Locality Optimized Shared-Memory Implementations of Iterated Runge-Kutta Methods

Matthias Korch and Thomas Rauber

University of Bayreuth, Department of Computer Science
{matthias.korch, rauber}@uni-bayreuth.de

Abstract. Iterated Runge-Kutta (IRK) methods are a class of explicit solution methods for initial value problems of ordinary differential equations (ODEs) which possess a considerable potential for parallelism across the method and the ODE system. In this paper, we consider the sequential and parallel implementation of IRK methods with the main focus on the optimization of the locality behavior. We introduce different implementation variants for sequential and shared-memory computer systems and analyze their runtime and cache performance on two modern supercomputer systems.

1 Introduction

Owing to their large computational requirements, parallel solution methods for initial value problems (IVPs) of ordinary differential equations (ODEs), defined by

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

have been addressed by many authors during the last decades. Among the methods considered are extrapolation methods [1], waveform relaxation techniques [2], and iterated Runge-Kutta (IRK) methods [3,4]. An overview can be found in [2]. 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 Runge-Kutta (RK) methods.

In this paper, we consider the parallel and sequential implementation of IRK methods, paying particular attention to the locality of memory references. Based on the classical implicit RK methods

$$\mathbf{Y}_l = \mathbf{y}_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mathbf{F}_i, \quad l = 1, \ldots, s; \quad \mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{F}_l, \tag{2}$$

with the coefficient matrix $A = (a_{ij}) \in \mathbb{R}^{s,s}$, the weight vector $\mathbf{b} = (b_i) \in \mathbb{R}^s$, the node vector $\mathbf{c} = (c_i) \in \mathbb{R}^s$, and $\mathbf{F}_i = \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{Y}_i)$, explicit IRK methods suitable for non-stiff equations introduce the iteration process

$$\mathbf{Y}_l^{(k)} = \mathbf{y}_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mathbf{F}_i^{(k-1)}, \qquad l = 1, \ldots, s, \quad k = 1, \ldots, m. \tag{3}$$

We choose the 'trivial' predictor

$$\mathbf{Y}_l^{(0)} = \mathbf{y}_\kappa, \qquad l = 1, \ldots, s, \tag{4}$$

to start the iteration process and execute a fixed number of $m = p - 1$ corrector steps (3), where p is the order of the underlying implicit RK method (cf. [5]). Two approximations to $\mathbf{y}(t_{\kappa+1})$ of different order, $\mathbf{y}_{\kappa+1}$ and $\hat{\mathbf{y}}_{\kappa+1}$, are then computed by

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{F}_l^{(m)} \quad \text{and} \quad \hat{\mathbf{y}}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{F}_l^{(m-1)}. \tag{5}$$

In comparison to classical RK methods, IRK methods possess a larger potential for parallelism. While, in general, classical explicit RK methods are only suitable for a data-parallel execution exploiting parallelism across the ODE system, IRK methods exhibit an additional degree of method parallelism by enabling an independent computation of the s argument vectors $\mathbf{Y}_l^{(k)}$, $k = 1, \ldots, s$, within each corrector step.

On modern computer systems, which possess deep memory hierarchies, performance and scalability of applications often strongly depend on their locality of memory references. Moreover, large shared-memory systems are often built of physically distributed memory modules, which results in non-uniform memory access times. On such so-called NUMA architectures a good locality behavior is vitally important. Optimizations to increase the locality of memory references have been applied by other authors to many methods from numerical linear algebra (e.g., [6]), but not to ODE solvers. Many popular scientific libraries like LAPACK [7], PHiPAC [8] and ATLAS [9] pay regard to locality and thus can obtain a high efficiency. Modern optimizing compilers [10] try to reorder the instructions of the source program, e.g., of computationally intensive loops [11], to achieve an efficient exploitation of the memory hierarchy. Loop tiling [12] is considered to be one of the most successful techniques. An important analytical model for the effects of loop transformations is described in [13]. Locality optimizations for embedded Runge-Kutta (ERK) methods have been investigated in [14]. [15,16] extend this work by considering parallel implementations and by introducing a pipelining computation scheme which exploits the special access structure of a large class of ODE systems. An investigation of the practical performance of parallel IRK methods for distributed-memory architectures has been presented in [17].

In this paper, we consider parallel IRK implementations for shared-memory architectures. We focus on the optimization of the locality behavior, incorporating our experiences with ERK methods. We present several implementation variants with different loop structures resulting in different memory access patterns. Our focus lies on data-parallel implementations, because they can achieve a higher locality than task-parallel implementations. Moreover, purely task-parallel implementations are only scalable to at most s processors and are therefore not suitable for an execution on large parallel computer systems.

2 Access Distance of a Right-Hand-Side Function

While, in general, the evaluation of one component of the right hand-side-function $\mathbf{f}$, $f_j(t, \mathbf{y})$, may access all components of the argument vector $\mathbf{y}$, many ODE problems only require the use of a limited number of components. In many cases, the components accessed are located nearby the index j. To measure this property of a function $\mathbf{f}$, we make use of the following definitions:

Definition 1. *The* access distance of a component function $f_j(t, \mathbf{y})$, *denoted as* $d(f_j)$, *is the smallest value* b, *such that* f_j *accesses only the subset* $\{y_{j-b}, \ldots, y_{j+b}\}$ *of the components of the argument vector* $\mathbf{y}$.

Definition 2. *The* access distance of a vector function $\mathbf{f}$, *denoted as* $d(\mathbf{f})$, *is the largest access distance of all component functions* f_j *of* $\mathbf{f}$, *i.e.,* $d(\mathbf{f}) = \max_{1 \le j \le n} d(f_j)$.

Definition 3. *We say the access distance of* $\mathbf{f}$ *is* limited *if* $d(\mathbf{f}) \ll n$.

Remark. A function $\mathbf{f}$ with limited access distance $d(\mathbf{f})$ has a banded Jacobian $\frac{\partial \mathbf{f}(t,\mathbf{y})}{\partial \mathbf{y}}$ with bandwidth $2d(\mathbf{f}) + 1$.

3 Equations with Arbitrary Access Structures

The implementation of an IRK method given by Eqs. (3), (4) and (5) leads to a program structure consisting of nested loops. The most time consuming part is the iteration of the corrector steps. Therefore, we will focus our discussion on the loop structure realizing Eq. (3) within one time step. The iteration space of Eq. (3) consists of 4 dimensions:

- $k = 1, \ldots, m$: Iteration over the corrector steps.
- $l = 1, \ldots, s$: Iteration over the vectors $\mathbf{Y}_l^{(k)}$.
- $i = 1, \ldots, s$: Iteration over the summands of $\sum_{i=1}^{s} a_{li} \mathbf{F}_i^{(k-1)}$.
- $j = 1, \ldots, n$: Iteration over the system dimension, i.e., the elements of a vector of dimension n.

Since, in general, each corrector step k depends on the previous corrector step $k-1$, the corrector steps must be computed sequentially, and the k-loop must be the outermost loop. But all other loops are independent. Hence, we are free to choose the loop structure inside the k-loop as it is desirable to obtain a high locality. That means we can interchange, merge and split the l-, i- and j-loops, and loop tiling is also possible. One particular consequence of this is that a computation scheme similar to the pipelining technique presented in [15,16], which uses the j-loop as an outer loop surrounding the l- and the i-loops, can be realized for general problems with arbitrary access structures. Moreover, since the iterations of the l-loop are independent, a pipelining of the stages resulting in a diagonal iteration is not necessary.

The realization of the loop structure affects and is affected by the choice of data structures. In general, at each step k of the iteration process (3), only data from the current iteration k and the preceding iteration $k-1$ is required. Hence, the vectors $\mathbf{Y}_l^{(1)}, \ldots, \mathbf{Y}_l^{(k-2)}$ and $\mathbf{F}_i^{(1)}, \ldots, \mathbf{F}_i^{(k-2)}$ do not have to be kept in memory. Instead, it is sufficient to use the $4s$ vectors $\mathbf{Y}_l^{(\mathrm{cur})}$, $\mathbf{Y}_l^{(\mathrm{prev})}$, $\mathbf{F}_i^{(\mathrm{cur})}$ and $\mathbf{F}_i^{(\mathrm{prev})}$ if we swap the pointers to the respective (prev) and (cur) vectors between the iterations of the k-loop. Further, if we keep the vectors $\mathbf{F}_i^{(\mathrm{cur})}$ and $\mathbf{F}_i^{(\mathrm{prev})}$ in memory, only one additional temporary argument vector $\mathbf{Y} \in \mathbb{R}^n$ is required to compute all $\mathbf{F}_i^{(\mathrm{cur})}$ based on the values of $\mathbf{F}_i^{(\mathrm{prev})}$ if the l-loop is processed sequentially (see implementations (A) and (E) below). However, a task-parallel implementation which processes the iterations of the l-loop simultaneously requires s temporary argument vectors $\mathbf{Y}_1, \ldots, \mathbf{Y}_s \in \mathbb{R}^n$. Another possibility is to store the vectors $\mathbf{Y}_l^{(\mathrm{cur})}$ and $\mathbf{Y}_l^{(\mathrm{prev})}$ instead of the vectors $\mathbf{F}_i^{(\mathrm{cur})}$ and $\mathbf{F}_i^{(\mathrm{prev})}$. Then, a loop structure can be realized which requires only one single scalar variable to handle all results of function evaluations (see implementation (D) and derived implementations below). In addition to the vectors $\mathbf{Y}_l^{(\mathrm{cur})}$ and $\mathbf{Y}_l^{(\mathrm{prev})}$ or $\mathbf{F}_i^{(\mathrm{cur})}$ and $\mathbf{F}_i^{(\mathrm{prev})}$, usually three further vectors of dimension n are required: the two approximation vectors, $\mathbf{y}_{\kappa+1}$ and $\hat{\mathbf{y}}_{\kappa+1}$, and a backup copy of $\mathbf{y}_\kappa$ for the case that the current time step is rejected by the step control.

The choice of data structures is particularly important in parallel implementations. Taking the NUMA character of many modern shared-memory systems into account, distributed data structures corresponding to the physically distributed structure of the memory subsystem are often favorable. Since function evaluations $\mathbf{f}(t, \mathbf{Y})$ may access all components of $\mathbf{Y}$, storing the s argument vectors $\mathbf{Y}_1, \ldots, \mathbf{Y}_s$ separately may increase the working space of a data-parallel implementation and thus lead to a worse locality compared to storing $\mathbf{F}_i^{(\mathrm{cur})}$, $\mathbf{F}_i^{(\mathrm{prev})}$ and only one temporary argument vector $\mathbf{Y}$. But, on the other hand, storing $\mathbf{Y}_1, \ldots, \mathbf{Y}_s$ separately enables alternative loop structures, which can be more efficient for particular problems.

Data-parallel implementations distributing the iterations of the j-loop to different processors can achieve a smaller working space than task-parallel implementations. The reason is that, if we parallelize across the l-loop, the i- and the j-loop of a task-parallel implementation touch all n components of the vectors $\mathbf{F}_i^{(\mathrm{cur})}$ and $\mathbf{F}_i^{(\mathrm{prev})}$. If we parallelize across the i-loop, then the l- and the j-loop access all n components of the vectors $\mathbf{Y}_l^{(\mathrm{cur})}$ and $\mathbf{Y}_l^{(\mathrm{prev})}$. In a data-parallel implementation, on the other hand, which stores the vectors $\mathbf{F}_i^{(\mathrm{cur})}$ and $\mathbf{F}_i^{(\mathrm{prev})}$, a thread uses only n/p components of each vector, where p is the number of processors. But if the argument vectors $\mathbf{Y}_l^{(\mathrm{cur})}$ and $\mathbf{Y}_l^{(\mathrm{prev})}$ are stored, $n/p+2d(\mathbf{f})-1$ components are accessed. Thus, in the latter case, the working space has a similar size as in the task-parallel implementation if the access distance $d(\mathbf{f})$ is large, but it is significantly smaller if the access distance is limited.

Table 1 shows a summary of the implementations we have realized that support equations with arbitrary access structures. All implementation variants have

Table 1. Implementations suitable for problems with arbitrary access structures.

Implement.	Data structures	Loop struct.	Remarks
(A)	$p \times \mathbf{F}_1^{(\mathrm{cur})}, \ldots, \mathbf{F}_s^{(\mathrm{cur})} \in \mathbb{R}^{n/p}$, $p \times \mathbf{F}_1^{(\mathrm{prev})}, \ldots, \mathbf{F}_s^{(\mathrm{prev})} \in \mathbb{R}^{n/p}$, $\mathbf{Y} \in \mathbb{R}^n$	k–l–i–j	vector oriented: inner loops iterate over system dimension; high spatial locality
(E)	$p \times \mathbf{F}_1^{(\mathrm{cur})}, \ldots, \mathbf{F}_s^{(\mathrm{cur})} \in \mathbb{R}^{n/p}$, $p \times \mathbf{F}_1^{(\mathrm{prev})}, \ldots, \mathbf{F}_s^{(\mathrm{prev})} \in \mathbb{R}^{n/p}$, $\mathbf{Y} \in \mathbb{R}^n$	k–l–j–i	exploits temporal locality of the i-loop, i.e., writes to argument vector components
(D)	$\mathbf{Y}_1^{(\mathrm{cur})}, \ldots, \mathbf{Y}_s^{(\mathrm{cur})} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(\mathrm{prev})}, \ldots, \mathbf{Y}_s^{(\mathrm{prev})} \in \mathbb{R}^n$	k–i–j–l	exploits temporal locality of the l-loop, i.e., reads from results of function evaluations
(Dblock)	$\mathbf{Y}_1^{(\mathrm{cur})}, \ldots, \mathbf{Y}_s^{(\mathrm{cur})} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(\mathrm{prev})}, \ldots, \mathbf{Y}_s^{(\mathrm{prev})} \in \mathbb{R}^n$, $p \times \mathbf{F} \in \mathbb{R}^B$	k–i–j–l–jj	similar to (D), but loop tiling of the j-loop with the l-loop
(PipeDe2m)	$\mathbf{Y}_1^{(\mathrm{cur})}, \ldots, \mathbf{Y}_s^{(\mathrm{cur})} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(\mathrm{prev})}, \ldots, \mathbf{Y}_s^{(\mathrm{prev})} \in \mathbb{R}^n$	k–j–i–l	based on (D); j-loop surrounds l- and i-loop; exploits temporal locality of both loops
(PipeDb2m)	$\mathbf{Y}_1^{(\mathrm{cur})}, \ldots, \mathbf{Y}_s^{(\mathrm{cur})} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(\mathrm{prev})}, \ldots, \mathbf{Y}_s^{(\mathrm{prev})} \in \mathbb{R}^n$	k–j–i–jj–l	similar to (PipeDe2m), but loop tiling of the j-loop with the i-loop
(PipeDb2mt)	$\mathbf{Y}_1^{(\mathrm{cur})}, \ldots, \mathbf{Y}_s^{(\mathrm{cur})} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(\mathrm{prev})}, \ldots, \mathbf{Y}_s^{(\mathrm{prev})} \in \mathbb{R}^n$, $p \times \mathbf{F} \in \mathbb{R}^B$	k–j–i–(jj)–l–jj	similar to (PipeDb2m), but loop tiling expanded to the l-loop

been realized in a sequential version and a data-parallel version. Additionally, a task-parallel version of implementation (A) has been realized to compare the locality behavior of task- and data-parallel implementations.

4 Exploiting a Limited Access Distance

Similar to ERK methods [15,16], IRK methods can take advantage of a limited access distance of the function $\mathbf{f}$. While the stages of each corrector step are already decoupled, a limited access distance furthermore allows a reduction of the storage space, a pipelining of the corrector steps and a distributed storage of the argument vectors.

A reduction of the storage space can be achieved by using a loop structure in which the l- and the i-loop are inner loops of the j-loop. Then, a limited access distance allows an overlapping of the matrices $Y^{(k)} = \left(y_{l,j}^{(k)}\right) \in \mathbb{R}^{s,n}$, for $k = 1, \ldots, m$, where $Y^{(k)} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)^T$. This can be realized by choosing a blocksize $B \geq d(\mathbf{f})$ and embedding $Y^{(1)}, \ldots, Y^{(m)}$ into a matrix $\bar{Y} = (\bar{y}_{l,j}) \in \mathbb{R}^{s,(m-1)\cdot 2B+n}$ such that $y_{l,j}^{(k)} := \bar{y}_{l,j+(m-k)\cdot 2B}$ and, hence, $y_{l,j}^{(k)}$, $k = 2, \ldots, m$, points to the same memory location as $y_{l,j+2B}^{(k-1)}$. Implementations (PipeDb1m) and (PipeDb1mt) are based on this strategy (see Table 2).

This optimization reduces the working space of one iteration of the outermost loop (the k-loop) nearly by a factor of 2. But this working space still has a size of $\Theta(sn)$ since the j-, the l- and the i-loop are inner loops of the k-loop. In order to obtain an asymptotically smaller working space of the outermost loop,

Table 2. Implementations specialized in problems with a limited access distance.

Implement.	Data structures	Loop struct.	Remarks
(PipeDb1m)	$p \times \bar{Y} \in \mathbb{R}^{s,m \cdot 2B+n/p}$	k–j–i–jj–l	similar to (PipeDb2m), but the vectors $\mathbf{Y}_l^{(k)}$ are overlapped to reduce space requirements
(PipeDb1mt)	$p \times \bar{Y} \in \mathbb{R}^{s,m \cdot 2B+n/p}$, $p \times \mathbf{F} \in \mathbb{R}^B$	k–j–i–(jj)–l–jj	similar to (PipeDb1m), but loop tiling expanded to the l-loop
(ppDb1m)	$p \times \bar{Y} \in \mathbb{R}^{s,m \cdot 2B+n/p}$	j–k–i–jj–l	based on (PipeDb1m); j- and k-loop are interchanged using a pipelining approach
(ppDb1mt)	$p \times \bar{Y} \in \mathbb{R}^{s,m \cdot 2B+n/p}$, $p \times \mathbf{F} \in \mathbb{R}^B$	j–k–i–(jj)–l–jj	similar to (ppDb1m), but loop tiling expanded to the l-loop

we need to restructure the loops such that the loop with the largest dimension, i.e., the j-loop running from $1, \ldots, n$, becomes the outermost loop. However, if the problem to be integrated has a limited access distance, a pipelining of the corrector steps is possible, which allows an interchange of the k- and the j-loop. This approach is a straight-forward adoption of the pipelining of the stages in ERK methods suggested in [15,16]. It leads to a significant reduction of the working space of the iterations of the outer loop, which now access only $\Theta(sm \cdot 2B)$ distinct vector components. The pipelining of corrector steps has been realized in implementations (ppDb1m) and (ppDb1mt) (see Table 2).

Since many modern shared-memory systems are NUMA systems with a physically distributed memory architecture, a distributed storage of the argument vectors, where each thread keeps a part of each argument vector in its local memory, seems desirable. But in the general case, where the function evaluations $f_j(t, \mathbf{Y})$ may access all components of the argument vector $\mathbf{Y}$, a distributed storage of the argument vectors might not be profitable as it would require an additional level of indirection which maps the indices $j = 1, \ldots, n$ to the associated addresses inside the distributed parts of the vectors. The only alternative would be the replication of the whole argument vector. A limited access distance, however, enables a cost-efficient distributed storage of the argument vectors. Since the function evaluations performed by a processor P on its range of components $J^{(P)} = \left\{ j_{\text{first}}^{(P)}, \ldots, j_{\text{last}}^{(P)} \right\}$ may only access the components $\left\{ j_{\text{first}}^{(P)} - d(\mathbf{f}), \ldots, j_{\text{last}}^{(P)} + d(\mathbf{f}) \right\}$ of the argument vector, the processors can store their own range of components in local vectors, and only $d(\mathbf{f})$ components need to be copied from a processor's predecessor and its successor before a function evaluation can be performed. All of the implementations (PipeDb1m), (PipeDb1mt), (ppDb1m) and (ppDb1mt) make use of a distributed organization of the argument vectors.

5 Experimental Results

Experiments to analyze the scalability and the locality behavior of the implementations have been performed on two of the largest German supercomputer systems, the Jülich Multiprocessor (JUMP) at the NIC Jülich and the High End

Table 3. Execution times (in s) of task- and data-parallel versions of (A).

	JUMP				HLRB II			
	Radau I A(5) 3 Threads		Lobatto III C(8) 5 Threads		Radau I A(5) 3 Threads		Lobatto III C(8) 5 Threads	
N	Data-Par.	Task-Par.	Data-Par.	Task-Par.	Data-Par.	Task-Par.	Data-Par.	Task-Par.
500	8.5	8.9	11.7	12.9	16.4	33.5	23.2	56.6
750	45.4	47.3	62.7	66.0	103.3	184.2	135.5	299.4
1000	142.2	151.0	200.3	210.9	347.8	591.1	474.9	941.7

Table 4. Comparison of the L2 performance of task- and data-parallel versions of (A) on JUMP. The table shows the number of times a cache line in the L1 data cache was reloaded from the local L2 cache, an L2 cache of another chip on the same MCM or from an L2 cache of another chip on a remote MCM.

	Radau I A(5), 3 Threads			Lobatto III C(8), 5 Threads		
Implementation	Local	Same MCM	Other MCM	Local	Same MCM	Other MCM
Data-Parallel	$1.4 \cdot 10^9$	$6.7 \cdot 10^4$	$1.3 \cdot 10^5$	$2.3 \cdot 10^9$	$1.6 \cdot 10^5$	$3.3 \cdot 10^5$
Task-Parallel	$1.4 \cdot 10^9$	$1.0 \cdot 10^7$	$2.3 \cdot 10^7$	$2.3 \cdot 10^9$	$3.0 \cdot 10^7$	$7.2 \cdot 10^7$

System in Bavaria II (HLRB II) at the LRZ Munich. JUMP is a SMP cluster system consisting of 41 IBM eServer pSeries 690 nodes, each equipped with 32 Power4+ cores running at 1.7 GHz. The nodes are built up of 4 multi-chip modules (MCM), where each MCM carries 4 Power4+ chips, and 2 CPU cores are housed in one chip. The HLRB II is an SGI Altix 4700 system. Currently, the system is equipped with 4096 Intel Itanium 2 Madison 9M processors at a clock rate of 1.6 GHz. As an example problem we use BRUSS2D [2], a typical example of a partial differential equation (PDE) discretized by the method of lines. Using an interleaving of the two dependent variables resulting in a mixed-row oriented ordering of the components (BRUSS2D-MIX, cf. [15,16]), this problem has a limited access distance of $d(\mathbf{f}) = 2N$, where the system size is $n = 2N^2$. Most experiments presented in the following have been performed using the 3-stage method Radau I A(5) as the corrector method.

5.1 Task vs. Data Parallelism

Table 3 shows a comparison of the execution times of a data-parallel and a task-parallel version of implementation (A) using the 3-stage method Radau I A(5) and the 5-stage method Lobatto III C(8). In most experiments performed on JUMP, the data-parallel version is slightly faster than the task-parallel version. On HLRB II, where remote memory accesses are more expensive, the higher locality of the data-parallel version evidently outperforms the task-parallel version. An analysis of the cache performance using performance counter data on JUMP points out that the lower performance of the task-parallel version on this machine can be attributed to an exceedingly higher number of loads from L2 caches of other chips (cf. Table 4).

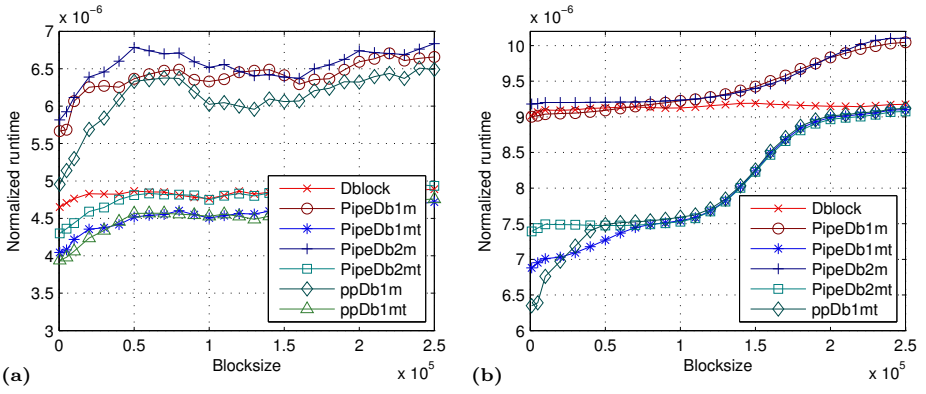

Fig. 1. Normalized sequential runtime as a function of the blocksize measured on **(a)** JUMP and **(b)** HLRB II using BRUSS2D-MIX with Radau I A(5) and $N = 1000$.

5.2 Choosing a Suitable Blocksize

All implementations which employ loop tiling and all implementations specialized in a limited access distance require the selection of a suitable blocksize B. In case of the specialized implementations, B must at least be as large as the access distance $d(\mathbf{f})$. We choose the blocksize based on an experimental evaluation of the normalized sequential runtime, which is the runtime per time step and per ODE component. A blocksize which leads to a minimum of the normalized runtime would be the optimal choice. Figure 1 shows the normalized runtime as a function of the blocksize measured on JUMP and HLRB II using our example problem. On both systems the optimal blocksize is below 10 000 for all implementations. We choose $B = d(\mathbf{f}) = 2N$, i.e., $B = 2000$ for $N = 1000$, since it is the smallest possible value to be used with the specialized implementations. This value is close enough to the optimal value of all implementations that it enables an accurate comparison.

5.3 Performance Analysis and Comparison of the Implementations

As a first step of the performance analysis, we compare the sequential runtimes of the implementations (Table 5). On both machines we observe that the various implementations obtain significantly differing runtimes. Since all implementations are realizations of Eqs. (3), (4) and (5) only differing in the loop structures and the data structures used, the different runtimes mainly result from the different locality behavior of the implementations. On both systems the best general implementation is (PipeDb2mt), which combines a loop structure where the i- and the l-loop run inside the j-loop with loop tiling of both the i- and the l-loop. The specialization in a limited access distance improves the locality and enables a better runtime. Thus, implementation (PipeDb1mt) runs faster than (PipeDb2mt). The best sequential runtime is obtained by the implementation with the smallest working space, (ppDb1mt).

Table 5. Sequential runtimes measured using BRUSS2D-MIX and Radau I A(5).

	JUMP			HLRB II		
Implementation	$N = 500$	$N = 750$	$N = 1000$	$N = 500$	$N = 750$	$N = 1000$
(A)	26.9	153.6	481.6	61.1	330.9	1071.2
(D)	25.8	142.7	465.2	60.5	322.8	1032.1
(Dblock)	24.8	137.2	454.5	54.0	286.8	907.1
(E)	25.2	140.9	459.8	55.4	294.6	950.9
(PipeDb1m)	25.7	139.9	441.1	53.7	286.5	913.4
(PipeDb1mt)	23.1	126.2	413.0	40.9	218.2	698.5
(PipeDb2m)	25.6	138.0	439.4	54.8	291.9	930.6
(PipeDb2mt)	24.0	135.9	436.5	44.0	234.9	750.3
(PipeDe2m)	31.6	182.6	580.1	55.3	294.2	924.3
(ppDb1m)	25.5	136.1	430.0	n/a[1]	n/a[1]	n/a[1]
(ppDb1mt)	23.0	123.5	395.2	37.7	201.9	644.5

1. Not available due to an incorrect translation by the compiler.

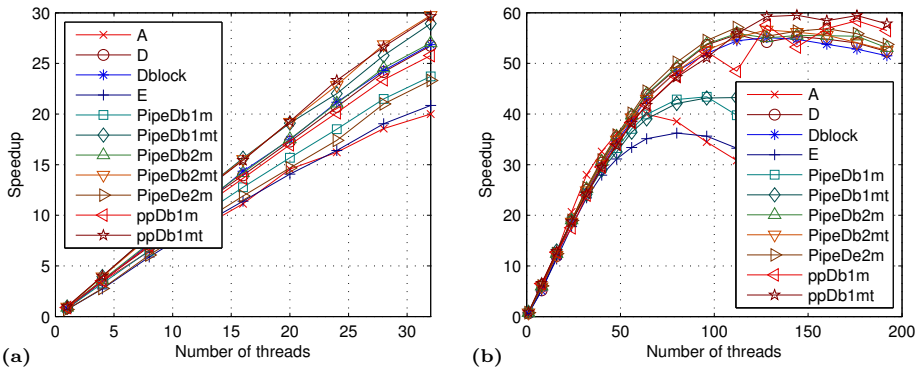

Fig. 2. Speedups measured on **(a)** JUMP and **(b)** HLRB II using BRUSS2D-MIX with Radau I A(5) and $N = 1000$.

To analyze the scalability of the implementations, we measured their speedups on JUMP and HLRB II (Fig. 2). On JUMP all implementations show a good scalability. Even the slowest implementation, (A), obtains a speedup of 20.0 on 32 processors. The best implementations are those with the best locality behavior, i.e., (PipeDb2mt), (PipeDb1mt) and (ppDb1mt). They obtain speedups between 28.9 and 29.6. The speedups on HLRB II are satisfactory for small numbers of processors up to 32, because the data-parallel execution leads to a reduction of the working space of each thread and, hence, to a reduction of the number of cache misses. But for larger numbers of processors all data accessed during one time step fits in the cache, and, as a consequence, the scalability no longer profits from these cache effects. Moreover, the communication of more than 32 processors requires the use of one additional level of the hierarchically organized interconnection network. Therefore, the efficiency of the implementations severely degrades if the number of processors is increased fur-

Table 6. Cache access statistics for 32 threads on JUMP measured using BRUSS2D-MIX with Radau I A(5) and $N = 1000$.

Implementation	Execution Time (s)	L1 Prefetch Requests	L1 Data Misses		L2 Store Misses	Data from L2	Data from L3	Data from Memory
			Load	Store				
(A)	19.83	$1.3 \cdot 10^8$	$9.3 \cdot 10^7$	$7.4 \cdot 10^8$	$4.4 \cdot 10^7$	$1.3 \cdot 10^8$	$1.1 \cdot 10^7$	$1.9 \cdot 10^6$
(D)	14.78	$7.8 \cdot 10^7$	$2.4 \cdot 10^7$	$2.5 \cdot 10^8$	$1.5 \cdot 10^7$	$8.8 \cdot 10^7$	$4.7 \cdot 10^6$	$4.0 \cdot 10^5$
(Dblock)	14.71	$1.1 \cdot 10^8$	$4.5 \cdot 10^7$	$3.7 \cdot 10^8$	$1.5 \cdot 10^7$	$1.2 \cdot 10^8$	$4.4 \cdot 10^6$	$8.2 \cdot 10^5$
(E)	18.96	$9.8 \cdot 10^7$	$4.1 \cdot 10^8$	$8.5 \cdot 10^8$	$3.6 \cdot 10^7$	$1.0 \cdot 10^8$	$7.6 \cdot 10^6$	$2.8 \cdot 10^5$
(PipeDb1m)	16.66	$8.0 \cdot 10^7$	$3.4 \cdot 10^7$	$2.6 \cdot 10^8$	$6.2 \cdot 10^6$	$9.9 \cdot 10^7$	$2.1 \cdot 10^6$	$7.9 \cdot 10^4$
(PipeDb1mt)	13.66	$1.1 \cdot 10^8$	$3.4 \cdot 10^7$	$3.8 \cdot 10^8$	$6.3 \cdot 10^6$	$1.3 \cdot 10^8$	$2.0 \cdot 10^6$	$1.4 \cdot 10^5$
(PipeDb2m)	14.62	$7.9 \cdot 10^7$	$1.3 \cdot 10^8$	$3.5 \cdot 10^8$	$1.5 \cdot 10^7$	$9.4 \cdot 10^7$	$2.0 \cdot 10^6$	$5.1 \cdot 10^4$
(PipeDb2mt)	13.28	$1.1 \cdot 10^8$	$3.6 \cdot 10^7$	$3.7 \cdot 10^8$	$1.5 \cdot 10^7$	$1.3 \cdot 10^8$	$2.0 \cdot 10^6$	$1.7 \cdot 10^5$
(PipeDe2m)	16.95	$3.5 \cdot 10^7$	$4.7 \cdot 10^7$	$1.4 \cdot 10^7$	$7.6 \cdot 10^6$	$6.4 \cdot 10^7$	$9.7 \cdot 10^6$	$1.1 \cdot 10^5$
(ppDb1m)	16.65	$8.2 \cdot 10^7$	$3.1 \cdot 10^7$	$2.6 \cdot 10^8$	$9.8 \cdot 10^6$	$1.1 \cdot 10^8$	$1.8 \cdot 10^6$	$2.7 \cdot 10^4$
(ppDb1mt)	13.33	$1.2 \cdot 10^8$	$4.7 \cdot 10^7$	$4.2 \cdot 10^8$	$1.0 \cdot 10^7$	$1.4 \cdot 10^8$	$1.8 \cdot 10^6$	$6.8 \cdot 10^4$

ther. The highest speedup of 59.5 has been obtained by (ppDb1mt) using 144 processors.

5.4 Correlation between Speedup and Locality

Table 6 shows the counts of selected cache events measured using hardware performance counters on JUMP. These statistics confirm our argument that the scalability of the implementations is strongly related to their locality behavior. Thus, the slowest implementations, (A) and (E), generate the highest number of cache misses. The fastest implementations, (PipeDb1mt), (PipeDb2mt) and (ppDb1mt), which all apply some form of loop tiling, have a high temporal locality but take also profit of spatial locality and the prefetching of cache lines. This leads to a smaller number of cache misses and lower waiting times.

6 Conclusions

In this paper, we have presented several different implementations of iterated Runge-Kutta methods which are suitable for parallel computers with a shared address space. The implementations employ different strategies, i.e., different loop structures and different data structures, to achieve a beneficial utilization of the memory hierarchy. In general, the iteration of the corrector steps must be executed sequentially. But if the problem to be solved has a limited access distance, it can be exploited to reduce the storage space and to reduce the working space of the outer loop by the implementation of a pipelining of the corrector steps. Our runtime experiments performed on two modern supercomputer systems confirm that the locality behavior has a major influence on the scalability of IRK methods. Therefore, data-parallel implementations often obtain a better performance than task-parallel implementations. Loop tiling is one important technique to adjust the size of the working sets to the size of the cache hierarchy. The best scalability, particularly for large numbers of processors up to 144,

has been achieved using an implementation which combines loop tiling with a pipelining of the corrector steps.

Acknowledgments. We thank the NIC Jülich and the LRZ Munich for providing access to their supercomputer systems JUMP and HLRB II.

References

1. Ehrig, R., Nowak, U., Deuflhard, P.: Massively parallel linearly-implicit extrapolation algorithms as a powerful tool in process simulation. In: Parallel Computing: Fundamentals, Applications and New Directions. Elsevier (1998) 517–524
2. Burrage, K.: Parallel and Sequential Methods for Ordinary Differential Equations. Oxford Science Publications (1995)
3. Nørsett, S.P., Simonsen, H.H.: Aspects of parallel Runge-Kutta methods. In: Numerical Methods for Ordinary Differential Equations. Volume 1386 of LNM. (1989) 103–117
4. van der Houwen, P.J., Sommeijer, B.P.: Parallel iteration of high-order Runge-Kutta methods with stepsize control. J. Comput. Appl. Math. **29** (1990) 111–127
5. Jackson, K.R., Nørsett, S.P.: The potential for parallelism in Runge-Kutta methods. Part 1: RK formulas in standard form. SIAM J. Numer. Anal. **32**(1) (February 1995) 49–82
6. Choi, J., Dongarra, J.J., Ostrouchov, L.S., Petitet, A.P., Walker, D.W., Whaley, R.C.: Design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. Sci. Prog. **5** (1996) 173–184
7. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarlin, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, Third Edition. SIAM (1999)
8. Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: 11th ACM Int. Conf. on Supercomputing. (1997)
9. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Par. Comp. **27**(1–2) (2001) 3–35
10. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence Based Approach. Morgan Kaufmann (2002)
11. McKinley, K.S.: A compiler optimization algorithm for shared-memory multiprocessors. IEEE Trans. Par. Dist. Syst. **9**(8) (August 1998) 769–787
12. Irigoin, F., Triolet, R.: Supernode partitioning. In: ACM Symposium on Principles of Programming Languages, San Diego, Calif. (January 1988) 319–329
13. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Trans. Prog. Lang. Syst. (TOPLAS) **21**(4) (1999) 703–746
14. Rauber, T., Rünger, G.: Improving locality for ODE solvers by program transformations. Sci. Prog. **12**(3) (2004) 133–154
15. Korch, M.: Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität. Doctoral thesis, University of Bayreuth, Bayreuth, Germany (December 2006)
16. Korch, M., Rauber, T.: Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining. J. Par. Distr. Comp. **66**(3) (March 2006) 444–468

17. Rauber, T., Rünger, G.: Parallel implementations of iterated Runge-Kutta methods. Int. J. Supercomp. App. **10**(1) (1996) 62–90