead:

$$\begin{aligned} T_E(n,p) &= s\,\frac{n}{p}\,T_f \\ C_E(n,p) &= 2\,s\,T_{sync}(p) + T_{acc}(p) \end{aligned}$$

where T_f denotes the time for evaluating one component of function f, $T_{sync}(p)$ is the time for synchronizing p processors, and $T_{acc}(p)$ is the time for an accumulation operation on p processors.

2.2 Distributed-memory implementation

We consider a simple model of a distributed-memory machine (DMM), see Figure 2, in which each processor has exclusive access to its local memory and in which no global memory exists. Accesses to non-local data are performed by message-passing operations involving the processor that wants to access the data and the processor that has the corresponding data in its local memory. More complex data exchange operations are performed by *collective* communication operations. These include a *broadcast* operation in which a local data item of one processor is made available to a set of processors such that the processors have the data item in their local memories after the operation. Another important collective communication operation is a *multi-broadcast* operation in which each of the participating processors contributes a data item which is in the local memory of each of the processors after the operation, i.e., a multi-broadcast operation can be considered as being composed of broadcast operations performed by each of the participating processors with its local data. These operations can be executed on subsets of processors.

For embedded RK methods, a multi-broadcast operation is used to make $\tilde{\mu}(l)$ available to all processors. After the computation of k_l, no communication is required, because in the next iteration step, each processor uses exactly those components of the previous stage vectors that it has computed locally. The same applies to the last iteration step of the stage vector computation, because each processor q computes exactly those components of $y_{\kappa+1}$ and $\hat{y}_{\kappa+1}$ that correspond to the components of the stage vectors that processor q has already computed. Analogously, no communication is necessary before the execution of the next time step because for the computation of the first argument vector $\tilde{\mu}(1)$, each processor needs exactly the local components of y_κ that it has computed in the previous time step. Figure 3 depicts the resulting structure of the DMM implementation. For the stepsize control comprising the decision on the acceptance of $y_{\kappa+1}$ and the computation of a new stepsize $h_{\kappa+1}$, a global accumulation operation determines the global maximum and a broadcast operation makes the global maximum available to each processor. Altogether, s multi-broadcast operations, one accumulation operation and one broadcast operation are necessary in each time step, resulting in the following runtime costs consisting of computation costs $T_E(n,p)$ and communication costs $C_E(n,p)$:

$$\begin{aligned} T_E(n,p) &= s\,\frac{n}{p}\,T_f \\ C_E(n,p) &= s \cdot T_{mb}(p, n/p) + T_{acc}(p,1) + T_{sb}(p,1) \end{aligned}$$

$T_{mb}(p,x)$ denotes the time for the execution of a multi-broadcast operation executed on p processors where each processor contributes x floating point values. Similarly, $T_{acc}(p,x)$ denotes the time for the execution of an accumulation operation and $T_{sb}(p,x)$ denotes the time for the exeution of a broadcast operation. In the following, we will denote the time for a group-multi-broadcast operation with y disjoint groups each containing z processors by $T_{gmb}(p,y,z,x)$.

3 Iterated Runge-Kutta Methods

Iterated RK methods have been derived by van der Houwen and Sommeijer for execution on parallel machines [11]. They are based on s-stage implicit RK methods

$$k_l = f(x_\kappa + c_l \cdot h_\kappa, y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} k_i), l = 1, \dots, s, \qquad (4)$$

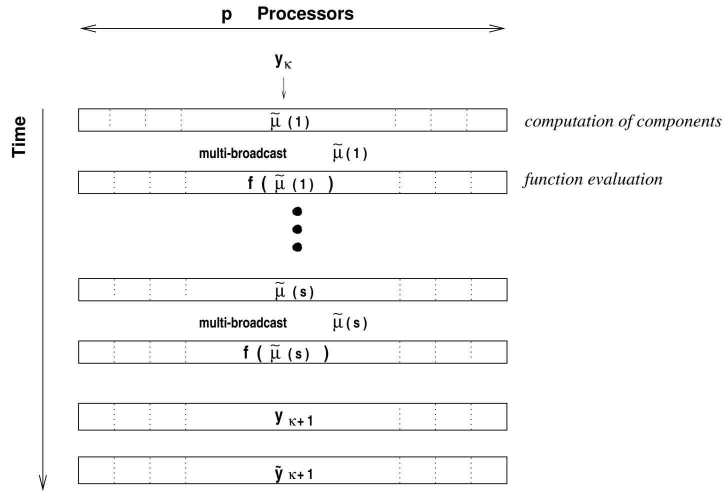

Figure 3: Parallel execution of one time step of an embedded RK method with message passing.

$$y_{\kappa+1} = y_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l k_l$$

which require the solution of an implicit equation system (4). Iterated RK methods iterate this (potentially non-linear) equation system for a fixed number of times m producing approximations $\mu^l_{(j)}$, $j = 1, \ldots, m$, of the stage vectors k_l, $l = 1, \ldots, s$. The last approximations $\mu^l_{(m)}$ are used for the computation of the next approximation of the solution function (denoted by $y^{(m)}_{\kappa+1}$) according to the corresponding computation scheme:

$$\begin{aligned} \mu^l_{(0)} &= f(y_\kappa), \qquad l = 1, \ldots, s \\ \mu^l_{(j)} &= f(y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mu^i_{(j-1)}), \qquad l = 1, \ldots, s, \qquad j = 1, \ldots, m \\ y^{(m)}_{\kappa+1} &= y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu^l_{(m)} \end{aligned} \tag{5}$$

The order of the approximation $y^{(m)}_{\kappa+1}$ is $min\{r, m+1\}$ where r is the order of the underlying implicit RK method. Embedded solutions result when using approximations $\mu^l_{(j)}$ of the stage vectors k_l with $j < m$.

$$y^{(m-1)}_{\kappa+1} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu^l_{(m-1)}$$

is an embedded solution of order m.

One time step of an s-stage iterated RK method requires $s \cdot m + 1$ evaluations of f. A typical example is an iterated RK method based on the 3-stage implicit Radau IA method of order 5. The iterated RK method of order 5 with $m = 4$ iterations requires 13 evaluations of f per time step. The table in Figure 4 contains examples of iterated RK methods specified by the underlying implicit RK method that they are based on.

Order	Embedded RK Method	Iterated RK Method
r=5	Dormand-Prince 5(4) DOPRI5 $s = 7$, f-evaluations: 7	RADAU IA $s = 3$, $m = 4$, f-evaluations: 13
r=6	Verner's method 6(5) DVERK $s = 8$, f-evaluations: 8	LOBATTO IIIA6 $s = 4$, $m = 5$, f-evaluations: 21
r=8	Dormand-Prince 8(7) DOPRI8 $s = 13$, f-evaluations: 13	LOBATTO IIIC8 $s = 5$, $m = 7$, f-evaluations: 36

Figure 4: Comparison of embedded RK methods and iterated RK Methods.

We assume that for the iterated RK methods $m = r - 1$ stage vector iteration steps are performed so that the order of the resulting iterated RK method is r [6, 11].

The advantage of iterated RK methods for parallel execution is that the iteration system (5) of size $s \cdot n$ (n is the system size of the ODE system) consists of s independent systems to determine the approximations $\mu^l_{(j)}$, $j = 1, \dots, m$.

These function evaluations can be performed in parallel by s independent, disjoint subsets of processors (groups) $G_1, \dots, G_s$ of processors. Group $G_l = \{q_{l,1}, \dots, q_{l,g_l}\}$ is responsible for the computation of one subvector $\mu^l_{(j)}$, $l \in \{1, \dots, s\}$. Because each function evaluation requires the same amount of computational work, all groups should have the same number of processors to guarantee a good load balance. This is only possible, if s divides the number of processors p. In the following, we assume that G_l contains g_l processors.

The parallel implementation is given in Figure 5. In each iteration step j, $j = 1, \dots, m$, each processor $q \in G_l$ computes $\lceil n/g_l \rceil$ components of $\tilde{\mu}(l, j)$ and evaluates $\lceil n/g_l \rceil$ components of the function f. Between these two steps, $\tilde{\mu}(l, j)$ has to be made available to all processors of G_l because the evaluation of f may require all components of the argument vector.

The group concept organizes the distribution of the work among the processors. Necessary data exchanges between the processors depend on the memory organization of the parallel machines. The shared-memory implementation requires synchronizations after the computation of the argument vectors (group synchronization) and after the computation of $\mu^l_{(j)}$, $l \in \{1, \dots, s\}$ (global synchronization), thus avoiding conflicting memory accesses.

3.1 DMM implementation for iterated RK methods

For the DMM implementation of the iterated RK method, we consider two cases. In the general case, we must assume that s does not divide p and that p does not divide n. In this case, the different groups may contain different numbers of processors and the different processors may have to compute a different number of components. In the special case

```
initialize $\mu^l_{(0)} = f(y_\kappa), l = 1, \ldots, s$;
for $j = 1, \ldots, m$ do sequentially {
  forall groups $G_l$, $l \in \{1, \ldots, s\}$ do in parallel {
    forall processors $q \in G_l$ do in parallel {
      compute $\lceil n/g_l \rceil$ el. of $\tilde{\mu}(l,j) = y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mu^i_{(j-1)}$;
      make $\tilde{\mu}(l,j)$ available to all processors of $G_l$;
      compute $\lceil n/g_l \rceil$ el. of $\mu^l_{(j)} = f(\tilde{\mu}(l,j))$; }}
  forall processors $q$ do
    exchange the computed components of $\mu^l_{(j)}$;
}
forall processors $q$ do {
  compute $\lceil n/p \rceil$ el. of $y_{\kappa+1} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu^l_{(m)}$;
  compute $\lceil n/p \rceil$ el. of $\hat{y}_{\kappa+1} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu^l_{(m-1)}$;
}
make $y_{\kappa+1}$ available to all processors;
```

Figure 5: Parallel execution of a time step of the iterated RK method using a group distribution.

that s divides p, all groups have the same size. Hence, each processor of each group computes the same number of components. Therefore, a more efficient communication scheme can be used. We first consider the general case.

In the general case, there are two possibilities for the initialization at the beginning of each time step:

- distributed computation without computational redundancy: each processor computes (at most) $\lceil n/p \rceil$ components of $f(y_\kappa)$. Then the complete vector $f(y_\kappa)$ is made available to each processor using a multi-broadcast operation. This can be interpreted as *communication redundancy*, since each processor $q \in G_l$ actually only needs $\lceil n/g_l \rceil$ components to perform its local initializations. But making exactly those components available would result in an irregular communication pattern resulting from the different group sizes.
- parallel computation with *computational redundancy*: each processor $q \in G_l$ computes exactly those $\lceil n/g_l \rceil$ components of $f(y_\kappa)$ that it needs for its local initializations. Thus, each component of $f(y_\kappa)$ is evaluated s times, but no communication is required.

The first version should be preferred, if the evaluation of f is expensive compared to the execution of a multi-broadcast operation. Otherwise, the second version is less expensive.

For the stage vector iteration, each processor $q \in G_l$ computes $\lceil n/g_l \rceil$ components of $\tilde{\mu}(l,j)$ which are then made available to all processors of G_l by a group-multi-broadcast operation. As the different groups are working independently from each other, the group-multi-broadcast operations can be performed by all processors in parallel. After the computation of $\mu^l_{(j)} = f(\tilde{\mu}(l,j))$ by processor group G_l, each processor must obtain $\lceil n/g_l \rceil$ components of all iteration vectors $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ to execute the next iteration. If $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ are

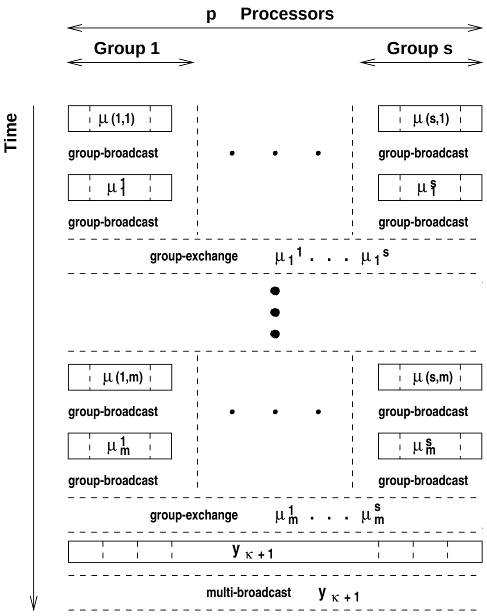

Figure 6: Parallel execution of a time step of the iterated RK method — Group distribution.

allocated in a contiguous memory block on each processor, a multi-broadcast operation can be used with $q \in G_l$ contributing a subvector of size $\lceil n/g_l \rceil$. This is communication redundancy, but avoids complicated special cases. An alternative realization in two steps works if $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ are allocated in separate memory blocks: In the first step, the first processor q_{l1} of each group G_l collects $\mu^l_{(j)}$ by a group-multi-broadcast operation in G_l. In the second step, q_{l1} broadcasts $\mu^l_{(j)}$ to all $q \notin G_l$.

The computation of the new approximation vector is performed by all processors in a non-group way, i.e., each processor computes $\lceil n/p \rceil$ components of $y_{\kappa+1}$ and $\hat{y}_{\kappa+1}$. After the stepsize control which is similar as for the embedded RK methods, $y_{\kappa+1}$ is made available to all processors before the next time step by a multibroadcast operation. This corresponds to the implementation in Figure 6. The runtime costs for computation $T_I(n,p)$ and communication $C_I(n,p)$ are as follows:

$$\begin{aligned} T_I(n,p) &= (s \cdot m + 1)\frac{n}{p}T_f \\ C_I(n,p) &= m \cdot T_{gmb}(p,s,g_l,n/g_l) + m \cdot T_{mb}(p,g_l) + 2 \cdot T_{mb}(p,n/p) + T_{acc}(p,1) + T_{sb}(p,1) \end{aligned}$$

We have tested two other variants of this algorithm that are derived from a modified computation scheme resulting from (5) by a delay the function evaluations, i.e., by computing modified iteration vectors $\sigma^l_{(j)}$ defined by the transformation $f(\sigma^l_{(j)}) = \mu^l_{(j)}$. The modified computation scheme can be computed in a data-parallel as well as in a group parallel way. A detailed description can be found in [10].

3.2 Optimized special case

For the special case that all groups have the same size $g = p/s$ and that p divides n, a more efficient communication can be used for the iterated RK methods. This communication is based on the definition of groups $Q_1 \ldots, Q_g$ with $|Q_k| = s$ and

$$Q_k = \{q_{lk} \in G_l \mid l = 1, \ldots, s\}.$$

For the initialization, each processor computes n/p components of $f(y_\kappa)$ such that the processors of Q_k compute exactly those components that are needed for the initialization of their local components of $\mu^1_{(0)}, \ldots, \mu^s_{(0)}$. Thus, each group Q_k computes n/g components of $f(y_\kappa)$. Since each processor of Q_k needs all these components, they are made available by a group-multi-broadcast operation that is performed on $Q_1 \ldots, Q_g$ in parallel.

Up to the computation of the iteration vectors $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$, the iteration steps are performed in the same way as in the general case. But since each processor computes the same number of components, the exchange of the elements of $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ can be performed more efficiently: Instead of making all components available to each processor, group-multibroadcast operations can be performed on $Q_1 \ldots, Q_g$ in parallel with each processor q_{lk} of Q_k contributing n/g components of $\mu^l_{(j)}$, thus making to each processor exactly those components of $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ available that it needs for the execution of the next iteration. After the last iteration, each processor of Q_k has n/g corresponding components of $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$, i.e., each of the s processors of Q_k can compute a different part of size n/p of $y_{\kappa+1}$ and $\hat{y}_{\kappa+1}$ without preceding communication. In particular, part i can be computed by processor $q_{i\%(s+1),\lceil i/g \rceil}$ of $Q_{\lceil i/g \rceil}$. The stepsize control is the same as for the general case. This results in the following runtime costs:

$$\begin{aligned}
T_I(n,p) &= (s \cdot m + 1)\frac{n}{p}T_f \\
C_I(n,p) &= m \cdot T_{gmb}(p, s, g, n/g) + m \cdot T_{gmb}(p, g, s, n/g) \\
&\quad + T_{gmb}(p, g, s, n/p) + T_{mb}(p, n/p) + T_{acc}(p, 1) + T_{sb}(p, 1)
\end{aligned}$$

4 Runtime Analysis and Experiments

For the RK methods in table in Figure 4, we compare the runtime of embedded and iterated RK methods of the same order r. For the iterated RK methods, the number of iterations is chosen as $m = r - 1$. For a comparison, the following observations are important:

- The embedded RK methods require less computation time than the iterated RK methods; this effect increases with the order of the method.
- For the embedded RK methods, the number of communication operations depends on the number of stages s, whereas for the iterated RK methods it depends on the

number of iterations m; for increasing order, s grows faster than m, see Figure 4 and [6].

- The execution of a group-multi-broadcast usually requires less runtime than the execution of a multi-broadcast because for a group-multi-broadcast, the data elements of one processor need to be sent only to the processors of the same group but not to all processors; this effect increases with the number of processors.
- Similarly, the execution of a group-exchange operation requires less runtime than the execution of a multi-broadcast operation because only s instead of p processors are sending data elements to other processors; this effect is also increasing with the number of processors.
- On the other hand, for both the group-multi-broadcast and the group-exchange operations executed by the iterated RK method, each processor contributes more data elements than for the corresponding multi-broadcast operations of the embedded RK methods.

If the computation time dominates the communication time, the embedded RK methods have a smaller execution time than the corresponding iterated RK methods; this is usually the case if the evaluation of f is costly. Otherwise, the outcome of the comparison depends on the order of the method, the size of the ODE system, the number of executing processors, and the communication behavior of the target machine. As target machines, we consider the Intel Paragon and the IBM SP2. Both machines are DMMs which we program with C and MPI. We performed runtime tests with $p = 2, 4, 8, 16$ and 32 processors. As test problems, we consider two ODE systems.

The first problem is a reaction-diffusion equation (Brusselator equation), see [7]:

$$\begin{aligned}\frac{\partial u}{\partial t} &= 1 + u^2 v - 4.4u + \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)\\ \frac{\partial v}{\partial t} &= 3.4u - u^2 v + \alpha\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)\end{aligned}$$

for $0 \leq x \leq 1$, $0 \leq y \leq 1$, $t \geq 0$, and $\alpha = 2 * 10^{-3}$. The unknown functions u and v describe the concentrations of the two substances. A Neumann boundary condition

$$\frac{\partial u}{\partial n} = 0, \qquad \frac{\partial v}{\partial n} = 0,$$

and the initial conditions

$$u(x, y, 0) = 0.5 + y, \quad v(x, y, 0) = 1 + 5x$$

are used. A standard discretization of the spatial derivatives with a uniform grid with mesh size $1/(N-1)$ leads to the following ODE system of dimension $2N^2$:

$$\frac{du_{ij}}{dt} = 1 + u_{ij}^2 v_{ij} - 4.4u_{ij} +$$

$$\begin{aligned} & \alpha(N-1)^2\left(u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{i,j}\right) \\ \frac{dv_{ij}}{dt} &= 3.4u_{ij} - u_{ij}^2 v_{ij} + \\ & \alpha(N-1)^2\left(v_{i+1,j}+v_{i-1,j}+v_{i,j+1}+v_{i,j-1}-4v_{i,j}\right). \end{aligned}$$

The second problem is a Schrödinger–Poisson system for the numerical simulation of a collisionless electron plasma [9]:

$$\begin{aligned} i\hbar\frac{\partial\Psi}{\partial t} &= -\frac{\hbar}{2m}\frac{\partial^2\Psi}{\partial x^2} + e\Phi\Psi \\ \frac{\partial^2\Phi}{\partial x^2} &= \frac{e\ n_0}{\varepsilon_0}(1-\Psi\Psi^*) \end{aligned}$$

with electron charge e and electron mass m. Ψ is the unknown Schrödinger wave function, Φ is a known potential. A Galerkin approximation with the Galerkin Ansatz

$$\Psi_n = \sum_{|l|\le n}\alpha_l(t)h_l, \quad \Phi_n = \sum_{|l|\le n}\beta_l(t)h_l, \qquad n\in N,$$

results in the following ODE system for the coefficients $\alpha_l(t)$ and $\beta_l(t)$ of Ψ_n and Φ_n

$$\begin{aligned} i\frac{d\alpha_l}{dt} &= -\lambda_l\alpha_l + \sum_{|j|\le n}\alpha_l\cdot(\phi_n h_j, h_l),\ l=0,\pm1,..,\pm n \\ \beta_l &= \frac{1}{\lambda_l}(|\psi_n|^2, h_l), \quad l=\pm1,\dots,\pm n \end{aligned}$$

with $\lambda_l = (\frac{2\pi l}{L})^2$. The functions

$$h_l(x) = \frac{1}{\sqrt{l}}\ \exp\Big(i\ \frac{2\pi l}{L}\ x\Big), \quad l\in Z$$

are an orthonormal basis of $L^2([0,L])$ with the usual scalar product (u,v).

A characteristic of the Brusselator ODE system is that the evaluation time of f does not depend on the size of the ODE system (*sparse* f). The evaluation time is $T_f = 7.5\mu s$ on the Paragon and $T_f = 2.1\mu s$ on the SP2. A characteristic of the Schrödinger ODE system is that the evaluation time for f depends linearly on the size n of the ODE system (*dense* f). The evaluation time is $T_f = 0.35\mu s\cdot n$ on the Paragon and $T_f = 0.31\mu s\cdot n$ on the SP2.

To compare embedded and iterated RK methods, we apply the methods from Figure 4 to the test problems with tolerances between 10^{-2} and 10^{-6}. For the Brusselator ODE, the precision of the solution is computed by comparing the numerical result with a solution obtained with DOPRI8 using tolerance 10^{-13}. For the Schrödinger ODE, the precision is computed by comparing the numerical result with an exact. that is known for a special intial value. Figures 7, 8, and 9 show the result for methods of the same order on different numbers of processors of the Intel Paragon.

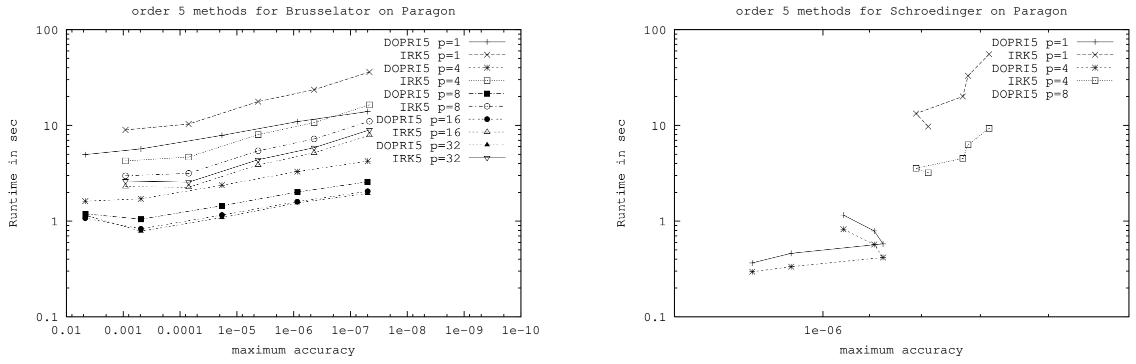

Figure 7: Runtimes and precision for Brusselator (left) and Schrödinger (right) on the Intel Paragon.

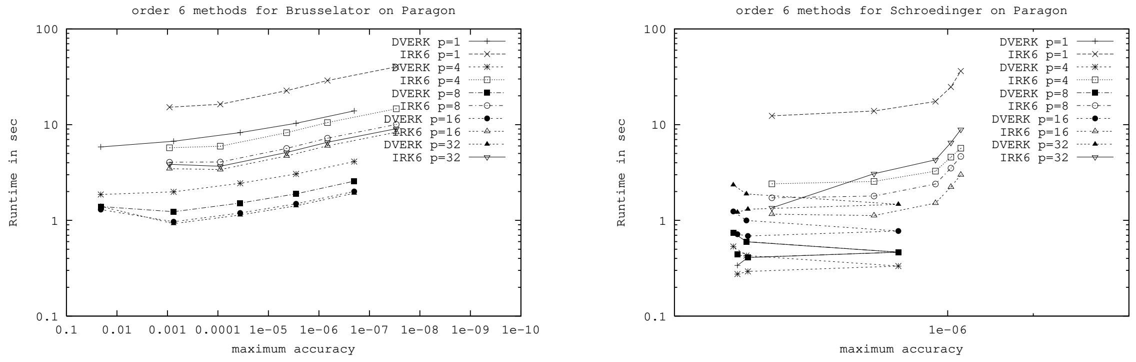

Figure 8: Runtimes and precision for Brusselator (left) and Schrödinger (right) on the Intel Paragon.

For the Schrödinger ODE, the iterated RK methods of order 5 and 6 lead to a higher precision, but require more computation time. For order 8, the embedded RK methods lead to a higher precision, but for a larger number of processors, the execution time is higher than the execution time of the corresponding iterated RK method. Because a group implementation of the iterated RK method with 5 stages is not possible with 4 processors, this variant is not shown in Figure 9. For the Brusselator ODE, the iterated RK methods

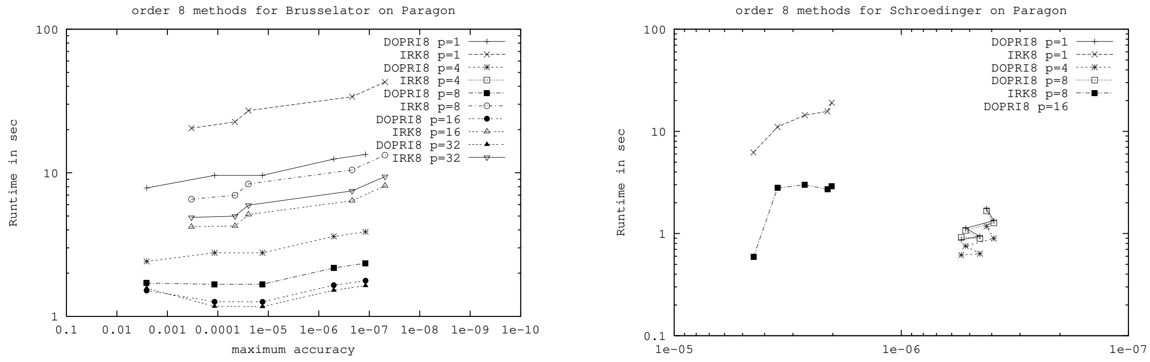

Figure 9: Runtimes and precision for Brusselator (left) and Schrödinger (right) on the Intel Paragon.

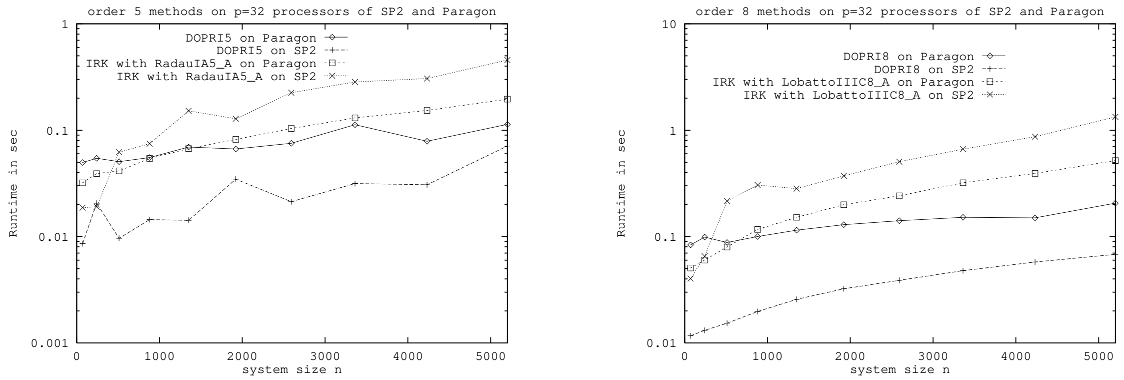

Figure 10: Runtimes of RK methods applied to sparse ODEs on the Paragon and the SP2.

lead to a slightly higher precision, but require considerably more computation time. In all cases, the iterated RK methods show a better scalability than the embedded RK methods due to their larger degree of parallelism.

Figure 10 shows the runtime of the corresponding embedded and iterated RK methods on 32 processors of the Intel Paragon and IBM SP2 for methods order 5 and 8. For dense f, the embedded RK methods are significantly faster than the corresponding iterated RK methods on both target machines for different numbers of processors and different sizes of the ODE system (between $n = 100$ and $n = 10000$). For sparse f, the iterated RK methods are slightly faster than the embedded RK methods for $p = 32$ processors and small sizes n of the ODE system. This effect can be observed on the Paragon for methods of different order, see Figure 10, where the basic iterated RK methods have the smallest runtime for ODE systems with up to $n = 800$ equations. For such small systems, the startup times of the communication operations have a much larger influence on their execution time than the number of data items transferred. For example, the startup time of a multi-broadcast operation with p processors is $24\mu s + 210\mu s \cdot p$ on the Paragon and $68\mu s + 180\mu s \cdot p$ on the SP2. The transmission time per byte is $0.04\mu s \cdot p$ on the Paragon and $0.06\mu s \cdot p$ on the SP2. The group-multi-broadcast operations and group exchange operations have smaller startup times because they are composed of single-transfer operations. Therefore the communication overhead of the iterated RK methods is smaller than the communication overhead of the embedded RK methods for large p and small n. For small p or large n, the smaller computation times of the embedded RK methods dominate the communication times so that the embedded RK methods always have the smallest execution times. Moreover, in one time step the embedded RK methods require the communication of less data than the iterated RK methods. Since the influence of the transmission times is increasing with n, the embedded RK methods derive an additional advantage. The runtime difference between the embedded and iterated RK methods is larger on the SP2 than on the Paragon, because the SP2 has a slower transmission time and because the difference between the startup times of the global communication operations and the group communication operations is smaller than on the Paragon.

5 Conclusion

This article compares parallel implementations of embedded RK methods and iterated RK methods of different order by applying them to ODE systems with different characteristics. The comparison shows that the embedded RK methods yield the smallest runtime in most cases because of their efficient computational organization.

6 Acknowledgements

We would like to thank the SFB 124, Saarbrücken, for financial support and the GMD in St. Augustin for giving access to the SP2.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. On the Cost–Effectiveness of PRAMs. In *Proceeding of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1991.

[2] R.W. Brankin, I. Gladwell, and L.F. Shampine. *RKSUITE release 1.0*, 1991.

[3] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations.* Oxford Science Publications, 1995.

[4] Wayne H. Enright, Desmond J. Higham, Brynjulf Owren, and Philip W. Sharp. A Survey of the Explicit Runge-Kutta Method. Technical Report 94-291, University of Toronto, Department of Computer Science, 1995.

[5] E. Fehlberg. Classical fifth–, sixth–, seventh– and eighth order Runge–Kutta formulas with step size control. *Computing*, 4:93–106, 1969.

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

[7] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II.* Springer, 1991.

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

[9] T. Rauber and G. Rünger. Parallel Solution of a Schrödinger–Poisson system. In *International Conference on High-Performance Computing and Networking*, pages 697–702. Springer LNCS 919, 1995.

[10] T. Rauber and G. Rünger. Parallel Iterated Runge–Kutta Methods and Applications. *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.

[11] 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.