Parallel Iterated Runge–Kutta Methods and Applications

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

November 3, 1994

Abstract

The iterated Runge–Kutta (IRK) method is an iteration scheme for the numerical solution of initial value problems (IVP) of ordinary differential equations (ODEs) that is based on a predictor–corrector method with an Runge-Kutta (RK) method as corrector. Embedded approximation formulae are used to control the stepsize. We present different parallel algorithms of the IRK method on distributed memory multiprocessors for the solution of systems of ODEs. The parallel algorithms are given in an SPMD (single–program multiple–data) programming style where data exchanges are described with appropriate communication primitives. A theoretical performance analysis and a runtime simulation allow to value the presented algorithms. The implementation on the Intel iPSC/860 confirms the predicted runtimes. The speedup values strongly depend on the particular system of ODEs to be solved. The parallel IRK method is applied to a typical discretization problem, the discretized Brusselator equation. Application specific modifications of the general parallel ODE solver are developed which result in a considerable reduction of the parallel execution time.

1 Introduction

Large systems of ordinary differential equations (ODEs) with initial value conditions arise, e.g. when discretizing time dependent partial differential equations. The numerical solution of those systems require a very large amount of computing power which may be covered by parallel machines. Although the numerical solution of ODEs with initial value conditions is an inherently sequential procedure (and, thus, difficult to parallelize), systems of ODEs provide a large potential for parallel processing.

The general form of an initial value problem (IVP) of a system of first order ODEs of dimension n is

$$\frac{dy(x)}{dx} = f(x, y(x)), \qquad y(x_0) = y_0, \qquad x_0 \leq x \leq x_{end}, \tag{1}$$

*supported by DFG, SFB 124, TP D4

where $y : \mathbb{R} \to \mathbb{R}^n$ and $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$. For the numerical solution of system (1), several parallel methods have been proposed in the literature [10] [4] [5] [15] [7] [1] [19] [20]. But most of these methods only have a small potential of parallelism. The most promising methods for a parallel execution are extrapolation methods and iterated Runge-Kutta (RK) methods [17]. Extrapolation methods have been proposed in [18] for a parallel execution. [12] and [13] consider the implementation of extrapolation methods on DMMs. Van der Houwen and Sommeijer suggest IRK methods in [16] and [17] for a parallel execution on a shared memory machines. They concentrate on mathematical characteristics (stability, convergence order) of the methods and don't give a runtime analysis or predict or measure speedup values.

For sequential implementations, implicit RK methods are seldom used as corrector in predictor–corrector methods because they are much more expensive than linear multistep correctors. The advantages of using RK methods for a parallel implementation are smaller error-constants and a high degree of parallelism.

In this article, we consider a predictor–corrector method that uses an implicit Runge-Kutta (RK) method as corrector. By iterating the corrector equation for a fixed number of times (fixed point iteration), an explicit Runge–Kutta method is obtained [6]. Those methods are called iterated Runge-Kutta methods (IRK methods). The used implicit or explicit RK corrector does not influence the stability properties of the resulting IRK method. The IRK method is implicit or explicit because of its iteration behavior whatever corrector is used and we, therefore, concentrate on nonstiff ODEs. We propose parallel versions for the IRK method with stepsize control. We describe the parallel algorithms in a coarse–grain compute–communicate SPMD (single–program multiple–data) scheme suitable for the execution on asynchronously working DMMs. Thus, the breakdown of the workload into subtasks, explicit synchronization points and necessary data exchanges are specified. The data exchanges of the presented algorithms are expressed with appropriate communication primitives which are available on all common topologies [2].

The suggested SPMD programming model allows the prediction of runtimes and speedup values *before* an actual implementation is performed. An algorithm needs only to be implemented, if the expected speedup values are satisfactory compared with other algorithms. The timing model has successfully been applied to design parallel algorithms and predict runtimes for extrapolation methods [12], [13].

The developed parallel versions of the IRK-methods exploit a *parallelization across the method (time)* which means that different parts of one time step of the method are assigned to different processors. The presented algorithms mainly differ in the ways of distributing the workload and the data among the processors. Starting with an implementation where groups of processors are responsible for the computation of subsystems, algorithms with delayed function evaluation and cyclic data distribution are developed.

The theoretical performance analysis and a comparison of the presented algorithms are carried out for the machine parameters of the Intel iPSC/860. The analysis is used to decide for a practical implementation of one of the proposed algorithms on the Intel iPSC/860. Practical tests with the implementation on the iPSC/860 provide the numerical evidence of the theoretical predictions.

The attainable speedup values strongly depend on the evaluation time of the right hand side function f. Large speedup values can be reached, if the function evaluation requires a lot of computation time. Such functions result e.g. from the solution of partial differential equations by variational methods [14]. But if the function evaluation only requires a few operations, the communication time dominates the computation time and the speedup values are small. A typical example for such a function f is the function that results from the discretization of the *Brusselator equation*, a partial differential equation describing a chemical reaction [9].

For the improvement of the performance of the IRK method when applied to Brusselator like functions, we consider several alternatives. These alternatives take advantage of the specific access structure of the Brusselator function by using appropriate communication operations or a data distribution specially chosen for this application. By this, the efficiency of the method can be increased by a factor of 2.5.

The remaining part of the paper is organized as follows: Section 2 describes the iterated Runge-Kutta method with stepsize control. Section 3 briefly presents the parallel computation model that we use for the runtime analysis. Section 4 investigates parallel algorithms and the resulting runtimes for the iterated Runge-Kutta method. Section 5 contains the comparisons of the algorithms which comprises the results of the practical implementation on the iPSC/860 and theoretical investigations. Section 6 describes the Brusselator equation and discusses application specific implementations and numerical results.

2 Iterated Runge–Kutta Methods

The iterated Runge–Kutta method is a predictor–corrector method with an s-stage implicit RK corrector, [16], [17]. The chosen iteration strategy for the corrector phase results in an explicit ODE solver.

2.1 Iterated Runge-Kutta (IRK) method with Fixed Number of Iterations

An s–stage, implicit, one–step RK–method has the form

$$y_{\kappa+1} = y_\kappa + h \sum_{l=1}^{s} b_l v^l$$

where y_κ and $y_{\kappa+1}$ are n-dimensional iteration vectors and the n-dimensional vectors v^l, $l = 1, \ldots, s$, are implicitly defined by the following system of equations of dimension $s \cdot n$:

$$v^l = f(y_\kappa + h \sum_{i=1}^{s} a_{li} v^i), \qquad l = 1, \ldots, s.$$

$b = (b_1, \ldots, b_s)$ is an s–dimensional vector and $A = (a_{li})$ is an $s \times s$ matrix specifying the particular RK method under consideration.

From this s–stage RK–method an explicit (iterative) RK–method is obtained by iterating the equations for v^l for a fixed number of times m:

$$\mu^l_{(j)} = f(y_\kappa + h \sum_{i=1}^{s} a_{li}\, \mu^i_{(j-1)}), \qquad l = 1, \ldots, s, \qquad j = 1, \ldots, m,$$

and using the mth iterates $\mu^1_{(m)}, \ldots, \mu^s_{(m)}$ as approximations for $v^1, \ldots, v^s$. The IRK method proposed in [16] uses the iterated s–stage RK method described above as corrector method and a simple (one–step) predictor method for computing the initial approximation $\mu^1_{(0)}, \ldots, \mu^s_{(0)}$ for $v^1, \ldots, v^s$. Thus, one time step of the IRK method is described by the following iteration scheme (I):

$$\mu^l_{(0)} = f(y_\kappa), \qquad l = 1, \ldots, s, \tag{2}$$

$$\mu^l_{(j)} = f(y_\kappa + h \sum_{i=1}^{s} a_{li} \mu^i_{(j-1)}), \qquad l = 1, \ldots, s, \qquad j = 1, \ldots, m, \tag{3}$$

$$y_{\kappa+1} = y_\kappa + h \sum_{l=1}^{s} b_l\, \mu^l_{(m)}. \tag{4}$$

The vector y_κ represents an approximation of the solution y at the point x_κ and $y_{\kappa+1}$ is an approximation of $y(x_\kappa+h)$ that is obtained from y_κ by applying one step of scheme (I) with stepsize h. The computation of $y_{\kappa+1}$ starting from y_κ according to system (I) is called a *macrostep*. The convergence order of the described method is $p^* = min(p, m+1)$, where p is the order of the used RK–method [16].

2.2 Stepsize Control

For the solution of the ODE system (1) in the intervall $x_0 \leq x \leq x_{end}$, several macrosteps using iteration scheme (I) are necessary in order to approximate the solution y at the points,

$$x_0, x_1, x_2, \ldots, x_{end}, \text{ with } x_{i+1} = x_i + h_i.$$

In order to achieve a good approximation 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, we exploit the following automatic stepsize control [6]. With the same given starting stepsize h two different (embedded) approximations y_{i+1} and $\tilde{y}_{i+1}$ for the solution $y(x_{i+1})$ are computed. The new stepsize h_{new} is computed according to the formula

$$h_{new} = h * min(6, max(\frac{1}{3}, 0.9 * \left(\frac{bound}{error}\right)^{1/(ord+1)})) \tag{5}$$

which uses the error between those two approximations $error = ||y_{i+1} - \tilde{y}_{i+1}||$ and the upper bound $bound = max(|y_i|, |y_{i+1}|)$ of the solution in the intervall $[x_i, x_{i+1}]$. ord is the minimal convergence order of the used approximation method.

The new approximation vector y_{i+1} is accepted if $error \leq bound$. In this case, h_{new} is used to compute y_{i+2}. Otherwise, the computation of y_{i+1} is rejected and is repeated with stepsize h_{new}.

The system (I) provides several embedded approximation solutions by using iterations $\mu^l_{(j)}$ for $j < m$ and equation (4) (see [16], [6], [11])

$$y^{(j)} = y_\kappa + h \sum_{l=1}^{s} b_l \, \mu^l_{(j)}.$$

3 Parallel Programming Model

This section proposes a programming model that is suitable for a DMM where 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 formulated in a coarse–grain compute–communicate scheme. The computations are performed according to the SPMD model, i.e. similar subcomputations on different portions of problem data are executed. Thus, the division of the problem data and their assignment to different processors is an important part of the design of an algorithm. In order to avoid data redistribution when combining different modules, one has to ensure a similar distribution structure for the modules.

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]. (Each processor represents one node of the network.)

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

The transfer time of a message of M bytes between two processors P_1 and P_2 (single node to single node) using a transfer path with $d = dist(P_1, P_2)$ processors can be computed by a formula [3] which is independent of the special interconnection network of the DMM

$$t_{s\_s}(d, M) = \tau(d, M) + M \cdot t_c(d, M) \tag{6}$$

$\tau(d, M)$ is the startup time of the message which mainly depends on the distance d, but may also depend on M, e.g. if the target machine uses different communication protocols for messages of different sizes as in the case of the Intel iPSC/860. $t_c(d, M)$ is the time to transfer 1 byte. This time may also depend on M if different protocols are used.

For a hypercube network, Johnsson and Ho address the exact running times of the other communication primitives [8]. The complexities of the primitives for different topologies are given in [2].

The performance of a developed parallel algorithm is measured in a *timing model* that contains the problem sizes and machine descriptions like the processor number or the startup time and the bytetransfer time as parameters. By substituting the actual values of the parallel machine for these parameters and by using topology dependent runtime formulae for the communication primitives, we predict upper bounds of the exact runtime of an algorithm on this machine.

Notation: For the formulation of the parallel algorithm, we use a C-like pseudocode notation. The communication is described with the presented primitives, e.g. `single-broadcast`, `single-gather`, `multi-broadcast`. The execution of a single node gather at a node q with reduction operation *op* and local data R_{local} of the single processors is denoted by

$$R = \texttt{single-gather}_q(op)(R_{local}).$$

After the execution, R is available on q.

A computation is expressed by informal descriptions and some control statements. Those statements are `forall` and `for`. The iterations of a `forall` statement are executed in parallel whereas the iterations of a `for` statement are executed one after another.

For the prediction of the runtimes, we use the abbreviations $t_{s\_broad}(M)$, $t_{s\_gather}(M)$, $t_{m\_broad}(M)$ which denote the times to execute a single node broadcast, a single node gather or a multinode broadcast operation of M bytes. We suppose that an arithmetic operation takes time t_{op}, independently of the operation. Note that this assumption is correct for most of the modern RISC architectures like the SPARC or SuperSPARC processors or the Intel *i*860.

```
/* equation (2) */
forall l ∈ {1,...,s} do
   forall q ∈ G_l do {
      compute ⌈n/g_l⌉ components of f(y_κ);
      initialize ⌈n/g_l⌉ components of μ^1_(0),...,μ^s_(0);
   }
/* equation (3) */
for  j = 1,...,m do {
   forall l ∈ {1,...,s} do
      forall q ∈ G_l do {
         compute ⌈n/g_l⌉ components of μ̃(l,j) = y_κ + h Σ_{i=1}^s a_li μ^i_(j-1);
         multi-broadcast ⌈n/g_l⌉ components of μ̃(l,j);
         compute ⌈n/g_l⌉ components of μ^l_(j) = f(μ̃(l,j));
         multi-broadcast the computed components of μ^l_(j);
      }
/* equation (4) */
   forall processors q do
      compute ⌈n/p⌉ components of y_{κ+1};
      multi-broadcast the computed components of y_{κ+1};
   }
```

Figure 1: algorithm (A) – Group distribution, System (I)

4 Parallel Iterated Runge-Kutta (PIRK) Algorithm

We propose several parallel algorithm for the implementation of the IRK method. These algorithms combine different ways of distributing the computational work and the data among the processors. (p denotes the number of available processors.)

(A) Group distribution: First, we describe the group distribution scheme for the case that the number of available processors is greater than the number of stages, i.e. $p \geq s$. The pseudocode program of this algorithm is given in Figure 1.

The set of processors is divided into s groups $G_1, \ldots, G_s$. The groups G_l contain about the same number $g_l = \lceil p/s \rceil$ or $g_l = \lfloor p/s \rfloor$ of processors, $l = 1, \ldots, s$. The initialization (equation (2)) is performed by each group such that each processor owns $\lceil n/g_l \rceil$ components of $\mu^1_{(0)}, \ldots, \mu^s_{(0)}$ which are needed for the first iteration step. In each iteration step $j = 1, \ldots, m$ of equation (3), group G_l is responsible for the computation of one subvector $\mu^l_{(j)}$, $l \in \{1, \ldots, s\}$. This consists of the computation of vector $\tilde{\mu}(l,j) = y_\kappa + h \sum_{i=1}^{s} a_{li} \mu^i_{(j-1)}$ and the evaluation of $f(\tilde{\mu}(l,j)) = (f_1(\tilde{\mu}(l,j)), \ldots, f_n(\tilde{\mu}(l,j)))$. In order to achieve an even distribution of the computational work among the processors, each processor $q \in G_l$ computes at most $\lceil n/g_l \rceil$ components of $\tilde{\mu}(l,j)$ and executes at most $\lceil n/g_l \rceil$ function evaluations $f_i(\tilde{\mu}(l,j))$. Between these steps, processor q communicates its local elements of $\tilde{\mu}(l,j)$ to the other members of the same group. After each iteration step, each processor sends its local elements of $\mu^l_{(j)}$ to all other processors and, thus, the vectors $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ are available on all processors for the next step. The computation of $y_{\kappa+1}$ is performed in parallel by all processors and the result is broadcasted such that $y_{\kappa+1}$ is available on all processors for the next macrostep.

(B) Group distribution and delayed function evaluation: In order to save communication time, it seems to be convenient to delay the evaluation of function f to the next iteration step by applying the transformation

$$f(\sigma_{(j)}^l) = \mu_{(j)}^l \quad , \; j = 0, \ldots, m-1.$$

This yields a macrostep of the IRK method given by the following system (II):

$$\sigma_{(0)}^l = y_\kappa, \quad l = 1, \ldots, s \quad , \tag{7}$$

$$\sigma_{(j)}^l = y_\kappa + h \sum_{i=1}^{s} a_{li} f(\sigma_{(j-1)}^i) \quad , \quad l = 1, \ldots, s \quad , \quad j = 1, \ldots, m, \tag{8}$$

$$y_{\kappa+1} = y_\kappa + h \sum_{l=1}^{s} b_l f(\sigma_{(m)}^l) \quad . \tag{9}$$

Again, the set of processors is divided into s groups $G_1, \ldots, G_s$ of processors. The initialization of $\sigma_{(0)}^1, \ldots, \sigma_{(0)}^s$ is performed by all processors in parallel. In each iteration step j, group G_l is responsible for the computation of subvector $\sigma_{(j)}^l$, $l = 1, \ldots, s$, i.e. each processor of group G_l performs at most $\lceil n/g_l \rceil$ function evaluations of $f(\sigma_{(j-1)}^i)$, $i = 1, \ldots, s$, and computes at most $\lceil n/g_l \rceil$ components of $\sigma_{(j)}^l$. Because of the delay of the function evaluation, no communication of local elements between these two steps is required. Only at the end of an iteration step, each processor sends its local elements of $\sigma_{(j)}^l$ to all other processors such that $\sigma_{(j)}^l$, $l = 1, \ldots, s$ are available on each processor. The evaluation of $f(\sigma_{(m)}^l)$ and the computation of $y_{\kappa+1}$ is distributed among all processors. A broadcast operation ensures that $y_{\kappa+1}$ is available on all processors for the next macrostep. The pseudocode program of this algorithm is given in Figure 2.

(C) Cyclic block distribution: The cyclic block distribution exploits the fact that the system (II) consists of s subsystems each creating one of the next subvector iteration $\sigma_{(j)}^l$, $l \in \{1, \ldots, s\}$. The initialization is performed by all processors. The computation of each subsystem is evenly distributed among **all** the processors in a similar blockwise way. Considering the entire system (8), this results in a cyclic blockwise distribution with s cycles and block sizes $\lceil n/p \rceil$. Thus, each processor is responsible for the computation of those components of every subvector $\sigma_{(j)}^1, \ldots, \sigma_{(j)}^s$ with the same indices. This consists in at most $\lceil n/p \rceil$ function evaluations of $(f_1(\sigma_{(j-1)}^i), \ldots, f_n(\sigma_{(j-1)}^i))$ and the computation of a block of $\lceil n/p \rceil$ components of the new iteration vector $\sigma_{(j)}^l$ of each subsystem l. The broadcast operation (*) performs the data exchange such that $\sigma_{(j)}^l$, $l = 1, \ldots, s$, are available for the function evaluation in the next iteration step. The computation of $y_{\kappa+1}$ exploits the same blockwise distribution of the computational work as the subsystems. The pseudocode program is given in Figure 3. The blockwise distribution avoids multiple computations of the same function evaluation.

Stepsize control: The stepsize control mechanism presented in Section 2.2 is combined with the macrostep of the IRK-method. We consider the embedded solutions $y_{\kappa+1} = y^{(m)}$ and $y^{(m-1)}$ and choose the maximum norm. For the error, we get the formula

$$\begin{aligned} error &= ||y_{\kappa+1} - y^{(m-1)}|| \\ &= |h| * || \sum_{i=1}^{s} b_l * (f(\sigma_{(m)}^l) - f(\sigma_{(m-1)}^l)|| \\ &= |h| * \max_{i=1,\ldots,n} | \sum_{i=1}^{s} b_l * (f(\sigma_{(m)}^l) - f(\sigma_{(m-1)}^l)| \end{aligned} \tag{10}$$

```
/* equation (7) */
forall q do
   for  l = 1,...,s
      initialize all components of σ^l_(0);
/* equation (8) */
for  j = 1,...,m do {
   forall l ∈ {1,...,s} do
      forall q ∈ G_l do {
         for  i = 1,...,s do
            compute ⌈n/g_l⌉ components of f(σ^i_(j-1));
         compute ⌈n/g_l⌉ components of σ^l_(j);
         multi-broadcast the computed components of σ^l_(j);
      }
}
/* equation (9) */
forall processors q do {
   for   i = 1,...,s do
       compute ⌈n/p⌉ components of f(σ^l_(m));
   compute ⌈n/p⌉ components of y_{κ+1};
   multi-broadcast ⌈n/p⌉ components of y_{κ+1};
}
```

Figure 2: algorithm (B) – Group distribution, System (II)

```
/* equation (7) */
forall q do
   for  l = 1,...,s
      initialize all components of σ^l_(0);
/* equation (8) */
for  j = 1,...,m do
   forall q do {
      for  i = 1,...,s do
         compute ⌈n/p⌉ components of f(σ^i_(j-1));
      for  l = 1,...,s do
         compute ⌈n/p⌉ components of σ^l_(j);
         multi-broadcast the ⌈n/p⌉ computed components of σ^l_(j);   (*)
   }
/* equation (9) */
forall q do {
   for  i = 1,...,s do
      compute ⌈n/p⌉ components of f(σ^i_(m));
   compute ⌈n/p⌉ components of y_{κ+1};
   multi-broadcast ⌈n/p⌉ components of y_{κ+1};
}
```

Figure 3: algorithm (C) : Cyclic data distribution, System (II)

The parallel computation of the stepsize control is given in Figure (4). The value *bound* is computed by determining the local maximum and collecting the local results with a single node gather operation with maximum reduction. *error* is determined according to (10) by computing $f(\sigma^l_{(m)}) - f(\sigma^l_{(m-1)})$ in a distributed way and again collecting the results with a single node gather operation.

The following lemma determines approximations t_A, t_B, t_C and c_A, c_B, c_C to the computation times and the communication times of the presented algorithms (A), (B) and (C).

Lemma 1 *The parallel algorithms of the IRK–method according to Figures 1, 2 and 3 require computation times*

$$t_A = \left(m\left\lceil\frac{n}{g_{min}}\right\rceil + \left\lceil\frac{n}{p}\right\rceil\right)(2s+1)\,t_{op} + \left(m\left\lceil\frac{n}{g_{min}}\right\rceil + \left\lceil\frac{n}{g_{min}}\right\rceil\right)T_f + \left\lceil\frac{n}{g_{min}}\right\rceil s\,t_{op} \quad (11)$$

$$t_B = \left(m\left\lceil\frac{n}{g_{min}}\right\rceil + \left\lceil\frac{n}{p}\right\rceil\right)(2s+1)\,t_{op} + \left(ms\left\lceil\frac{n}{g_{min}}\right\rceil + s\left\lceil\frac{n}{p}\right\rceil\right)T_f + n\,s\,t_{op} \quad (12)$$

$$t_C = \left(m\left\lceil\frac{n}{p}\right\rceil s + \left\lceil\frac{n}{p}\right\rceil\right)(2s+1)\,t_{op} + \left(ms\left\lceil\frac{n}{p}\right\rceil + s\left\lceil\frac{n}{p}\right\rceil\right)T_f + n\,s\,t_{op} \quad (13)$$

and communication times

$$c_A = 2m\,t_{m\_broad}\left(\left\lceil\frac{n}{g_{min}}\right\rceil\right) + t_{m\_broad}\left(\left\lceil\frac{n}{p}\right\rceil\right) \quad (14)$$

$$c_B = m\,t_{m\_broad}\left(\left\lceil\frac{n}{g_{min}}\right\rceil\right) + t_{m\_broad}\left(\left\lceil\frac{n}{p}\right\rceil\right) \quad (15)$$

$$c_C = s\,m\,t_{m\_broad}\left(\left\lceil\frac{n}{p}\right\rceil\right) + t_{m\_broad}\left(\left\lceil\frac{n}{p}\right\rceil\right) \quad (16)$$

```
while  x < x_end do {
     parallel IRK algorithm (A), (B) or (C);
     /* computation of bound */
     forall q do
        compute local-max_q= max of local elements of |y_{κ+1}| ;
     for processor 0 do {
        norm = single-gather_0(max) (local-max_q);
        compute bound = max(||y_{κ+1}||, ||y_κ||);
        broadcast bound;
     }
     /* computation of error */
     forall q do {
        compute ⌈n/p⌉ components of S_q = Σ_{i=1}^{s} b_l * (f(σ^l_(m)) − f(σ^l_(m−1)));
        compute the maximum M_q = max_q(S_q) ;
     }
     for processor 0 do {
        M = single-gather_0(max) (M_q);
        compute error = h * M;
        broadcast error;
     }
     /* equation (5) */
     forall q do {
        compute h_new;
        if (error ≤ bound) x = x + h
        else reject the computed approximation vector
        h = h_new;
     }
}
```

Figure 4: IRK-method with stepsize control

where $g_{min} = \min_{l=1,\ldots,s}(g_l)$ *and* $T_f = \max_{i=1,\ldots,n} t_{eval}(f_i)$.

Proof:

Algorithm (A): The initialization of $\mu^l_{(0)}$ requires $\lceil n/g_l \rceil s$ function evaluations and $\lceil n/g_l \rceil$ assignments. In each of the m iterations, the computation of $\tilde{\mu}(l,j)$ and $\mu^l_{(j)}$ takes time $\lceil n/g_l \rceil (2s+1) t_{op}$ and $\lceil n/g_l \rceil T_f$, respectively. The computation of the next iteration $y_{\kappa+1}$ vector takes time $\lceil n/p \rceil (2s+1)\, t_{op}$. The multi-broadcast operations result in the given communication time.

Algorithm (B): The initialization requires ns assignments. For each iteration, the computation of $\sigma^l_{(j)}$ require $\lceil n/g_l \rceil s$ function evaluations and $\lceil n/g_l \rceil (2s+1)$ arithmetic operations. For the computation of $y_{\kappa+1}$, $\lceil n/p \rceil s$ function evaluations and $\lceil n/p \rceil (2s+1)$ arithmetic operations are necessary.

Algorithm (C): The blockwise initialization require ns assignments. Each iteration step performs $\lceil n/p \rceil s$ function evaluations and $\lceil n/p \rceil s(2s+1)$ operations. The computation of $y_{\kappa+1}$ requires $\lceil n/p \rceil s$ function evaluations and $\lceil n/p \rceil (2s+1)$ arithmetic operations. □

Lemma 2 *The stepsize control presented in Fig.4 requires computation time*

$$t_{STEP} = ((3s+1)\left\lceil \frac{n}{p} \right\rceil + 3)\, t_{op} + 2s \left\lceil \frac{n}{p} \right\rceil T_f \tag{17}$$

and communication time

$$c_{STEP} = 2(t_{s\_gather}(1) + t_{s\_broad}(1)) \tag{18}$$

Proof: Follows directly from the algorithm presented in Figure 4. □

Lemma 3 *A sequential implementation of the IRK method according to system (I) is faster than a sequential implementation according to system (II). The sequential computation times are*

$$\begin{aligned} t_{(I),seq} &= (ms+1)\, n\, (2s+1)\, t_{op} + (sm+1)\, n\, T_f \\ t_{(II),seq} &= (ms+1)\, n\, (2s+1)\, t_{op} + s(m+1)\, n\, T_f \end{aligned}$$

5 Numerical Experiments and Comparisons of the Algorithms

5.1 Runtime Behavior of the iPSC/860

For the prediction of the runtimes on the hypercube iPSC/860, the machine specific times for t_{op}, t_{s_broad} and t_{s_gather} have to be determined and substituted into the formulae of Lemma 1. For t_{op}, we use mean values of the measured runtimes for different operations. This comprises arithmetic operations but also array access times. For t_{s_gather} and t_{s_broad}, we use the theoretically developed runtime formula for the hypercube architecture [8] that depends on the message size M, the startup time τ and the bytetransfer time t_c:

$$t_{s\_broad}(p,M) = t_{s\_gather}(p,M) = \frac{p-1}{\log p} M t_c + \left(\frac{p}{\log p} + \log p \right) \tau$$

The runtime of t_{m_broad} not only depends on the interconnection network of the machine but also on the runtime system. On the iPSC/860, the runtime function *gcolx* is the fastest way to realize

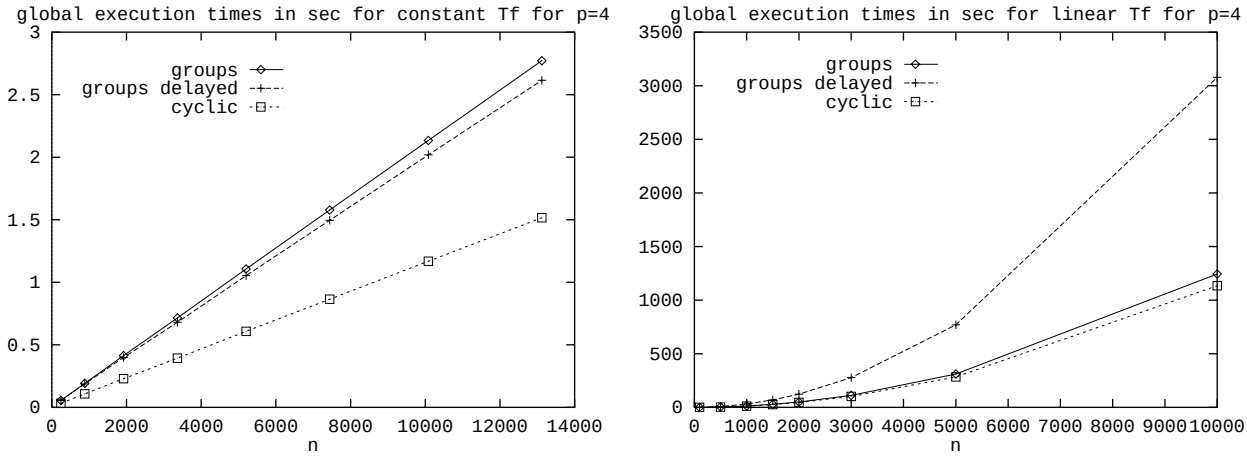

Figure 5: Predicted global execution times with stepsize control for $p = 4$.

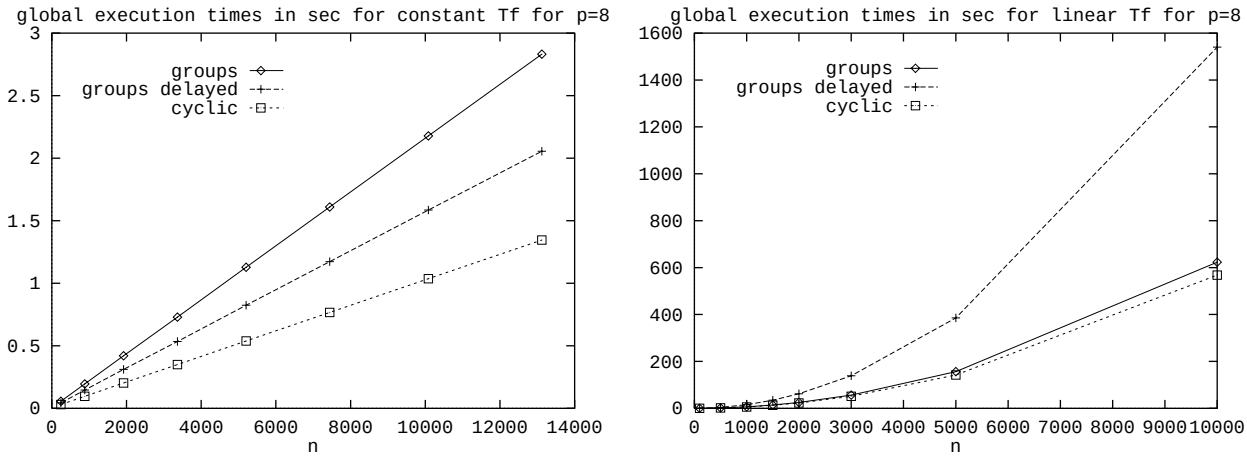

Figure 6: Predicted global execution times with stepsize control for $p = 8$.

a multinode broadcast operation. But the concrete implementation on the topology is hidden for the user. Furthermore, the runtime of *gcolx* does not obey one of the theoretically developed runtime functions presented in [8]. Practical tests show that for fixed number of processors p the runtime for *gcolx* depends linearly on the size of the transmitted messages M, i.e.

$$T_{gcolx}(M) = a_{t_c,\tau}(p)\, M + b_\tau(p)$$

The coefficient $b_\tau(p)$ only depends on p and τ. A possible message size dependent part of the startup time τ is contained in the coefficient $a_{t_c,\tau}(p)$ which additionally depends on t_c and p. The values for t_c and τ are fixed for a special machine like the iPSC/860. The coefficients a and b are monotonically increasing functions of the number of processors p. Tests show $a_{t_c,\tau}(2) = 0.5$, $a_{t_c,\tau}(4) = 2.3$, $a_{t_c,\tau}(8) = 5.6$, and $a_{t_c,\tau}(16) = 12$. For $b_\tau(p)$, we get $b_\tau(2) = 320$, $b_\tau(4) = 800$, $b_\tau(8) = 1200$, and $b_\tau(16) = 1800$.

5.2 Runtime Comparison of the Algorithms

The runtime prediction formulae of Lemma 1 used with the iPSC/860 specific parameters and runtime functions result in simulations of the expected runtimes. The Figures 5, 6, 7 present the predicted runtimes of the parallel IRK algorithms (A), (B) and (C) for different numbers of

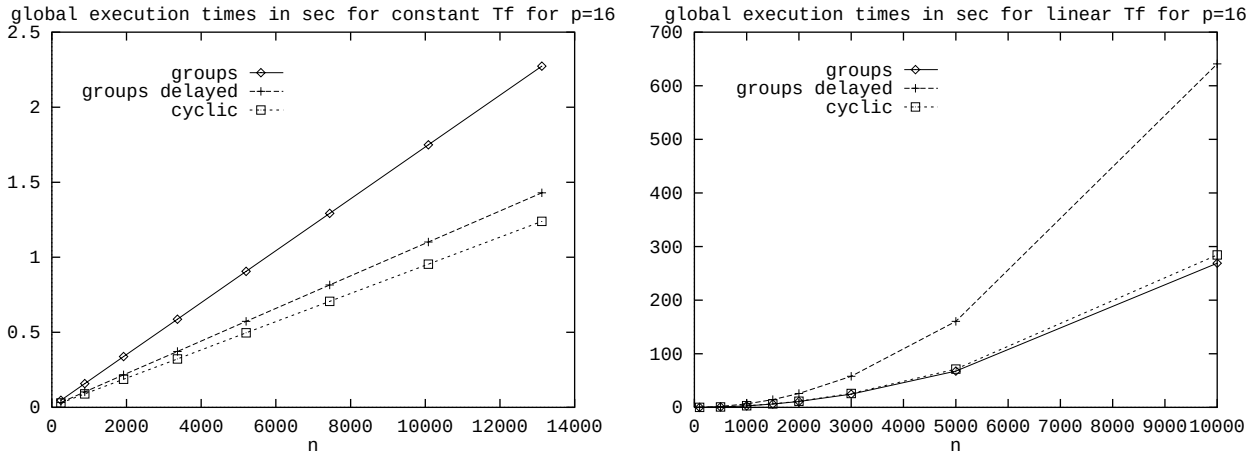

Figure 7: Predicted global execution times with stepsize control for $p = 16$.

processors $p = 4, 8, 16$. We use a 3–stage Radau method [6] of order $p = 5$ as corrector. Because of $p^* = min(p, m + 1)$, we execute 4 corrector iterations.

Each of the Figures 5, 6, 7 contains the runtimes for solving a system of ODEs with two different classes of right hand side functions f:

(con) f has constant evaluation time T_f and
(lin) f has an evaluation time T_f that depends linearly on the system size n.

Both cases may occur in applications when solving time dependent partial differential equations. For example, the spatial discretization of partial differential equations leads to functions f with constant T_f and a variational method may lead to an ODE with functions f with system size dependent T_f.

Let T_A, T_B and T_C denote the global execution times of the algorithms (A), (B) and (C), respectively, i.e. for example $T_A = t_A + c_A$. (The dependence on p, n and T_f is omitted in this notation.) ¿From the simulations, several observations concerning T_A, T_B, T_C can be made:

- The difference of the runtimes of the algorithms (A), (B) and (C) depend on both, the processor number p and the assumption (con) or (lin) for the right hand side function f.
- For f with constant evaluation time we have:
  - $T_A > T_B > T_C$, as it was expected when developing the algorithms.
  - For increasing processor number $p = 4, 8, 16$ the differences between the runtimes change from $T_A > T_B >> T_C$ to $T_A >> T_B > T_C$,
- For a right hand side function f with evaluation time depending on n, we have:
  - $T_B > T_A$ and $T_B > T_C$.
  - For processor numbers $p = 4, 8$ we have $T_B > T_A > T_C$, but for processor number $p = 16$, we have $T_B > T_C > T_A$, i.e. the first algorithm is the fastest.

The algorithms (B) and (C) have been developed because of an expected speeding up of a parallel implementation by reducing the data exchange (from (A) to (B)) or the number of function evaluations (from (B) to (C)). Although this improvement seems to be obvious at first glance, the observation of the theoretical simulation (for some cases of p and T_f) do not confirm the intuition and do even contradict them in some cases. The simulations suggest that the choice

of the algorithm with the fastest computation time strongly depends on the special application and the number of the used processors. In order to explain the observed phenomena and to get a general criteria deciding which of the algorithms is the best in a particular case, we study the runtime formulae of Lemma 1 in more detail. To this end, the runtime formulae are conceived as functions with parameters p and T_f and the asymptotic behavior for large n is investigated.

5.3 Asymptotic Behavior of the Runtimes

For a consideration of the asymptotic behavior, it is sufficient to assume that p and $\lfloor \frac{p}{s} \rfloor$ divide n. (In comparison with n, the numbers p and $\lfloor \frac{p}{s} \rfloor$ are very small.) Let $d = \frac{1}{g_{min}} - \frac{s}{p}$ denote the difference between $\frac{1}{g_{min}}$ and $\frac{s}{p}$ with $g_{min} = \lfloor \frac{p}{s} \rfloor$. Thus, $d > 0$ and $d = 0$ if and only if p is dividable by s.

Lemma 4 *(Runtime comparison for algorithms (B) and (C))*

a) For $s = 1$, we have $T_C = T_B$.
b) If $s > 1$ and $\frac{p}{s} b_\tau(p) \leq T_f$, then for all $n > 1$ we get: $T_B > T_C$.
c) If $s > 1$ and $\frac{p}{s} b_\tau(p) > T_f$, then for all $n > \frac{p}{s} \frac{b_\tau(p)}{T_f}$: $T_B > T_C$.
d) For fixed n and fixed T_f, the difference $T_B - T_C$ is getting smaller if the number of processors p is increasing.

Proof: a) follows directly from Lemma 1. For the other cases we use:

$$\begin{aligned} T_B - T_C &= A_1\, n + B_1 \text{ with} \qquad (19)\\ A_1 &= m\, s \left(\frac{s-1}{p} + d\right) T_f + d\, m \left((2s+1)t_{op} + a_{t_{c,\tau}}\right), \\ B_1 &= (1-s) m\, b_\tau(p), . \end{aligned}$$

from Lemma 1. For $\frac{1}{n} \frac{p}{s} b_\tau(p) \leq T_f$, we get from formula (19): $T_B - T_C > 0$. Thus, b) and c) follow. For d), we assume that s divides p, i.e. $d = 0$. Then, we have $T_B - T_C = (ms\frac{s-1}{p} T_f)n + (1-s) m\, b_\tau(p)$ which is an increasing function of p. □

Remark: Lemma 4 confirms that $T_B > T_C$ holds for almost all system sizes n. (The lower bound $\frac{p}{s} \frac{b_\tau(p)}{T_f}$ for n is small.) The behavior for the case that T_f linearly depends on n is covered by Lemma 4b). Lemma 4d) describes the effect when considering the three simulated plots one below the other in Figures 5, 6, 7 for fixed n.

Lemma 5 *(Runtime comparison for algorithms (A) and (B))*

a) For $s = 1$, we have $T_A = T_B + m t_{m\_broad}(\frac{n}{p}) + \frac{1-p}{p} t_{op} n$.
b) If $s > 1$ and $T_f \geq \frac{mp}{(s-1)(mp+g_{min})} \left(a_{t_{c,\tau}}(p) + g_{min} b_\tau(p)\right)$, then for all $n \in \mathbb{N}$: $T_A < T_B$.
c) If $s > 1$ and $T_f \leq \frac{mp}{(s-1)(mp+g_{min})} \left(s\frac{1-g_{min}}{m} t_{op} + a_{t_{c,\tau}}(p)\right)$, then for all $n \in \mathbb{N}$: $T_B < T_A$.
d) For fixed n and fixed T_f chosen according to c), the difference $T_A - T_B$ is increasing for an increasing number of processors p.

Proof: a) The case $s = 1$ follows directly from Lemma 1. For $s > 1$, we get from Lemma 1

$$\begin{aligned} T_B - T_A &= A_2\, n + B_2 \text{ with} \qquad (20)\\ A_2 &= \left((s-1)\left(\frac{m}{g_{min}} + \frac{1}{p}\right) T_f + s\left(1 - \frac{1}{g_{min}}\right) t_{op} - \frac{m\, a_{t_{c,\tau}}(p)}{g_{min}}\right), \\ B_2 &= -m\, b_\tau(p), \end{aligned}$$

b) From formula (20), we get $T_f \geq \frac{m\,p}{(s-1)\,(m\,p+g_{min})}\Big(a_{t_c,\tau}(p) + \frac{1}{n}\,g_{min}\,b_\tau(p)\Big)$ and thus $T_B > T_A$.
c) $T_f \leq \frac{m\,p}{(s-1)\,(m\,p+g_{min})}\Big(s\frac{1-g_{min}}{m}t_{op} + a_{t_c,\tau}(p) + \frac{1}{n}\,g_{min}\,b_\tau(p)\Big)$, and thus $T_B < T_A$.
d) $T_A - T_B \geq m b_\tau(p)$ which is an increasing function in p. □

Remark: Lemma 5 reflects the fact that $T_B > T_A$ if the additional computation time $(s-1)T_f$ is more expensive than the saved m broadcast operations of time. For small, constant T_f and fixed n, Lemma 5d) explains the varying distances between of T_A and T_B in Figures 5, 6, 7.

The investigation of the last runtime comparison, the difference between T_A and T_C, depends on the value $d = \frac{1}{g_{min}} - \frac{s}{p}$. Thus, for the decision whether algorithm (A) or algorithm (C) is the fastest algorithm and should be used, we have to consider T_f, p, n and d and their interacting influence on $T_A - T_C$.

Lemma 6 *(Runtime comparison for algorithms (A) and (C))*

a) If $s = 1$, then $T_A = T_C + m t_{m\_broad}(\frac{n}{p}) + \frac{1-p}{p}t_{op}n$.
b) If $s = 2$ and $m\,d - 1/p \geq 0$ (i.e. $d \neq 0$), then for all $n \in IN$: $T_A > T_C$.
c) If $s = 2$ and $m\,d - 1/p < 0$, then there exists a constant $G_{p,d} > 0$ such that for $T_f < G_{p,d}$ and all $n \in IN$ we get: $T_A > T_C$.
d) If $s > 2$ then for all n with $\frac{s\,a_{t_c,\tau}(p)}{s-1} - \frac{1}{n}\frac{(s-2)\,p\,b_\tau(p)}{(s-1)} > 0$, there exists a constant $G_{p,d} > 0$ (depending on d, p) such that for $T_f < G_{p,d}$: $T_A > T_C$.
The constant $G_{p,d}$ is an increasing function of d and a decreasing function in p.

Proof: a) follows directly from Lemma 1. For all other cases, we exploit:

$$\begin{aligned} T_A - T_C &= A_3\,n + B_3 \text{ with} \qquad (21)\\ A_3 &= \Big(m\,d + \frac{1-s}{p}\Big)\,T_f + d\,m\,\Big((2s+1)t_{op} + a_{t_c,\tau}\Big) + \frac{m\,a_{t_c,\tau}(p)}{g_{min}} + s(1 - \frac{1}{g_{min}})t_{op},\\ B_3 &= (2-s)m\,b_\tau(p), \end{aligned}$$

b) If $s = 2$, then $B_3 = 0$ and $A_3 > 0$ for $md - 1/p \geq 0$.
c) For $md - 1/p < 0$, we get $T_A > T_C$ if and only if T_f is small enough.
d) For

$$T_f < \frac{s\,m\,a_{t_c,\tau}(p)}{s-1} + d\,\frac{p}{s-1}\Big(m(2s+1) + t_{op} + m\,a_{t_c,\tau}\Big) - \frac{1}{n}\frac{(s-2)\,p\,b_\tau(p)m}{(s-1)} - \frac{s^2}{s-1}(\frac{p}{s} - 1)t_{op},$$

we get according to formula (21) that $T_A > T_C$. □

Remark: Lemma 6 shows that algorithm (C) is faster than algorithm (A) if the time T_f is bounded by a constant depending on the number of processors p and the introduced parameter d. Thus, for small T_f of case (con) the result is obvious (see Figures 5, 6, 7). For the case (lin) and large T_f the situation may change. Because the constant $G_{p,d}$ is decreasing in p (with fixed n and T_f), algorithm (C) may become slower than algorithm (A) which can be observed in Figure 7 for case (lin).

5.4 Implementation

¿From the runtime analysis we get that the third algorithm is the best algorithm for most of our cases. Following the theoretically derived results about the runtimes, we have implemented algorithm (C) on the Intel iPSC/860 using system (II) with a cyclic data distribution. Figures

constant T_f			T_f linear in n		
n	meas.	pred.	n	meas.	pred.
242	0.044	0.032	100	0.156	0.124
882	0.143	0.108	500	2.885	2.876
1992	0.213	0.229	1000	11.418	11.420
3362	0.383	0.393	1500	25.575	25.633
5202	0.625	0.608	2000	45.346	45.517
7442	0.891	0.865	3000	101.598	102.294
10082	1.176	1.169	5000	282.220	283.889
13122	1.524	1.515	10000	1128.543	1134.775

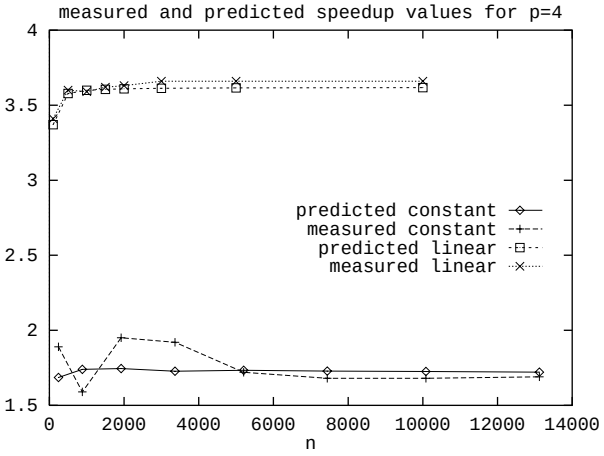

Figure 8: Algorithm (C): measured and predicted running times in seconds and speedup values for $p = 4$.

constant T_f			T_f linear in n		
n	meas.	pred.	n	meas.	pred.
242	0.047	0.030	100	0.083	0.071
882	0.113	0.096	500	1.472	1.474
1992	0.215	0.203	1000	5.731	5.756
3362	0.352	0.349	1500	12.810	12.920
5202	0.551	0.539	2000	22.643	22.849
7442	0.760	0.767	3000	50.737	51.283
10082	1.006	1.036	5000	140.834	142.169
13122	1.336	1.345	10000	561.850	567.835

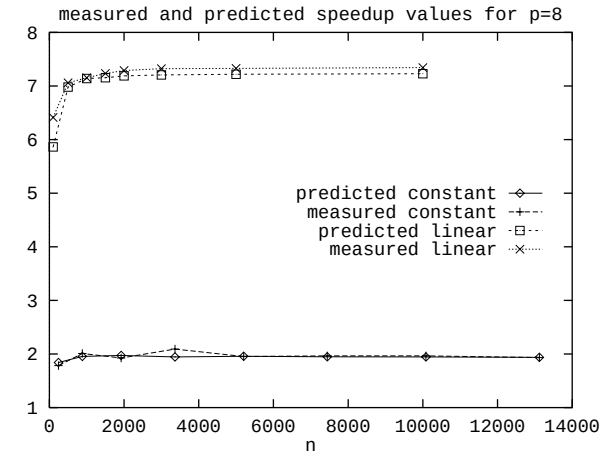

Figure 9: Algorithm (C): measured and predicted running times in seconds and speedup values for $p = 8$.

8,9,10 contain tables with the measured and predicted global execution times and diagrams with the measured and predicted speedup values for $p = 4$, $p = 8$, and $p = 16$ processors. The execution times are given for the cases (con) and (lin), where T_f in the first case comes from the Brusselator example (see Section 6.1) with $n = 2N^2, N \in I\!N$. The given speedup values are obtained by comparing the parallel global execution times for algorithm (C) with the global execution time of the sequential program for (I) (see Lemma 3). All speedup values contains the costs for the stepsize control. Because this is executed by each processor, the speedup values are reduced.

A comparison of the predicted and the measured execution times and speedup values shows that the predicted values are quite accurate. Only for small n, the predicted execution times are smaller than the measured times. This may be caused by a fixed overhead for the parallel program which is not considered in the runtime analysis.

For all considered processor numbers, the speedup values of the (lin) case are much higher than those of the (con) case. The reason for the good speedup values for the (lin) case are the function evaluation times which extremely dominate the communication times. On the other hand, the attained speedup values in the (con) case are very poor.

The given speedup values suggest that the described IRK method is only suited for an implementation on a DMM, if the evaluation time for the right hand side function is large compared with the time to execute a multi–broadcast operation ((lin) case). One possibility to improve

constant T_f			T_f linear in n		
n	meas.	pred.	n	meas.	pred.
242	0.052	0.029	100	0.061	0.046
882	0.113	0.090	500	0.786	0.774
1992	0.211	0.188	1000	2.950	2.947
3362	0.341	0.323	1500	6.510	6.527
5202	0.505	0.496	2000	11.451	11.513
7442	0.714	0.707	3000	25.623	25.842
10082	0.953	0.954	5000	70.733	71.416
13122	1.239	1.239	10000	281.975	284.349

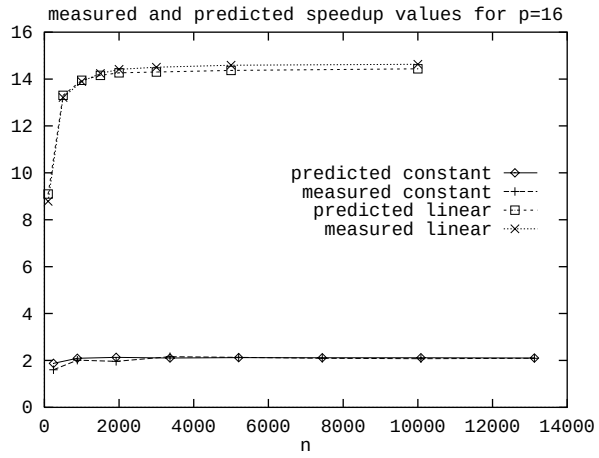

Figure 10: Algorithm (C): measured and predicted running times in seconds and speedup values for $p = 16$.

the communication is to copy the data that has to be communicated into a buffer before the broadcast operation and to execute the broadcast in one step. But experiments on the iPSC/860 show that this only results in a smaller communication time for small system sizes. For larger system sizes, the savings in communication time is outperformed by the costs of the copy operations. For the (con) case, we consider a typical example and try to improve the poor speedup values in the next section.

6 Application to a Discretization Problem

Systems of ODEs with right hand side function of case (con) arise when solving a time dependent partial differential equation by spatial discretization. A typical example is the Brusselator equation which we solve numerically in this section.

6.1 The Brusselator Equation

The Brusselator equation is a nonlinear partial differential equation from chemical kinetics that describes the reaction of two chemical substances [9]. We consider a particular Brusselator equation, the following reaction–diffusion equation [6]:

$$\frac{\partial u}{\partial t} = 1 + u^2 v - 4.4u + \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{22}$$

$$\frac{\partial v}{\partial t} = 3.4u - uv^2 + \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{23}$$

$$0 \le x \le 1, \qquad 0 \le y \le 1, \qquad t \ge 0, \qquad \alpha = 2 * 10^{-3} \tag{24}$$

with Neumann boundary conditions

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

and initial conditions

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

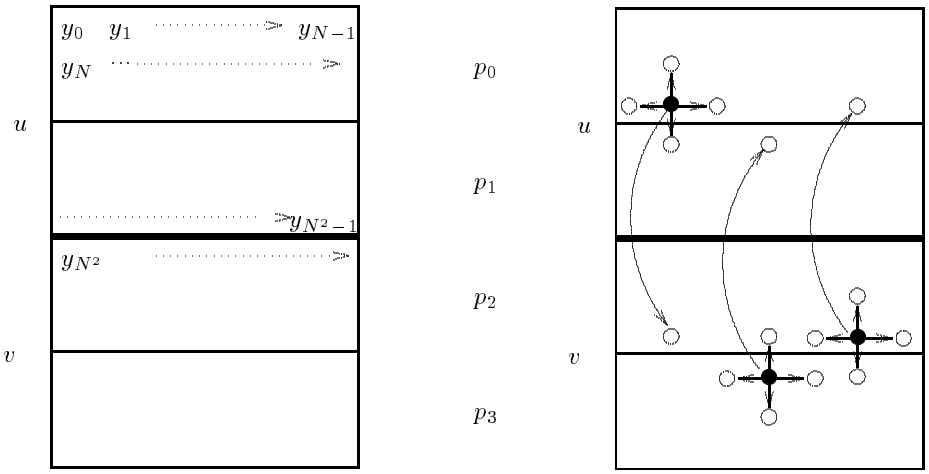

Figure 11: Access structure of the Brusselator function.

The standard discretization of the spatial derivatives on a uniform grid with mesh size $1/(N-1)$ leads to the ODE of dimension $2N^2$.

$$\frac{du_{ij}}{dt} = 1 + u_{ij}^2 v_{ij} - 4.4u_{ij} + \alpha(N-1)\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}\right) \quad (25)$$

$$\frac{dv_{ij}}{dt} = 3.4u_{ij} - u_{ij}^2 v_{ij} + \alpha(N-1)\left(v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1} - 4v_{i,j}\right) \quad (26)$$

The boundary conditions result in:

$$u_{0,j} = u_{2,j}, \quad u_{N+1,j} = u_{N-1,j}, \quad u_{i,0} = u_{i,2}, \quad u_{i,N+1} = u_{i,N_1}$$

6.2 Parallel Solution of the Brusselator Equation

Explicit Runge-Kutta methods are the adequate numerical method for the solution of the spatially discretized Brusselator equation [6]. For the parallel solution, we use the third parallelized version of the IRK method presented in Figure 3.

As mentioned before, this algorithm does not attain good speedups for Brusselator like equations. This is mainly caused by the costs of the broadcast operation (*) in algorithm (C) in Figure 3 after each corrector iterations step. The data exchange is necessary for the numerical correctness of the method and, thus, no communication phase could be omitted in the general case. But for a specific application, it is possible to reduce the time needed for each communication phase by investigating the necessary updating before starting the next iteration.

6.2.1 Application Specific Communication

In the general case, the communication is realized by a multi-broadcast operation, i.e. each processor executes a broadcast operation, such that the whole iteration vector of size $n * s$ is available on each processor after the communication. According to the access structure of the Brusselator equation, it is possible to replace this communication by a more cost–effective data transmission.

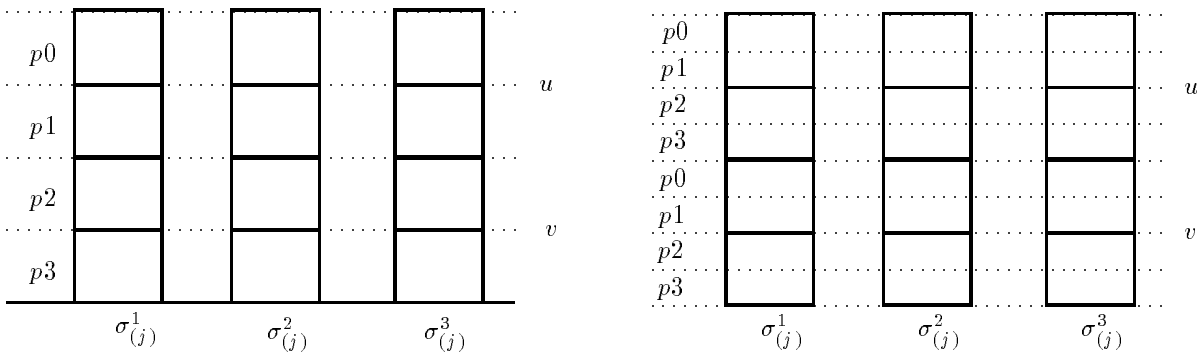

Figure 12: Cyclic and double cyclic data distribution for the Brusselator function. The dotted lines indicate the data domains of the processors. The size of the system is $s \cdot 2N^2$. The figure shows the case $s = 3$.

For the Brusselator equation, the solution vector y of system (II) (and also the iteration vectors $\sigma^l_{(j)}$ and y_κ) have the form $y = (u, v)$ with

$$y(i * N + j) = u(i, j), \quad \text{and} \quad y(N^2 + i * N + j) = v(i, j), \quad 0 < i, j \leq N,$$

see Figure 11. Thus, we can illustrate the cyclic data distribution of algorithm (C) as shown in Figure 12.

The right hand side function $f = (f_1, \ldots, f_n)$, $n = 2 * N^2$, of the Brusselator equation (25) and (26) only needs a few of the updated values of $\sigma^l_{(j)}$ or y_κ, see Figure 11. The resulting data exchange of a processor q is given in the next lemma that implies algorithm (D). (The processors are numbered consecutively by $q = 0, \ldots, p-1$.)

Lemma 7 *If p is even and $N \geq p/2$, it is sufficient that processor q communicates with at most three processors. Those processors are:*

$$\begin{array}{rlll}
 & q+1, \quad q+p/2 & \textit{for} & q = 0 \\
q-1, & q+1, \quad q+p/2 & \textit{for} & 0 < q < p/2-1 \\
 & q-1, \quad q+p/2 & \textit{for} & q = p/2-1 \\
 & q+1, \quad q-p/2 & \textit{for} & q = p/2 \\
q-1, & q+1, \quad q-p/2 & \textit{for} & p/2 < q < p-1 \\
 & q-1, \quad q-p/2 & \textit{for} & q = p-1
\end{array}$$

Proof: Consider the computation of $u_{i,j}$. To access the neighboring elements of u according to formula (25), processor q has to access elements in the data domain of the neighboring processors $q-1$ and $q+1$. The accessed element $v_{i,j}$ is located in the data domain of processor $q + p/2$. This holds for all cases, also if $p/2$ does not divide N^2. □

(D) Algorithm with reduced communication: The multi–broadcast (*) in the parallel IRK algorithm (C) in Figure 3 is replaced by more cost-effective single-to-single transmissions:

```
if (q ≠ 0 and q ≠ p/2) send local elements of σ^l_(j) to q − 1 ;
if (q ≠ p/2 − 1 and q ≠ p − 1) send local elements of σ^l_(j) to q + 1 ;
if (0 ≤ q < p/2) send local elements of σ^l_(j) to q + p/2 ;
if (p/2 ≤ q < p) send local elements of σ^l_(j) to q − p/2 ;
```

6.2.2 Application Specific Data Distribution

A reduction of the number of processors that participate in the communication phase can be achieved by changing the data distribution to a double-cyclic blockwise distribution which allocates each of the u and v data blocks to to all processors, see Figure 12. The following Lemma describes the resulting communication that is used in algorithm (E).

Lemma 8 *If p is even and $N \geq p/2$, it is sufficient that processor q communicates with at most two processors. Those processors are:*

$$\begin{array}{rll} q+1, & \textit{for} & q = 0 \\ q-1, q+1, & \textit{for} & 0 < q < p/2-1 \\ q-1, & \textit{for} & q = p/2-1 \\ q+1, & \textit{for} & q = p/2 \\ q-1, q+1, & \textit{for} & p/2 < q < p-1 \\ q-1, & \textit{for} & q = p-1 \end{array}$$

Proof: Consider e.g. the computation of $u_{i,j}$. Because of the double cyclic data distribution, each processors produces the data from the v data block needed in the next iteration. □

(E) Algorithm with double-cyclic distribution and reduced communication: The parallel IRK algorithm (C) is used with double cyclic data distribution such that each data block u and v is distributed blockwise among the processors. The multi–broadcast (*) in the parallel IRK method in Figure 3 is replaced by a data exchange with at most two other processors. Compared to algorithm (D), the number of transmitted messages is increased because each processor now has to send two (smaller) blocks (for u and v) to each neighboring processor, see Figure 12.

```
if (q ≠ 0 and q ≠ p/2) send local elements of σ_(j)^l to q − 1 in 2 pieces;
if (q ≠ p/2 − 1 and q ≠ p − 1) send local elements of σ_(j)^l to q + 1 in 2 pieces;
```

6.3 Numerical Results

We have implemented the presented methods (D) and (E) for the solution of the Brusselator equation on an Intel iPSC/860. Figure 13 shows typical solutions for the resulting concentrations of the two chemical substances.

For the implementation on the Intel iPSC/860, we again use a 3–stage Radau method [6] of order $p = 5$ as corrector with $m = 4$ corrector iterations. Figures 14, 15, and 16 contain tables with the measured global execution times and diagrams with the measured speedup values for for $p = 4$, $p = 8$, and $p = 16$ processors. The given speedup values are obtained by comparing the parallel global execution times for algorithm 3 with the global execution time of the sequential program for system (I).

According to Figures 14, 15, and 16, the implementations of algorithms (D) and (E) result in larger speedup values than the original algorithm (C). Although the speedup values are increased by a factor of about 2.5 for 8 or 16 processors, they are by no means impressive. For $p = 16$, a reduction of the efficiency can be observed, if the data size is not increased. The fact that in algorithm (E) less data have to be transmitted does not result in considerably larger speedup values.

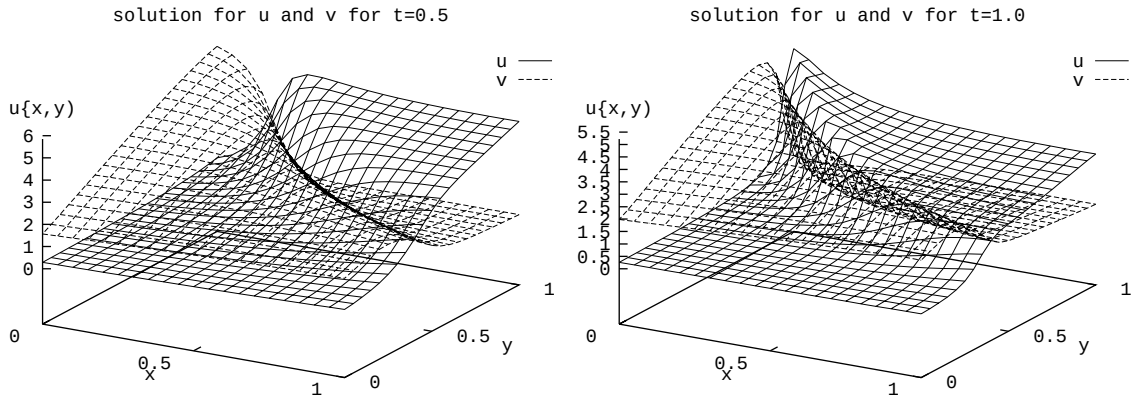

Figure 13: Solution of the Brusselator equation for $t = 0.5s$ and $t = 1.0s$. A discretization of $N = 21$ has been used for the figure.

N	n	(C)	(D)	(E)
11	242	0.044	0.044	0.053
21	882	0.143	0.096	0.116
31	1922	0.213	0.200	0.221
41	3362	0.383	0.336	0.357
51	5202	0.626	0.505	0.533
61	7442	0.891	0.697	0.750
71	10082	1.176	0.941	0.996
81	13122	1.524	1.235	1.282
91	16562	2.049	1.633	1.619

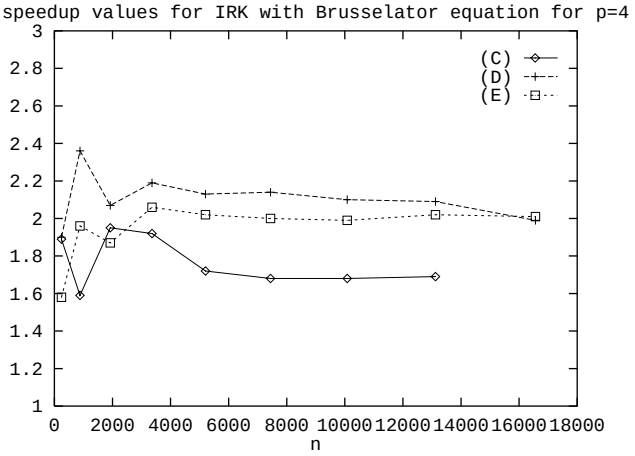

Figure 14: Measured running times in seconds and speedup values on 4 processors for one step of the IRK method applied to the Brusselator function.

N	n	(C)	(D)	(E)
11	242	0.047	0.038	—
21	882	0.113	0.079	0.091
31	1922	0.215	0.144	0.148
41	3362	0.352	0.236	0.229
51	5202	0.551	0.344	0.321
61	7442	0.760	0.493	0.453
71	10082	1.006	0.656	0.596
81	13122	1.336	0.844	0.771
91	16562	1.747	1.080	0.991

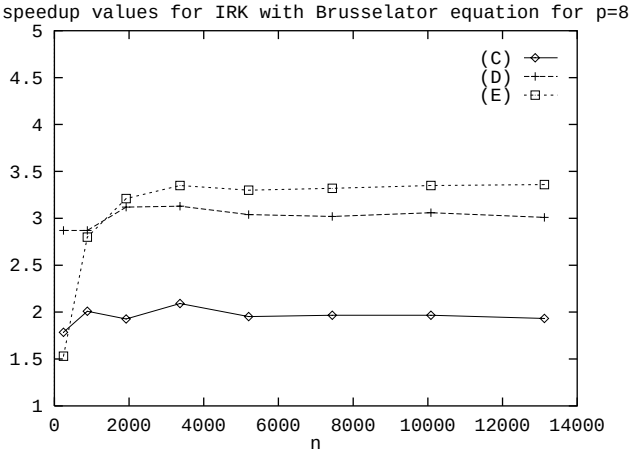

Figure 15: Measured running times in seconds and speedup values on 8 processors for one step of the IRK method applied to the Brusselator function.

N	n	(C)	(D)	(E)
11	242	0.052	—	—
21	882	0.113	0.057	0.075
31	1922	0.211	0.095	0.108
41	3362	0.341	0.151	0.167
51	5202	0.505	0.217	0.266
61	7442	0.714	0.282	0.306
71	10082	0.953	0.384	0.399
81	13122	1.233	0.514	0.507
91	16562	1.610	0.643	0.640

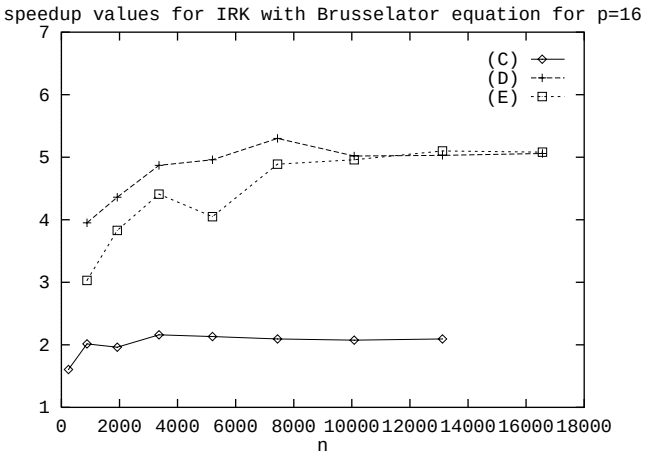

Figure 16: Measured running times in seconds and speedup values on 16 processors for one step of the IRK method applied to the Brusselator function.

6.4 Interpretations of the Numerical Experiments

For the interpretation of the observed phenomena, we use the timing model for the iPSC/860 presented in Section 3.

Lemma 9 *The speedup S_D of algorithm (D) is smaller than the speedup S_E of algorithm (E) if*

$$\tau < \frac{n}{p} t_c \tag{27}$$

Proof: The speedup values for algorithms (D) and (E) are

$$\begin{aligned} S_D &= \frac{t_{(I),seq}}{t_C + 3t_{s\_s}\left(\frac{n}{p}\right) + t_{m\_broad}\left(\frac{n}{p}\right)} \\ S_E &= \frac{t_{(I),seq}}{t_C + 4t_{s\_s}\left(\frac{n}{2p}\right) + t_{m\_broad}\left(\frac{n}{p}\right)} \end{aligned}$$

where $t_{(I),seq}$ is the computation time for the sequential method and t_C is the computation time for algorithm (C) ($t_C = t_D = t_E$). Using the formula for t_{m_broad}, we get that $S_D < D_E$ if

$$4\left(t_c \frac{n}{2p} + \tau\right) < 3\left(t_c \frac{n}{p} + \tau\right)$$

□

Remarks to Lemma 9:

1. According to Lemma 9, algorithm (E) outperforms algorithm (D), only if the saving that results from transmitting fewer data elements ($t_c n/p$) is larger than the additional startup time τ. For the iPSC/860, we get for 8 processors: $S_D < S_E$ if $n > 3589$.

2. One possibility to save startup times for algorithm (E) would be to pack the messages such that only 2 instead of 4 single–to–single transmissions with larger message sizes n/p (instead of $n/2p$) have to be performed. But this requires additional packing and unpacking operations, see Section 4, and does not increase the attained speedup values.

3. Another possibility to save communication time results from the observation that processor q only needs N data elements from each of its neighboring processors. But because the startup time dominates the communication time, this only leads to an improvement for large system sizes.

The following lemma shows that the speedup values cannot be increased considerably. This is shown by examining the time to execute one corrector iteration and determining the efficiency.

Lemma 10 *Let $T_{(II),seq}$ be the sequential time for one corrector iteration according to algorithm (C). The efficiency of iteration (8) of algorithms (C), (D) or (E) is $1/(1+co(p))$ with the communication overhead ratio*

$$co(p) = \frac{T_{comm}(n,p)}{T_{(II),seq}} p = \frac{p}{((2s+1)t_{op} + T_{Bruss})n} \cdot T_{comm}(n,p)$$

T_{Bruss} denotes the time to evaluate the Brusselator function and T_{comm} is the communication time for each iteration step. T_{comm} depends on n, p, and the used algorithms (C)/(D)/(E).

Proof: The computation time for the iteration of equation (8) is

$$(ms(2s+1)t_{op} + smT_{Bruss})\frac{n}{p}.$$

The communication time is $smT_{comm}(n,p)$ with T_{comm} according to the used algorithm (C), (D) or (E). Thus,

$$co(p) = \frac{smT_{comm}(n,p)}{ms((2s+1)t_{op} + T_{Bruss})n}\, p$$

□

Remarks to Lemma 10:

For the algorithm (E) with the lowest communication time ($2n/p \cdot t_c + 4\tau$), we get

$$co(p) = \frac{2nt_c + 4\tau p}{(2s+1)t_{op} + T_{Bruss})n}.$$

On the iPSC/860, we have $t_{op} \approx 0.5\mu s$, $T_{Bruss} \approx 7.5\mu s$. For $s = 3$, we get

$$co(p) = \frac{0.78n + 4 \cdot 175 \cdot p}{11n}.$$

For $p = 8$ and $n = 10000$, we have efficiency $1/(1+co(p)) \approx 0.92$ for the corrector iteration. On the other hand, for algorithm (C), we get

$$co(p) = \frac{a_{t_c,\tau}(p)n + b_\tau(p)p}{11n}.$$

For $p = 8$, this leads to $co(p) > 1$ and therefore we have efficiency < 0.5. The example illustrates that we really have improved the communication of the corrector iteration and thus, it is no longer responsible for the insufficient speedup values of algorithm (E). Rather, other effects cause the small speedup values:

- The multi–broadcast operation at the end of each macrostep (that cannot be improved because of the numerical correctness) is quite expensive.
- The stepsize control according to equation 5 and 10 is executed by each processor.
- The speedup values are computed by comparing the parallel algorithms not with their corresponding sequential algorithm, but with a sequential algorithm that has a smaller execution time.

7 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 consider the iterated Runge–Kutta methods and describe three parallel algorithms that differ in the order of the function evaluation and/or the data distribution on the DMM. A detailed runtime analysis compares the proposed algorithms and is used to select the most promising one for a real implementation. The runtime simulations do not confirm the intuitively expected behavior of the runtime but the observed phenomena can be explained by a theoretical runtime analysis. According to the suggestion of the theoretical investigation, we have implemented the algorithm with delayed function evaluation and cyclic data distribution on the Intel iPSC/860. The implementation confirms that the predicted runtime and speedup values are quite accurate. This shows that the used runtime prediction technique can be successfully used to compute the speedup values for a given parallel algorithm on a real parallel machine *before* implementing it.

We investigate the performance of the method for a typical example that results from the discretization of a reaction–diffusion equation. We show that the original method cannot achieve good speedup values for this application, but that the attained speedup values can be increased considerably by taking advantage of the access structure of the resulting right hand side function f. Nevertheless, the resulting efficiency is not impressing, but they cannot be improved because of the necessary communication. Other known numerical methods to solve initial value problems that are suited for a parallel execution like the extrapolation methods have similar communication behavior and do not result in a better performance [12] [13].

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] A. Bingert, A. Formella, A.M. Müller, and W.J. Paul. Isolating the Reasons for the Performance of Parallel Machines on Numerical Programs. In *International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction*, pages 34–64, 1993.

[4] M.A. Franklin. Parallel Solution of Ordinary Differential Equations. *IEEE Transactions on Computers*, C-27(5):413–420, 1978.

[5] C.W. Gear. Parallel Methods for Ordinary Differential Equations. Technical Report UIUCDCS-R-87-1369, Department of Computer Science, University of Urbana–Champaign, August 1987.

[6] 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.

[7] D. Hutchinson and B.M.S. Khalaf. Parallel Algorithms for Solving Initial Value Problems: Front Broadening and Embedded Parallelism. *Parallel Computing*, 17:957–968, 1991.

[8] S.L. Johnsson and C.T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[9] R. Lefever and G. Nicolis. Chemical Instabilities and Sustained Oscillations. *J. Theor. Biol.*, 30:267–284, 1971.

[10] W.L. Miranker and W. Liniger. Parallel Methods for the Numerical Integration of Ordinary Differential Equations. *Mathematics of Computation*, 21(99):303–320, 1967.

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

[12] 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.

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

[14] G. Rünger. *Über ein Schrödinger-Poisson-System*. PhD thesis, Köln, 1989.

[15] 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.

[16] 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.

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

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

[19] P.B. Worland. Parallel Methods for the Numerical Solution of Ordinary Differential Equations. *IEEE Transactions on Computers*, 25(10):1045–1048, 1976.

[20] P.B. Worland. Parallel Methods for ODEs with Improved Absolute Stability Boundaries. *Journal of Parallel and Distributed Computing*, 18(1):25–32, 1993.