Scalable Parallel RK Solvers for ODEs Derived by the Method of Lines

Matthias Korch and Thomas Rauber

University of Bayreuth, Faculty of Mathematics and Physics
{matthias.korch, rauber}@uni-bayreuth.de

Abstract. This paper describes how the specific access structure of the Brusselator equation, a typical example for ordinary differential equations (ODEs) derived by the method of lines, can be exploited to obtain scalable distributed-memory implementations of explicit Runge-Kutta (RK) solvers. These implementations need less communication and therefore achieve better speed-ups than general explicit RK implementations. Particularly, we consider implementations based on a pipelining computation scheme leading to an improved locality behavior.

1 Introduction

Several approaches towards the parallel solution of ODEs exist. These include extrapolation methods [8], relaxation techniques [2], multiple shooting [6], and iterated Runge-Kutta methods, which are predictor-corrector methods based on implicit Runge-Kutta methods [9,14]. Two-step RK methods based on the computation of s stage approximations are proposed in [10]. The approach exploits parallelism across the method and yields good speed-ups on a shared address space machine. A good overview of approaches for the parallel execution of ODE solution methods can be found in [2,3,4]. Most of these approaches are based on the development of new numerical algorithms with a larger potential for a parallel execution, but with different numerical properties than the classical embedded RK methods.

In this paper, we consider the solution of initial value problems (IVPs)

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \qquad \mathbf{y} : \mathbb{R} \to \mathbb{R}^n, \quad \mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n, \tag{1}$$

by explicit RK methods. In particular, we concentrate on ODEs derived from partial differential equations (PDEs) with initial conditions by discretizing the spatial domain using the method of lines. We choose the 2D-Brusselator equation [5] that describes the reaction of two chemical substances as a representative example for such ODEs. Two unknown functions u and v represent the concentration of the substances. A standard five-point-star discretization of the spatial derivatives on a uniform $N \times N$ grid with mesh size $1/(N-1)$ leads to an ODE system of dimension $2N^2$ for the discretized solution $\{U_{ij}\}_{i,j=1,\dots,N}$ and $\{V_{ij}\}_{i,j=1,\dots,N}$

$$\begin{aligned} \frac{dU_{ij}}{dt} &= 1 + U_{ij}^2 V_{ij} - 4.4 U_{ij} + \alpha(N-1)^2 \left(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{i,j}\right), \\ \frac{dV_{ij}}{dt} &= 3.4 U_{ij} - U_{ij}^2 V_{ij} + \alpha(N-1)^2 \left(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{i,j}\right), \end{aligned} \tag{2}$$

H. Kosch, L. Böszörményi, H. Hellwanger (Eds.): Euro-Par 2003, LNCS 2790, pp. 830–839, 2003.
© Springer-Verlag Berlin-Heidelberg 2003

which is a non-stiff ODE system for $\alpha = 2 \cdot 10^{-3}$ and appropriate values of N. Non-stiff ODE systems can be solved efficiently by explicit RK methods with stepsize control using embedded solutions. An embedded RK method with s stages which uses the argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ to compute the two new approximations $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ from the two previous approximations η_κ and $\hat{\eta}_\kappa$ is represented by the computation scheme

$$\mathbf{w}_l = \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i), \quad l = 1, \ldots, s,$$

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{f}(x_\kappa + c_l h_\kappa, \mathbf{w}_l), \quad \hat{\eta}_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} \hat{b}_l \mathbf{f}(x_\kappa + c_l h_\kappa, \mathbf{w}_l). \tag{3}$$

The coefficients a_{ij}, c_i, b_i, and $\hat{b}_i$ are determined by the RK method used.

If the right hand side function $\mathbf{f}$ of the ODE system is assumed to have an arbitrary dependence structure, in general, the process of computing the argument vectors is sequential, and only minor program modifications are possible to increase locality [12]. The discretized Brusselator equation—similar to other ODEs derived from PDEs—has a loosely coupled dependence structure that is induced by the five-point-star discretization. Thus, the evaluation of component U_{ij} only accesses the components V_{ij} and U_{ij}, $U_{i+1,j}$, $U_{i,j+1}$, $U_{i-1,j}$, $U_{i,j-1}$, if available. Similarly, the evaluation of component V_{ij} uses the components U_{ij} and V_{ij}, $V_{i+1,j}$, $V_{i,j+1}$, $V_{i-1,j}$, $V_{i,j-1}$, if available. When such special properties of $\mathbf{f}$ are known, they can be exploited to obtain faster solvers for this particular class of ODEs. In [7], a sequential pipelining computation scheme has been proposed that leads to an improved locality behavior resulting in a significant reduction of the execution time for different processors. In this paper, we develop parallel implementations for a distributed address space that exploit special properties of the right hand side function $\mathbf{f}$. In contrast to a straightforward implementation that uses global communication operations, these implementations are based on single transfer operations only. The new implementations try to hide the communication costs as far as possible by overlapping communication and computation. As a result, these parallel implementations show good scalability even for large numbers of processors. We demonstrate this for an UltraSPARC II SMP, a Cray T3E-1200 and a Beowulf cluster.

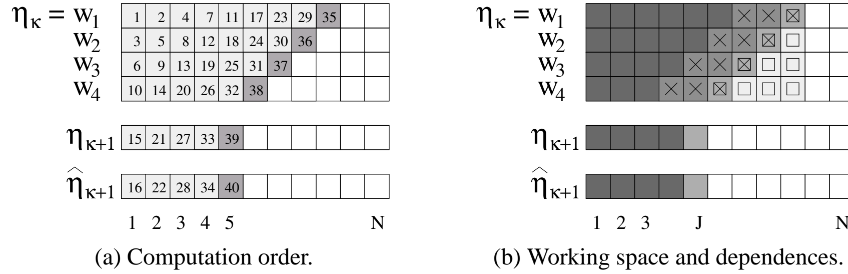
2 Exploiting Specific Access Structure for Locality Improvement

Though it is possible to improve the locality of RK methods by modifications of the loop structure and other rearrangements of the code [12], we can maximize locality only by exploiting special properties of the problem to be solved. In order to store the components $\{U_{ij}\}_{i,j=1,\ldots,N}$ and $\{V_{ij}\}_{i,j=1,\ldots,N}$ of the Brusselator equation in a linear vector of dimension $n = 2N^2$, different linearizations can be used. In our implementations, we use the *mixed row-oriented storage scheme* that reduces the maximum distance of the components accessed in the function evaluation of a single component:

$$U_{11}, V_{11}, U_{12}, V_{12}, \ldots, U_{ij}, V_{ij}, \ldots U_{NN}, V_{NN}. \tag{4}$$

Here, the evaluation of function component f_l accesses the argument components $l - 2N, l-2, l, l+1, l+2, l+2N$ (if available) for $l = 1, 3, \ldots, 2N^2 - 1$ and $l - 2N, l - 2, l-1, l, l+2, l+2N$ (if available) for $l = 2, 4, \ldots, N^2$. For this access structure, the most distant components of the argument vector to be accessed for the computation of one component of $\mathbf{f}$ have a distance equal to $2N$.

We consider the division of the mixed row-oriented storage scheme (4) into N blocks of size $2N$. Analyzing the data dependences occurring during the function evaluations, we can derive a pipelining computation scheme that computes all argument vectors during a single diagonal sweep [7]. Figure 1(a) illustrates the computation order of the pipelining computation scheme, and Fig. 1(b) shows the working space and the dependences of one pipelining step executed to compute block J of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. In Fig. 1(b), apart from the blocks completed during the pipelining step, all blocks are highlighted that are accessed during this step. Blocks required for function evaluations are marked by crosses and blocks updated using results of function evaluations are marked by squares. The working space of one pipelining step consists of $\Theta(s^2)$ blocks of size $2N$. This corresponds to a total of $\Theta(s^2 N)$ components. In contrast, at each stage a general implementation potentially accesses one complete argument vector during the function evaluations and all succeeding argument vectors in order to update the partial sums they hold. This corresponds to a working space of $\Theta(sn)$, which is equal to $\Theta(s \cdot 2N^2)$ in the case of the Brusselator function. The reduction of the working space by the pipelining computation scheme leads to increased locality and thus to better execution times on different sequential machines [7].



(a) Computation order. (b) Working space and dependences.

Fig. 1. Illustration of one pipelining step.

3 Exploiting Specific Access Structure for Parallelization

3.1 General Parallel Implementation

The general parallel implementation (Fig. 2) based on computation scheme (3) computes the argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ sequentially one after another. To exploit parallelism, the computation of the components of the argument vectors is distributed equally among the processors. But since the access structure is not known in advance, it must be assumed that each component of $\mathbf{f}$ accesses all components of the argument vector. Therefore, the argument vectors must be made available by multibroadcast operations.

```
multibroadcast(η_κ);
for (j = first_component; j ≤ last_component; j++) {
    compute F = h f_j(t + c_1 h, η_κ);
    set w_i[j] = η_κ[j] + a_i1 F for i = 2, . . . , s
    η_κ+1[j] = η_κ[j] + b_0 F; η̂_κ+1[j] = η_κ[j] + b̂_0 F; }
for (l = 2; l ≤ s; l++) {
    multibroadcast(w_l);
    for (j = first_component; j ≤ last_component; j++) {
        compute F = h f_j(t + c_l h, w_l);
        update w_i[j] += a_il F for i = l + 1, . . . , s
        η_κ+1[j] += b_l F; η̂_κ+1[j] += b̂_l F; }}
perform error control and stepsize selection
```

Fig. 2. Pseudocode of one time step of a general parallel RK implementation for arbitrary right hand side functions $\mathbf{f}$.

Usually, the execution time of multibroadcast operations increases linearly with the number of participating processors [15]. Therefore, this implementation is not scalable to a large number of processors. Previous experiments have shown that for ODEs resulting from the spatial discretization of PDEs only a limited speed-up can be expected (e.g., DOPRI8(7) on Intel Paragon obtained 5.1 on 8 processors, 4.6 on 32 processors).

3.2 Parallel Blockwise Implementation

The special structure of Brusselator-like functions allows the derivation of more efficient implementations by applying the division of argument vectors into blocks of size $2N$. Thus, N/p adjacent blocks of each argument vector are assigned to each processor for computation.

One possible parallel blockwise implementation is illustrated in Fig. 3. As the general implementation (Fig. 2), this implementation processes the stages successively. But in contrast to the general implementation, no global communication is necessary. Since only the blocks $J-1$, J, and $J+1$ of the previous argument vector are required to compute block J of one argument vector, it is sufficient to send the first and the last block of a processor's part of each argument vector to its predecessor and its successor, respectively, to satisfy the dependence structure of the function. This can be done using single transfer operations like `MPI_Isend()` and `MPI_Irecv()`. These operations usually have execution times consisting of a constant startup time and a transfer time increasing linearly with the data size to be transferred. Consequently, the blockwise implementation has communication costs invariant to the number of processors.

Using the order $\{\mathit{first_block}+1,\ldots,\mathit{last_block}-1,\mathit{last_block},\mathit{first_block}\}$ to compute the blocks of a stage, the communication time can be hidden completely if the neighboring processors send off their first block and their last block of the preceding argument vector by a non-blocking send operation directly after their computation is finished and if the time to compute the $N/p-2$ inner blocks is longer than the time needed to send two blocks through the network. The overall communication overhead in this ideal case consist of $2s$ times the startup time of the send operation and $2s$ times the startup time of the receive operation. If it is not possible to overlap communication and computation completely, the communication time increases by a fraction of $2s$ times the transfer time of $2N$ floating point values.

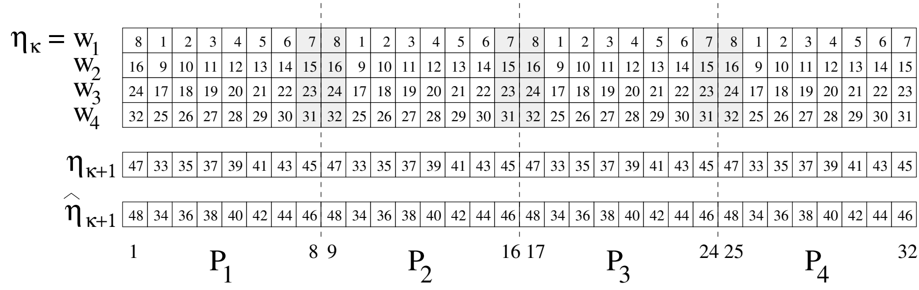


Fig. 3. Illustration of the parallel blockwise implementation.

3.3 Parallel Pipelining

Though the blockwise implementation has reduced communication costs, it does not exploit locality optimally. The pipelining computation scheme [7] allows to derive parallel implementations with similar communication costs but higher locality.

At each time step, the processors start the computation of the blocks $(\textit{first_block} + i-1), \ldots, (\textit{last_block}-i+1)$ of the argument vectors $\mathbf{w}_i$, $i = 1, \ldots, s-1$, in pipelining order. To ensure that each processor computes at least one block of $\mathbf{w}_s$, we assume that at least $2s$ blocks are assigned to each processor. In the second phase, the processors finalize their pipelines at both ends of their data area. To provide efficient communication, neighboring processors use a different order to finalize the two ends of their pipelines. Thus, when processor j finalizes the side of its pipeline with higher index, processor $j + 1$ finalizes the side of its pipeline with lower index simultaneously, and vice versa. Figure 4 illustrates the parallel pipelining scheme. The numbers determine the computation order of the blocks. Blocks accessed during the finalization phase and blocks transmitted between processors are highlighted.

Communication is first required during the finalization of the pipelines, because function evaluations of the first and the last blocks access blocks stored in neighboring processors. When processor j computes its last block of argument vector $\mathbf{w}_i$, $i = 2, \ldots, s$, it needs the first block of argument vector $\mathbf{w}_{i-1}$ of processor $j + 1$, and vice versa. Again, we use non-blocking single send and receive operations in this transfer.

If the processors work simultaneously, the computation of blocks stored in the neighboring processor that are required to perform function evaluations is finished before they are needed. In fact, one diagonal across the argument vectors is computed between the time when processor $j + 1$ finishes its first block of $\mathbf{w}_{i-1}$ and the time when processor j needs this block to compute its last block of $\mathbf{w}_i$, and vice versa. This time can be used to transfer the data required between the processors, thus overlapping communication and computation. But as the finalization phase proceeds, the length of the diagonals decreases from $s - 1$ to 1. Therefore, it is usually not possible to hide the transfer times at the end of the finalization phase completely. As a result, the communication costs of the parallel pipelining implementation are similar to the blockwise implementation, but the fraction of the transfer times that cannot be overlapped by computations is usually larger.

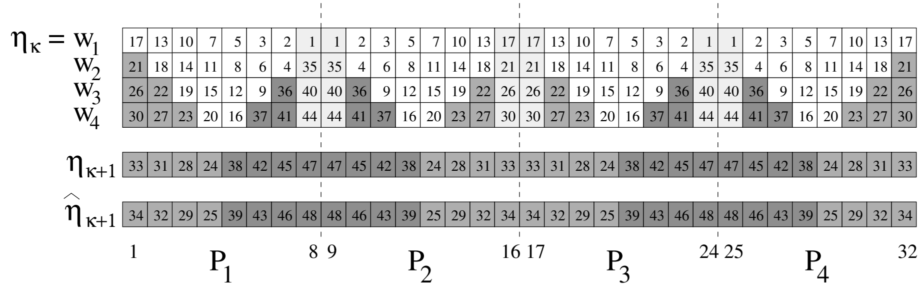


Fig. 4. Illustration of the parallel pipelining implementation (I).

3.4 Parallel Pipelining with Alternative Finalization

The start-up times, which largely determine the communication costs of the specialized parallel implementations previously described, can be quite expensive on some machines. Therefore, it is reasonable to investigate different implementations of parallel pipelining which execute fewer communication operations but transfer more data.

Such an implementation can be obtained by letting each processor finalize the higher end of its own pipeline and the lower end of the pipeline of its cyclic successor. Thus, to compute the stages, only one pair of communication operations is necessary to transfer data from a processor to its cyclic predecessor. This communication involves $2s + \sum_{i=2}^{s} i - 1$ blocks of argument vectors, s blocks of $\eta_{\kappa+1}$, s blocks of $\hat{\eta}_{\kappa+1}$, and s additional blocks (information for stepsize control). All in all, $s(s+9)/2$ blocks have to be transferred. The transfer can be started when the pipeline has been initialized, but the data are not needed before the finalization starts. Hence, the diagonal sweep across the argument vectors can be executed in parallel to the data transfer. During this sweep, $(N/p - 2s) \cdot s$ blocks of argument vectors are computed.

During the finalization of the pipelines, the processors compute parts of $\eta_{\kappa+1}$ that are required by their successors to start the next time step. Thus, another communication operation is needed to send s blocks of $\eta_{\kappa+1}$ to this processor. This is only necessary if the step is accepted by the error control, because otherwise the step is repeated using η_κ and, therefore, $\eta_{\kappa+1}$ is not needed (implementation (II)). But by exchanging the appropriate parts of $\eta_{\kappa+1}$ in every step, it is possible to hide part of the transfer time by computation. This presumes that the finalization starts at the lower end of the pipeline of the succeeding processor and proceeds toward the higher end of the pipeline of the processor considered (implementation (III)). Figure 5 illustrates the computation order of both implementations and the data transmitted.

4 Runtime Experiments and Analysis

We have executed several runtime experiments on three different parallel systems. The first machine is a shared-memory multiprocessor equipped with four Sun UltraSPARC II processors at 400 MHz. The second machine is a Cray T3E-1200 equipped with DEC Alpha 21164 processors at 600 MHz. Also, we have performed measurements on the

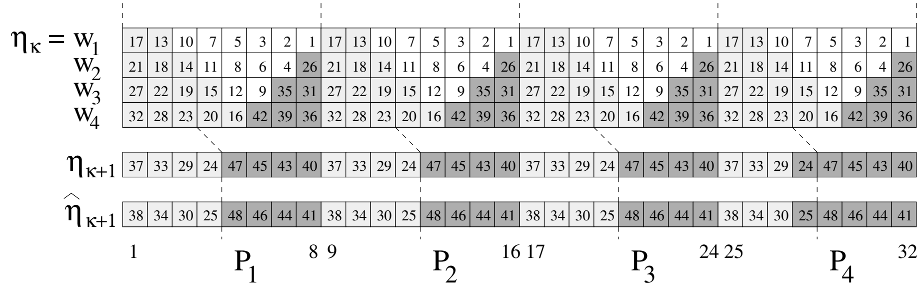


Fig. 5. Illustration of the parallel pipelining implementations (II) and (III).

Chemnitz Linux Cluster (CLiC)—a Beowulf cluster consisting of 528 Pentium III machines at 800 MHz connected by a Fast Ethernet network. All programs have been implemented in C and use double precision. To perform communication, the MPI library [13] has been used, which provides message passing operations for distributed-memory machines but can also be used as a means of communication between processes on a shared-memory system. As RK methods we use the DOPRI5(4) and DOPRI8(7) methods [11]. The speed-up values presented compare the parallel execution times of the different parallel implementations with the execution time of the fastest sequential implementation on the respective machine.

All experiments we performed on the UltraSPARC II machine use the gridsize $N = 384$. The execution times of the general and the pipelining implementations measured for the two RK methods and the integration interval $H = 4.0$ are displayed in Tab. 1. With these parameters, the specialized implementations obtain better speed-ups than the general implementation. For example, using DOPRI5(4) and $H = 4.0$, speed-ups between 3.21 and 3.46 have been measured for the specialized implementations while the general implementation only obtains a speed-up of 1.27. Because of improved locality, the pipelining implementations are faster than the blockwise implementation, particularly for small numbers of processors. But the difference in the speed-ups decreases when the number of processors is increased. The pipelining implementations (II) and (III) both obtain smaller speed-ups than the pipelining implementation (I). The difference increases with the number of processors. Using four processors, the pipelining implementations (II) and (III) are slower than the blockwise implementation for the DOPRI5(4) method.

Table 1. Execution time (in seconds) of the parallel implementations on an UltraSPARC II SMP.

Parameters	DOPRI5(4), H=4.0, N=384				DOPRI8(7), H=4.0, N=384			
Processors	1	2	3	4	1	2	3	4
general	536.07	323.77	307.23	339.90	863.77	511.33	447.13	476.85
blockwise	521.37	264.16	174.50	130.00	832.07	428.89	295.53	218.79
pipelining (I)	445.40	231.57	154.53	124.24	676.48	346.46	234.15	184.53
pipelining (II)	447.15	230.92	162.48	131.73	678.37	350.64	241.82	198.46
pipelining (III)	450.98	232.00	161.99	133.87	674.99	351.78	242.97	197.48

Table 2 shows the execution times measured for the gridsizes $N = 384$ and $N = 896$ and the integration intervals $H = 0.5$ and $H = 4.0$ using DOPRI5(4) and DOPRI8(7) on the Cray T3E. In all these experiments, the specialized implementations obtain much better speed-ups than the general implementation. Particularly, the experiment using DOPRI5(4) with $H = 0.5$ and $N = 896$ where many processors can be used, shows very good scalability for these implementations. Because of the precondition that at least $2s$ blocks must be assigned to every processor, the pipelining implementations are limited to 64 processors in this experiment. Their maximum speed-ups are in the range of 54.15 to 55.39. Using 64 processors, the blockwise implementation obtains a speed-up of 55.28. But since the limitation to at most $N/(2s)$ processors does not apply to it, higher speed-ups can be reached with larger numbers of processors. For example, a speed-up of 109.93 has been measured using 128 processors. The scalability of the general implementation is limited to about 32 processors in this experiment. Its maximum speed-up measured is 12.15.

Using the PCL library [1], we have measured that it takes about 370 cycles to perform the evaluation of one component of the right hand side function $\mathbf{f}$ on the Cray T3E. Since the cycle length of the processors of this machine is 1.67 ns and the maximum network bandwidth is 500 MB/s, at least 24.7 blocks of argument vectors consisting of $2N$ double values must be computed to hide the transfer time of one such block completely. Thus, considering the experiment using $N = 896$, the parallel blockwise implementation can overlap $(N/p - 2)/24.7 \approx 49\,\%$ of the communication in the case of 64 processors and $20\,\%$ when 128 processors are used. To hide all transfer times completely, less than 33 processors should be used. For the parallel pipelining implementations, the fraction of the communication time that can be overlapped by computations is smaller. The first data transfer concerning blocks of η_κ can usually be performed in the background since it can be started at the beginning of the time step. In the case of DOPRI5(4) ($s = 7$), $N = 896$ and $p = 64$, 56 blocks are computed in the meantime. But the first diagonal of the finalization phase only consists of $s-1$ blocks. Thus, using $s = 7$, only $24\,\%$ of the block transfer time is covered. At each stage of the finalization phase, the length of the diagonal computed is reduced by one block. At the end of the finalization phase, only $4\,\%$ of the transfer time is overlapped. In the pipelining implementations (II) and (III), $s(s+9)/2$ blocks are transferred in parallel to the computation of $sN/p - 2s^2$ blocks. That means that 98 blocks are computed during the transfer of 56 blocks in our example with $p = 32$. This corresponds to an overlap of $7\,\%$. If the current step is accepted, the parallel pipelining implementations (II) and (III) additionally need to transfer s blocks of $\eta_{\kappa+1}$ to the succeeding processor before the next time step can be started. In implementation (II), this transfer is not overlapped by computations. But implementation (III) tries to hide part of that transfer time by sending those s blocks in every step. This allows the computation of $\lceil s/2 \rceil (\lfloor s/2 \rfloor + 1)$ blocks in parallel to this transfer. Using $s = 7$, this leads to an overlap of $9\,\%$.

The results of the experiments with the parallel implementations on the CLiC are shown in Tab. 3. Due to the poor performance of global communication operations caused by the slow interconnection network, the execution time of the general implementation cannot be improved by parallel execution. Its execution time increases when two processors are used. The pipelining implementation (I) and the blockwise imple-

Table 2. Execution time (in seconds) of the parallel implementations on a Cray T3E-1200.

Parameters	DOP5(4), H=0.5, N=896			DOP5(4), H=4.0, N=384				DOP8(7), H=0.5, N=896		
Processors	1	64	128	1	24	64	128	1	32	128
general	4359.27	375.12		1135.75	91.80	94.27		11629.54	651.80	
blockwise	4598.50	74.19	37.31	1269.78	54.74	20.85	10.61	12375.57	405.15	101.30
pipelining (I)	4782.31	74.05	n/a	1629.33	54.67	n/a	n/a	12702.03	400.09	n/a
pipelining (II)	4874.49	75.62	n/a	1559.61	55.32	n/a	n/a	12790.38	399.60	n/a
pipelining (III)	5401.25	75.74	n/a	1578.41	55.76	n/a	n/a	12889.71	402.05	n/a

Table 3. Execution time (in seconds) of the parallel implementations on the CLiC.

Parameters	DOP5(5), H=0.5, N=896				DOP5(4), H=4.0, N=384				DOP8(7), H=0.5, N=896		
Processors	1	32	64	128	1	2	24	128	1	32	128
general	1141.49				316.05	666.68			1936.29		
blockwise	986.45	35.15	18.69	10.74	274.50	141.26	13.66	4.26	1724.24	59.35	17.62
pipelining (I)	1005.00	35.71	18.99	n/a	252.66	128.27	12.36	n/a	1747.24	60.29	n/a
pipelining (II)	1343.48	55.42	39.80	n/a	345.06	140.67	25.93	n/a	2164.65	93.86	n/a
pipelining (III)	991.33	54.75	39.20	n/a	247.73	137.61	25.14	n/a	1721.89	92.70	n/a

mentation obtain similar speed-ups as on the Cray T3E. The pipelining implementations (II) and (III) are slower. In the experiment with DOPRI5(4), $H = 0.5$ and $N = 896$, their speed-ups are 52.75 and 53.58, respectively, on 64 processors. The other pipelining implementations only reach 25.17 and 25.55. With 128 processors, the blockwise implementation even obtains a speed-up of 93.30. The part of the communication time that can be overlapped by computations is smaller than on the Cray T3E, because the processors are faster (cycle length 1.25 ns) and the network is slower (100 Mbit/s). Assuming that the evaluation of one component of $\mathbf{f}$ also takes about 370 cycles, 1349 blocks must be computed to hide the transfer time of one single block. Therefore, even in our experiment with the blockwise implementation using $N = 896$ and 64 processors where $49\,\%$ of the transfer times could be covered on the Cray T3E only an overlap of $0.9\,\%$ is possible on the CLiC.

5 Conclusions

We have derived parallel implementations of embedded RK methods for the Brusselator equation, a typical example for ODEs resulting from the spatial discretization of PDEs. These implementations require less communication than general implementations supporting arbitrary right hand side functions $\mathbf{f}$. While the new parallel blockwise implementation already reduces the communication remarkably, the parallel pipelining implementations also exploit locality by reducing the working space of the algorithm.

Runtime experiments confirm that the scalability of the new implementations is better than the scalability of the general implementation. Using 64 processors, speed-ups of about 55 have been measured on the Cray T3E and about 53 on the Beowulf cluster. The results on the Beowulf cluster are particularly interesting as the slow interconnection network (Fast Ethernet) makes it difficult to obtain good speed-ups. Whether the

locality optimizations of the pipelining implementations are successful, depends on the architecture of the machine. If large numbers of processors are used, communication issues usually outweigh the acceleration achieved by locality improvements. Overlapping communication and computations showed improvements on the Cray T3E. But on the Beowulf cluster the slow interconnection network prevented significant improvements.

Acknowledgment

We thank the NIC Jülich for providing access to the Cray T3E and the TU Chemnitz for providing access to the CLiC.

References

1. R. Berrendorf and B. Mohr. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.2)*. Research Centre Jülich, January 2003.
2. K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford Science Publications, 1995.
3. C. W. Gear. Massive Parallelism across Space in ODEs. *Applied Numerical Mathematics*, 11:27–43, 1993.
4. C. W. Gear and Xu Hai. Parallelism in Time for ODEs. *Applied Numerical Mathematics*, 11:45–68, 1993.
5. E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer–Verlag, Berlin, 1993.
6. B. M. S. Khalaf and D. Hutchinson. Parallel Algorithms for Initial Value Problems: Parallel Shooting. *Parallel Computing*, 18:661–673, 1992.
7. M. Korch, T. Rauber, and G. Rünger. Pipelining for locality improvement in RK methods. In *Proc. of 8th Int. Euro-Par Conf. (Euro-Par 2002)*, pages 724–733. Springer (LNCS 2400), 2002.
8. L. Lustman, B. Neta, and W. Gragg. Solution of ordinary differential initial value problems on an Intel Hypercube. *Computer and Math. with Applications*, 23(10):65–72, 1992.
9. S. P. Nørsett and H. H. Simonsen. Aspects of Parallel Runge–Kutta methods. In *Numerical Methods for Ordinary Differential Equations*, volume 1386 of *Lecture Notes in Mathematics*, pages 103–117, 1989.
10. H. Podhaisky and R. Weiner. A class of explicit two-step Runge-Kutta methods with enlarged stability regions for parallel computers. *Lecture Notes in Computer Science*, 1557:68–77, 1999.
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. Optimizing locality for ODE solvers. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 123–132. ACM Press, 2001.
13. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference*. MIT Press, Cambridge, Mass., second edition, 1998.
14. 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.
15. Z. Xu and K. Hwang. Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences. *Parallel Computing*, 22:917–942, 1996.