Parallel Low-Storage Runge–Kutta Solvers for ODE Systems with Limited Access Distance

Matthias Korch, Thomas Rauber

Bayreuth Reports on Parallel and Distributed Systems

No. 1, July 2010

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Telefon:	+49 921 55 7701
Fax:	+49 921 55 7702
E-Mail:	brpds@ai2.uni-bayreuth.de

UNIVERSITÄT BAYREUTH

Parallel Low-Storage Runge–Kutta Solvers for ODE Systems with Limited Access Distance

Matthias Korch Thomas Rauber

University of Bayreuth,
Applied Computer Science 2

{korch, rauber}@uni-bayreuth.de

Abstract

We consider the solution of initial value problems (IVPs) of large systems of ordinary differential equations (ODEs) for which memory space requirements determine the choice of the integration method. In particular, we discuss the space-efficient sequential and parallel implementation of embedded Runge–Kutta (RK) methods. We focus on the exploitation of a special structure of commonly appearing ODE systems, referred to as 'limited access distance', to improve scalability and memory usage. Such systems may arise, for example, from the semi-discretization of partial differential equations (PDEs).

The storage space required by classical RK methods is directly proportional to the dimension n of the ODE system and the number of stages s of the method. We propose an implementation strategy based on a pipelined processing of the stages of the RK method and show how the memory usage of this computation scheme can be reduced to less than three storage registers by an overlapping of vectors without compromising the choice of method coefficients or the potential for efficient stepsize control. We analyze and compare the scalability of different parallel implementation strategies in detailed runtime experiments on different parallel architectures.

1 Introduction

We consider the parallel solution of initial value problems (IVPs) of ordinary differential equations (ODEs) defined by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) \ , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \ , \tag{1}$$

where $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$. The numerical solution of such problems can be highly computationally intensive, in particular if the dimension n of the ODE system is large. Therefore, parallel solution methods have been proposed by many authors, for example, extrapolation methods [15, 17, 26], waveform relaxation techniques [10, 20, 33, 48], and iterated Runge–Kutta methods [14, 38, 36, 45]. An overview is presented in [10]. In this paper, we focus on low-storage implementations of RK methods suitable for large systems of ODEs where storage space may be the determining factor in the choice of method and implementation variant.

Starting with the initial value $\mathbf{y}_0$, numerical solution methods for ODE IVPs walk through the integration interval $[t_0, t_e]$ using a potentially large number of time steps. The different solution methods are distinguished mainly by the computations performed at each time step. The time step computations of classical explicit RK methods are arranged in s stages. Starting with the approximation value $\boldsymbol{\eta}_\kappa \approx \mathbf{y}(t_\kappa)$, at each stage $l = 1, \dots, s$, an argument vector

$$\mathbf{w}_l = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} \mathbf{v}_i \tag{2}$$

is computed that is used as input for the function evaluation $\mathbf{f}$ to compute a stage vector

$$\mathbf{v}_l = \mathbf{f}(t_\kappa + c_l h_\kappa, \mathbf{w}_l) \ . \tag{3}$$

Finally, a new approximation $\boldsymbol{\eta}_{\kappa+1} \approx \mathbf{y}(t_{\kappa+1})$ is computed by:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{v}_l \ . \tag{4}$$

The coefficients a_{li}, c_i, and b_l define the particular RK method. If the RK method provides an embedded solution, a second approximation $\hat{\boldsymbol{\eta}}_{\kappa+1} \approx \mathbf{y}(t_{\kappa+1})$ of different order,

$$\hat{\boldsymbol{\eta}}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{l=1}^{s} \hat{b}_l \mathbf{v}_l \ , \tag{5}$$

can be computed using different weights $\hat{b}_l$. Based on the difference $\mathbf{e} = \boldsymbol{\eta}_{\kappa+1} - \hat{\boldsymbol{\eta}}_{\kappa+1}$, this second approximation allows an inexpensive estimation of the local error and the selection of an appropriate stepsize without additional function evaluations. If the local error lies within a user-defined tolerance, the time step can be accepted; otherwise it has to be repeated with a smaller stepsize. Hence, RK methods with embedded solutions (referred to as *embedded RK methods*) require less computational effort to control the stepsize than other approaches such as, e.g., Richardson extrapolation [21].

In the general case, where no assumptions about the method coefficients or the access pattern of the function evaluation can be made, an implementation of the classical RK scheme has to compute the stages one after the other. Since each stage depends on all previously computed stages, classical RK schemes need to hold at least $s+1$ vectors of size n (so called *registers*) in memory to store $\boldsymbol{\eta}_\kappa$, $\mathbf{w}_2, \ldots, \mathbf{w}_s$, and $\boldsymbol{\eta}_{\kappa+1}$, where n is the dimension of the ODE system. The argument vector $\mathbf{w}_1$ is identical to $\boldsymbol{\eta}_\kappa$ and does not require the use of an extra register. One additional register ($\hat{\boldsymbol{\eta}}_{\kappa+1}$ or the error estimate $\mathbf{e}$) is required for the implementation of a stepsize controller which can reject and repeat steps.

In regard to storage space, on modern computer systems equipped with several GB of memory, the classical approach can be applied to ODE systems with tens or even hundreds of millions of components, even if a method of high order with a large number of stages is required. Some ODE problems, however, require an even larger number of components. These are, in particular, ODE systems derived from systems of partial differential equations (PDEs) by a spatial discretization using the method of lines. For example, a spatial resolution of 1 ‰ leads to a system with at least $n_v \cdot 10^6$ components for a 2D PDE system and $n_v \cdot 10^9$ components for a 3D PDE system, where n_v is the number of dependent variables in the PDE system. Enhancing the spatial resolution by a factor of 10 increases the number of ODE components by a factor of 10^2 for the 2D PDE system and by a factor of 10^3 for the 3D PDE system.

Parallel implementations of embedded RK methods which make no assumptions about the method coefficients or the coupling of the ODE system have to exchange the computed parts of the current argument vector between all participating processors at every stage. The scalability of such general implementations is therefore often not satisfactory. To overcome the limitations of general implementations, two possible approaches take advantage of special properties of either the embedded RK method [23] or the ODE system to be solved [29, 31]. In the following, we follow the second approach of taking advantage of special properties of the ODE system and investigate the scalability of data-parallel implementations of embedded RK methods. All implementation variants investigated in this paper have been written in C and use POSIX Threads (Pthreads) [11] or MPI [42] for synchronization.

In the following, after a discussion of related approaches to reduce the storage space of RK methods in Section 2, Section 3 discusses optimizations of the communication costs and the locality of memory references based on a pipelined processing of the stages. This pipelining approach is applicable to ODE systems with a special access pattern as is typical for ODE systems derived by the method of lines. In Section 4, we present a new approach to reduce the storage space that builds on the pipelining approach combined with an overlapped memory layout of the data structures. It reduces the storage space for a fixed-stepsize implementation to a single register plus $\Theta\left(\frac{1}{2}s^2 \cdot d(\mathbf{f})\right)$ vector elements, where $d(\mathbf{f})$ is the access distance of the right-hand-side function $\mathbf{f}$ (see Section 3.2), while preserving all degrees of freedom in the choice of RK coefficients. Stepsize control based on an embedded solution can be realized with only one additional register. Further, it is possible to develop a parallel implementation of the low-storage pipelining scheme that can efficiently utilize parallel computing resources and reduce the memory requirements of a single computing node. An experimental evaluation of runtime and scalability of the new implementation on five different modern parallel computer systems is presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

On modern computer systems, which are equipped with a complex memory hierarchy, the spatial and temporal locality of memory references can have a large influence on the execution time of a program. Therefore, numerous optimizations to improve the locality of memory references have been developed by several research teams for many applications. Particular interest has been given to the field of numerical linear algebra, including factorization methods such as LU, QR and Cholesky [13] and iterative methods like 2D Jacobi [19] and multi-grid methods [32]. PHiPAC (Portable High Performance ANSI C) [7] and ATLAS (Automatically Tuned Linear Algebra Software) [47] aim at efficient implementations of BLAS (Basic Linear Algebra Subprograms) [8] routines by automatic code generation. Modern optimizing compilers try to achieve an efficient exploitation of the memory hierarchy by reordering the instructions of the source program [2, 49]. A lot of the research in this area concentrates on computationally intensive loops [24, 25, 35], and loop tiling [4, 44] is considered to be one of the most successful techniques.

In this paper, we consider the sequential and parallel implementation of RK methods with embedded solutions. Embedded RK methods are among the most popular one-step methods for the numerical integration of ODE IVPs because they combine low execution times with good numerical properties [21]. Examples of popular embedded RK methods are the methods of Fehlberg [18] and Dormand & Prince [16, 37]. Variants of these methods are used in many software packages. Examples are the subroutine DVERK [22], which is part of IMSL, and the RK solver collection RKSUITE [9]. However, classical RK schemes (with or without embedded solutions) have the disadvantage that the storage space required grows linearly with the dimension of the ODE system, n, and the number of stages, s.

A first approach to get along with low storage space is the use of RK methods with a small number of stages. This, however, means abandoning desirable properties of the method, such as a high order. Therefore, several authors, e.g., [5, 6, 12, 28, 40], propose special explicit RK methods that can be implemented with low storage space (e.g., 2 or 3 registers) even though they have a larger number of stages and a higher order. This is possible by the construction of special coefficient sets such that, at each stage, only the data stored in the available registers are accessed, and the values computed at this stage fit in the available registers. These approaches, however, retain some deficiencies:

1. The coefficient sets must conform to additional constraints. Hence, fewer degrees of freedom are available in the construction of the coefficients. As a result, the methods proposed have weaker numerical properties than classical RK schemes with the same number of stages. If stronger numerical properties are required, one must resort to classical RK schemes with higher memory usage.

2. Using a higher number of stages to reach a specific order leads to higher computational costs and, as a consequence, to a higher runtime.

3. Many of the low-storage RK methods proposed do not provide embedded solutions. Therefore, these methods are suitable only for fixed-stepsize integration, unless additional computations and storage space are invested into stepsize control (e.g., using Richardson extrapolation).

4. Low-storage RK methods with embedded solutions have been proposed in [27]. They, however, impose additional constraints on the method coefficients, thus further reducing the degrees of freedom in the construction of the coefficients.

5. Stepsize control requires additional registers: One additional register is needed to estimate the local error ($\hat{\boldsymbol{\eta}}_{\kappa+1}$ or, alternatively, $\mathbf{e} = \hat{\boldsymbol{\eta}}_{\kappa+1} - \boldsymbol{\eta}_{\kappa+1}$). This register may be omitted if the embedded solution $\hat{\boldsymbol{\eta}}_{\kappa+1}$ is computed at stage $s-1$, so that it can be compared with $\boldsymbol{\eta}_{\kappa+1}$, which is computed at stage s, without explicitly storing it [27]. In order to be able to repeat the time step in case that the error is too large, a second additional register is needed that saves $\boldsymbol{\eta}_{\kappa}$.

6. Some of the 2-register low-storage RK schemes proposed rely on a specific structure of the coupling between the equations in the ODE system such that, for example, the argument vector of a function evaluation can be overwritten by the result of the function evaluation. If right-hand-side functions with arbitrary access patterns are to be supported, an additional register must be used to store the result of the function evaluation temporarily [28].

It is, therefore, desirable to find new methods or algorithms which overcome these deficiencies. In this paper, we reconsider a pipelining approach suggested, initially, to improve locality and scalability of embedded RK implementations in [30] from this perspective. As we will show in the following sections, for IVPs with a special dependence pattern, the storage space can be reduced by an overlapped memory layout of the data structures such that only one enlarged register of dimension n can be (re-)used for all stage computations.

A similar dependence pattern is exploited in [3] to perform in-place stencil computations as they are typical for finite element, finite volume and finite difference methods for the solution of PDEs. In contrast to this paper, the computations performed at one time step do not consist of several stages, and a constant stepsize is assumed. Our approach allows for time-step computations based on high-order (embedded) RK methods with a high number of stages and adaptive stepsize control.

3 Exploiting Limited Access Distance to Reduce Working Space and Communication Overhead

Many sparse ODEs, in particular many discretized PDEs derived by the method of lines, are described by a right-hand-side function $\mathbf{f} = (f_1, \ldots, f_n)$ where the components of the argument vector accessed by each component function f_j lie within a bounded index range near j. In the following, we review the pipelining approach proposed in [29, 30, 31], which exploits this property of the ODE system and supports arbitrary RK coefficients.

3.1 Motivation: Disadvantages of General Implementations

General embedded RK implementations suitable for ODE systems of arbitrary structure have to take into account that the evaluation of each component function $f_j(t, \mathbf{w}_l)$, $j \in \{1, \ldots, n\}$, may access all components of the argument vector $\mathbf{w}_l$. Therefore, the stages have to be computed sequentially, i.e., within one time step the outermost loop iterates over the stages $l = 1, \ldots, s$. The two inner loops iterate over the dimension of the ODE system ($j = 1, \ldots, n$) and over the preceding ($i = 1, \ldots, l-1$) or the succeeding ($i = l+1, \ldots, s$) stages and can be interchanged. The influence of several different loop structures, which have been named (A)–(E), on the locality behavior has been investigated in [29, 31, 39].

Since the outermost loop iterates over the stages and one of the inner loops iterates over the system dimension, the resulting working space of the outermost loop of the stage computation is $\Theta(s \cdot n)$. Thus, assuming a significantly large value of n, the working space of the outermost loop often does not fit into the lower cache levels. Moreover, if n is very large, the size of this working space may even exceed the available amount of memory, in particular if an RK method with a high number of stages is used.

To exploit parallelism, the computation of the components of the argument vectors (2) and the evaluations of the component functions $f_j(t, \mathbf{w}_l)$ in (3) can be distributed equally among the processors, i.e., the loop over the dimension of the ODE system is executed in parallel. At every stage, global synchronization is required to make the computed parts of the current argument vector available to all participating processors. In a shared-address-space implementation, barrier operations can be used after the computation of the argument vector to synchronize the processors before they access the argument vector to evaluate the right-hand-side function. In a distributed-address-space implementation, the computed parts of the argument vector can be exchanged by a multibroadcast operation (`MPI_Allgather()`). Usually, the execution time of multibroadcast operations increases severely with the number of participating processors. Therefore, general distributed-address-space embedded RK implementations do not scale well with the number of processors (cf. experimental evaluation in Section 5).

All specialized implementation variants considered in this papers have been derived from a general implementation variant named (D). (D) is an optimized implementation that aims at a high temporal locality by using the stage vector components $v_{l,j} = f_j(t, \mathbf{w}_l)$ as soon after their computation as possible. This is achieved by updating the corresponding components of the succeeding argument vectors $w_{l+1,j}, \ldots, w_{s,j}$ in the innermost loop. A more detailed description of this implementation can be found in [29, 31].

3.2 Access Distance and Resulting Block Dependence Structure

Many ODEs are sparse, i.e., the evaluation of a component function $f_j(t, \mathbf{w})$ uses only a small number of components of $\mathbf{w}$, and we can exploit this property to devise parallel algorithms with higher scalability. In many cases, the components accessed are located nearby the index j. To measure this property of a function $\mathbf{f}$, we use the *access distance* $d(\mathbf{f})$, which is the smallest value b, such that all component functions $f_j(t, \mathbf{w})$ access only the subset $\{w_{j-b}, \ldots, w_{j+b}\}$ of the components of their argument vector $\mathbf{w}$. We say the access distance of $\mathbf{f}$ is *limited* if $d(\mathbf{f}) \ll n$.

A limited access distance generally leads to a band-structured Jacobian of $\mathbf{f}$ with bandwidth $d(\mathbf{f})$. Usually, one can choose between different orderings of the equations which may influence the bandwidth/access distance. Several heuristic and exact optimization algorithms exist, e.g., [34, 46], which aim at a minimization of the bandwidth of sparse symmetric matrices and thus can be used for many ODE systems to find an ordering of the equations which provides a limited access distance.

Given a function $\mathbf{f}$ with limited access distance $d(\mathbf{f})$, we can subdivide all n-vectors into $n_B = \lceil n/B \rceil$ blocks of size B, where $d(\mathbf{f}) \leq B \ll n$. Then, the function evaluation of a block $J \in \{1, \ldots, n_B\}$ defined by

$$\mathbf{f}_J(t, \mathbf{w}) = (f_{(J-1)B+1}(t, \mathbf{w}), f_{(J-1)B+2}(t, \mathbf{w}), \ldots, \\ f_{(J-1)B+\min\{B, n-(J-1)B\}}(t, \mathbf{w}))$$

uses only components of the blocks $J-1$, J, and $J+1$ of $\mathbf{w}$ if these blocks exist, i.e., if $1 < J < n_B$.

3.3 Replacing Global by Local Communication

A major drawback of general parallel implementations suitable for ODE systems with arbitrary access structure is the necessity for global communication to make the current argument vector available to all participating processors. However, if the right-hand-side function has a limited access distance, it is possible to avoid global communication in the stage computation phase.

For that purpose, we consider a blockwise distribution of the blocks resulting from the subdivision of the n-vectors described in the previous section. Using a loop structure which is similar to that of the general implementation (D), we can now reduce the communication costs by exchanging only required blocks between neighboring processors. Since the function evaluation of a block J, $\mathbf{f}_J(t, \mathbf{w})$, uses only components of the blocks $J-1$, J, and $J+1$ of $\mathbf{w}$, each processor needs access to only one block stored in the preceding processor and to one block stored in the succeeding processor. Hence, in a distributed-address-space implementation the multibroadcast operation can be replaced by two non-blocking single-transfer operations (`MPI_Isend()`/`MPI_Irecv()`), which can be overlapped with the computation of $n_B/p - 2$ blocks. As a result, the major communication costs no longer grow with the number of processors, and the influence of the network bandwidth on scalability is reduced significantly. In a shared-address-space implementation, the barrier operations can be replaced by locks, which can be organized such that usually no waiting times occur. The resulting implementation variant is named (Dbc). Figure 1 illustrates the data distribution, the computation order of the blocks, and the blocks which have to be exchanged between neighboring processors. A more detailed description is given in [29, 31].

3.4 Reduction of the Working Space by Block-Based Pipelining

In addition to the reduction of the communication costs, right-hand-side functions with limited access distance provide the possibility to improve the locality of memory references by a reduction of the working space of the outermost loop.

The special dependence structure of such right-hand-side functions leads to a decoupling of the stages that enables the computation of the blocks of the argument vectors $\mathbf{w}_2, \ldots, \mathbf{w}_s$, the vector $\Delta\boldsymbol{\eta} = \boldsymbol{\eta}_{\kappa+1} - \boldsymbol{\eta}_\kappa$ and the error estimate $\mathbf{e} = \hat{\boldsymbol{\eta}}_{\kappa+1} - \boldsymbol{\eta}_{\kappa+1}$ as illustrated in Fig. 2 (a). The computation starts with the first and the second block of $\mathbf{w}_2$, which only requires components of $\mathbf{w}_1 = \boldsymbol{\eta}_\kappa$. Then the first block of $\mathbf{w}_3$ can be computed, since it only uses components of the first two blocks of $\mathbf{w}_2$. In the next step, the third block of $\mathbf{w}_2$ is computed, which enables the computation of the second block of $\mathbf{w}_3$, which again enables the computation of the first block of $\mathbf{w}_4$. This is continued until the computation of the first two blocks of $\mathbf{w}_s$ has been completed and the first block of $\Delta\boldsymbol{\eta}$ and the first block of $\mathbf{e}$ have been computed. Then the next block of $\Delta\boldsymbol{\eta}$ and the next block of $\mathbf{e}$ can be determined by computing only one additional block of $\mathbf{w}_2, \ldots, \mathbf{w}_s$. This computation is repeated until the last block of $\Delta\boldsymbol{\eta}$ and the last block of $\mathbf{e}$ have been computed.

An important advantage of the pipelining approach is that only those blocks of the argument vectors are kept in the cache which are needed for further computations of the current step. One step of the pipelining computation scheme computes s argument blocks and one block of $\Delta\boldsymbol{\eta}$ and $\mathbf{e}$. Since the function evaluation of one block J accesses the blocks $J-1$, J, and $J+1$ of the cor-

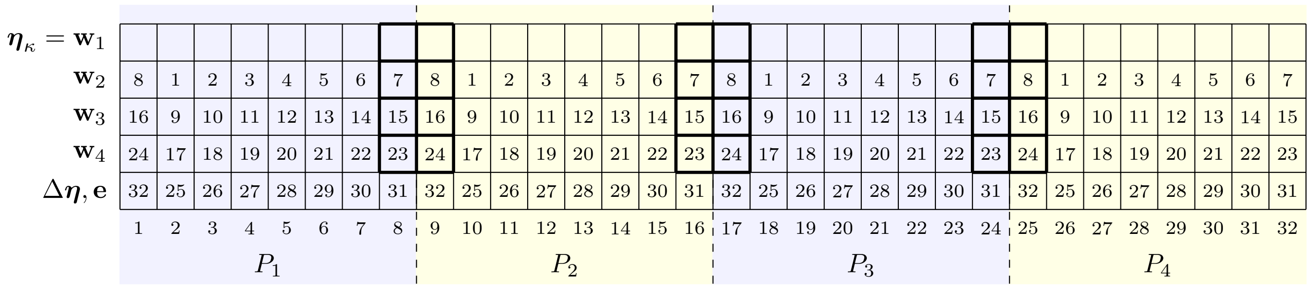

Fig. 1: Illustration of the data-parallel implementation variant (Dbc), which exploits a limited access distance to replace global by local communication. The numbers determine the computation order of the blocks. Blocks accessed during initialization or finalization phases in which communication is required are displayed as filled boxes, and blocks required by neighboring processors are emphasized by thick borders.

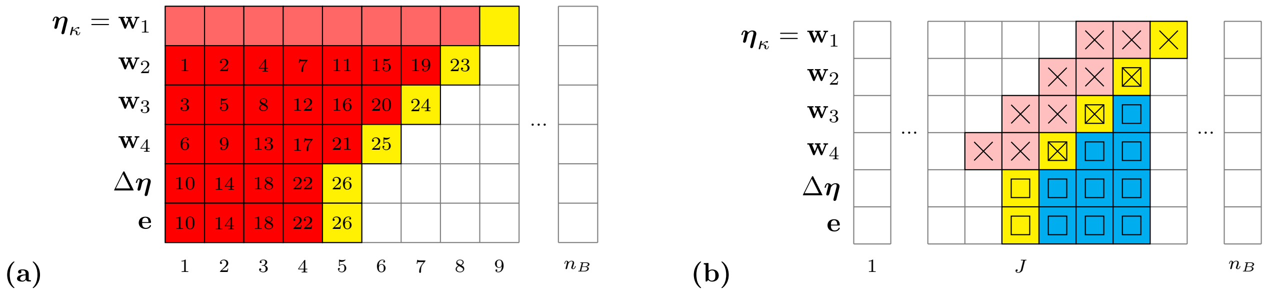

Fig. 2: **(a)** Illustration of the computation order of the pipelining computation scheme. After the blocks with index 4 of $\Delta\boldsymbol{\eta}$ and $\mathbf{e}$ have been computed, the next pipelining step computes the blocks with index 5 of $\Delta\boldsymbol{\eta}$ and $\mathbf{e}$. **(b)** Illustration of the working space of one pipelining step. Blocks required for function evaluations are marked by crosses. Blocks updated using results of function evaluations are marked by squares.

responding argument vector, $3s$ argument vector blocks must be accessed to compute one block of $\Delta\boldsymbol{\eta}$ and $\mathbf{e}$. Additionally,

$$\sum_{i=3}^{s}(i-2) = \frac{1}{2}(s-1)(s-2) = \frac{1}{2}s^2 - \frac{3}{2}s + 1$$

blocks of argument vectors, s blocks of $\Delta\boldsymbol{\eta}$ and s blocks of $\mathbf{e}$ must be updated. Altogether, the working space of one pipelining step consists of

$$\frac{1}{2}s^2 + \frac{7}{2}s + 1 = \Theta\left(\frac{1}{2}s^2\right) \tag{6}$$

blocks of size B, see Fig. 2 (b). Since $B \ll n$, this is usually only a small part of the working space of $\Theta(s \cdot n)$ adherent to general implementations suitable for arbitrary right-hand-side functions, which iterate over all s argument vectors in the outermost loop.

Combining the pipelining scheme with the optimized communication pattern described in Section 3.3 leads to highly scalable and efficient parallel implementations. In this paper, we consider four parallel implementation variants which exploit a limited access distance to reduce communication costs and the working space but still require $\Theta(s \cdot n)$ storage.

The distributed-address-space implementations try to overlap communication with computations by using non-blocking single-transfer operations (`MPI_Isend()` and `MPI_Irecv()`). The shared-address-space implementations use locks for synchronization and do not overlap communication with computations.

The first two variants of the pipelining computation scheme illustrated in Figs. 3 (a) and 3 (b), to which we refer as (PipeD) and (PipeD2), cf. [29, 31], have in common that they start by computing all inner blocks in pipelining computation order (towards increasing indices) which do not depend on blocks of neighboring processors. To ensure that each processor computes at least one block of $\mathbf{w}_s$ in this phase, at least $2s$ blocks are assigned to each processor.

In the second phase, the processors finalize the pipelines. (PipeD) and (PipeD2) differ in the finalization strategies used to compute the remaining blocks. In implementation (PipeD), 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. The resulting parallel computation order resembles a zipper. 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 process its last block of $\mathbf{w}_i$, and vice versa. Thus, the transfer of the blocks can be overlapped with the computation of one diagonal. But the fraction of the transfer times that can be overlapped by computations is not as large as in (Dbc), because the length of the diagonals decreases from $s-1$ blocks to 1 block as the finalization phase proceeds.

By contrast, (PipeD2) minimizes the number of communication operations and thus the communication costs arising from startup times by letting each proces-

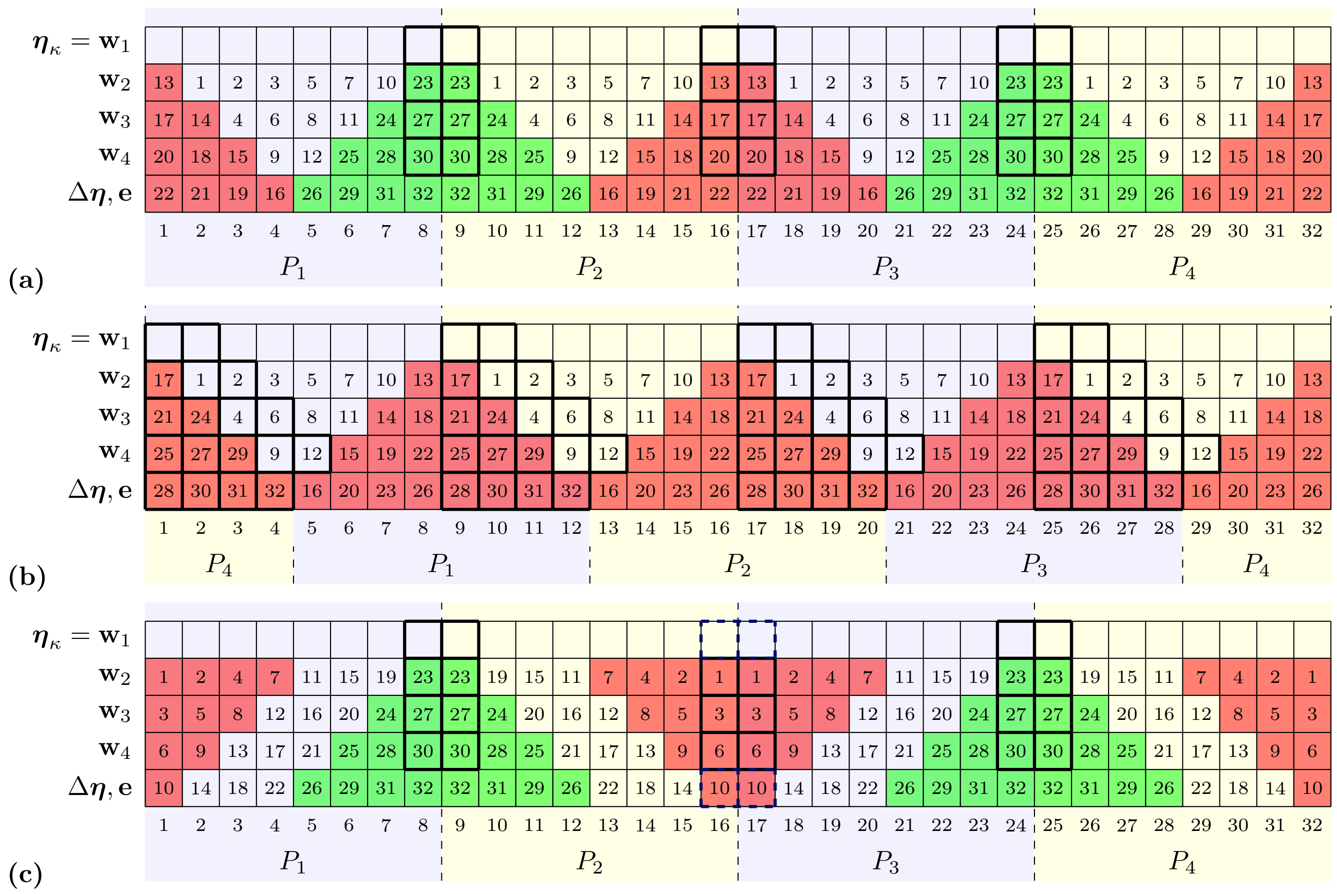

Fig. 3: Illustration of data-parallel implementation variants of embedded RK methods that exploit a limited access distance. The numbers determine the computation order of the blocks. Blocks accessed during initialization or finalization phases in which communication is required are displayed as filled boxes, and blocks required by neighboring processors are emphasized by thick borders. **(a)** (PipeD), **(b)** (PipeD2), **(c)** (PipeD4) and (PipeD5).

sor finalize the higher end of its own pipeline and the lower end of the pipeline of its cyclic successor. This finalization strategy requires only one non-blocking data transfer from a processor to its cyclic predecessor, but the complete working space of a pipelining step, which consists of $2s + \sum_{i=2}^{s} (i-1) = \frac{1}{2}\left(s^2 + 3s\right)$ blocks of argument vectors, s blocks of $\Delta\boldsymbol{\eta}$ and s blocks of $\mathbf{e}$, has to be sent over the network. The transfer can be started when the pipeline has been initialized, but the data are not needed before the finalization starts. Hence, the transfer can be conducted in parallel with the diagonal sweep across the argument vectors, which computes $(n_B/p - 2s) \cdot s$ blocks. During the finalization of the pipelines, the processors compute parts of $\Delta\boldsymbol{\eta}$ that are required by their successors to compute $\boldsymbol{\eta}_{\kappa+1}$ and to start the next time step. Therefore, another communication operation is needed to send s blocks of $\Delta\boldsymbol{\eta}$ to the successor. (PipeD2) exploits the fact that this is only necessary if the step is accepted by the error control, because otherwise the step is repeated using $\boldsymbol{\eta}_\kappa$ and, therefore, $\boldsymbol{\eta}_{\kappa+1}$ is not needed.

In addition to (PipeD) and (PipeD2), we consider two new parallel variants of the pipelining computation scheme, to which we will refer as (PipeD4) and (PipeD5). Both are covered by the illustration in Fig. 3 (c). In contrast to (PipeD) and (PipeD2), (PipeD4) and (PipeD5) let processors with odd and even index iterate over the dimension of the ODE system in opposite direction. Both work in three phases: initialization phase, sweep phase, and finalization phase.

The differences between (PipeD4) and (PipeD5) concern the overlapping of communication with computation and are not relevant to the shared-address-space implementations, which use blocking communication. Therefore, only one shared-address-space implementation called (PipeD4) that follows the computation scheme illustrated in Fig. 3 (c) has been implemented.

In the distributed-address-space implementations (PipeD4) and (PipeD5), non-blocking communication of single blocks similar to (PipeD) is performed during the initialization phase and during the finalization phase.

One disadvantage of the new computation order is that the transfer of the blocks of $\boldsymbol{\eta}_\kappa = \mathbf{w}_1$ which are required to start the initialization phase cannot be overlapped by computations. Therefore, while (PipeD4) does not overlap this transfer time, we devised the variant (PipeD5) which exchanges blocks of $\Delta\boldsymbol{\eta}$ instead. The transfer of the blocks of $\Delta\boldsymbol{\eta}$ can be overlapped with the computations performed during the sweep phase and the finalization phase. The required blocks of $\boldsymbol{\eta}_\kappa$ are then computed by adding the values of the blocks of $\Delta\boldsymbol{\eta}$ to the previous values of the blocks of $\boldsymbol{\eta}_\kappa$. This computation is redundant but may save communication costs. Which implementation is faster depends on whether it takes less time to transfer one block of size B through the network or to perform B additions and the corresponding local memory accesses.

4 Low-Storage Implementation of the Pipelining Scheme

Classical implementations of embedded RK methods, which make no assumptions about the method coefficients or the access pattern of the function evaluation, need to hold at least $s+2$ vectors of size n (so called *registers*) in memory to store $\boldsymbol{\eta}_\kappa$, $\mathbf{w}_2, \ldots, \mathbf{w}_s$, $\boldsymbol{\eta}_{\kappa+1}$, and $\hat{\boldsymbol{\eta}}_{\kappa+1}$. If the dimension n of the ODE system is large, the amount of storage space required to store all $s+2$ registers may be larger than the main memory available on the target platform.

In this section, we show how the pipelining approach can be implemented with stepsize control using only '2+' registers, i.e., 2 registers plus some small extra space that vanishes asymptotically if the ODE system is very large. Without stepsize control the storage space required can even be reduced to '1+' registers.

4.1 Motivation and Sequential Implementation

Reconsidering the pipelining computation scheme in terms of storage space, we observe that the pipelining computation scheme, by performing a diagonal sweep across the argument vectors, delivers the blocks of elements of $\Delta\boldsymbol{\eta}$ and $\mathbf{e}$ one after the other as the sweep proceeds along the dimension of the ODE system. The computation of the next blocks of these vectors depends only on those argument vector blocks that belong to the working space of the corresponding pipelining step. Argument vector blocks behind the sweep line that is determined by the working space of the current pipelining step are no longer required for computations in the current time step. Argument vector blocks ahead of the sweep line do not yet contain data computed in the current time step and thus do not contribute to the computations performed in the current pipelining step. Hence, a significant reduction of the storage space can be achieved if only those argument vector blocks are kept in memory that are part of the working space of the current pipelining step.

A second opportunity to save storage space lies in the computation of the error estimate $\mathbf{e}$. The error estimate $\mathbf{e}$ is used in the decision whether to accept the current time step or to reject it and repeat it with a smaller stepsize. Most stepsize control algorithms make this decision based on a scalar value that is an aggregate of the elements of $\mathbf{e}$, e.g., a vector norm. A common choice for the aggregate (ϵ) is the maximum norm of $\mathbf{e}/\boldsymbol{\sigma}$ divided by some user defined tolerance TOL:

$$\epsilon = \frac{1}{\mathrm{TOL}} \left\| \frac{\mathbf{e}}{\boldsymbol{\sigma}} \right\|_\infty = \frac{1}{\mathrm{TOL}} \max_{j=1,\ldots,n} \left| \frac{e_j}{\sigma_j} \right| \quad ,$$

where $\boldsymbol{\sigma}$ is a vector of scaling factors, e.g., $\sigma_j = \max\{|\eta_{\kappa,j}|, |\eta_{\kappa+1,j}|\}$ for $j = 1, \ldots, n$. But the computation of aggregates of this kind does not require all elements of $\mathbf{e}$ to be stored in memory. Rather, a sweep approach can be applied which updates a scalar variable as the elements of $\mathbf{e}$ are computed one after the other:[1]

1.) $\epsilon := 0$,

2.) $\epsilon := \max\left\{\epsilon, \left|\frac{e_j}{\sigma_j}\right|\right\}$ for $j := 1, \ldots, n$,

3.) $\epsilon := \frac{\epsilon}{\mathrm{TOL}}$.

This sweep approach to compute ϵ can easily be integrated into the pipelining scheme, which delivers one new block of $\mathbf{e}$ after each pipelining step.

All in all, it is sufficient to store the two n-vectors $\boldsymbol{\eta}_\kappa$ and $\Delta\boldsymbol{\eta}$, the $\frac{1}{2}s^2 + \frac{3}{2}s - 2$ blocks of $\mathbf{w}_2, \ldots, \mathbf{w}_s$ belonging to the working space of the current pipelining step and s blocks of the error estimate $\mathbf{e}$. This leads to a total amount of

$$2n + \left(\frac{1}{2}s^2 + \frac{5}{2}s - 2\right) B = 2n + \Theta\left(\frac{1}{2}s^2 B\right) \tag{7}$$

vector elements that have to be stored in memory. If no stepsize control is used, $\boldsymbol{\eta}_\kappa$ can be overwritten by $\boldsymbol{\eta}_{\kappa+1}$ because the steps are always accepted and $\boldsymbol{\eta}_\kappa$ is not needed to be able to repeat the time step. Thus, the memory space required even reduces to

$$n + \left(\frac{1}{2}s^2 + \frac{3}{2}s - 2\right) B = n + \Theta\left(\frac{1}{2}s^2 B\right) \tag{8}$$

vector elements, since no elements of $\Delta\boldsymbol{\eta}$ and $\mathbf{e}$ have to be stored.

One possibility to design an implementation of a low-storage pipelining scheme which supports stepsize control by an embedded solution is the overlapping of $\Delta\boldsymbol{\eta}$ with the vectors $\mathbf{e}$ and $\mathbf{w}_2, \ldots, \mathbf{w}_s$ as shown in Fig. 4 (a). The sequential implementation requires two registers, one of size n which stores the current approximation vector $\boldsymbol{\eta}_\kappa$, and one of size $n + \left(\frac{1}{2}s^2 + \frac{5}{2}s - 2\right) B$ which stores $\Delta\boldsymbol{\eta}$ and the elements of $\mathbf{e}$ and $\mathbf{w}_2, \ldots, \mathbf{w}_s$ which belong to the working space of the current pipelining step using the displacements shown in Table 1. Using this overlapping, the pipelining scheme can be executed without further changes as described in Section 3.4. As a result, the working space of the pipelining steps is embedded into the enlarged register holding $\Delta\boldsymbol{\eta}$ as a compact, consecutive window such that the computation of the next block of $\Delta\boldsymbol{\eta}$ moves this window one block forward.

4.2 Parallel Implementation

A parallel low-storage implementation of the pipelining scheme can easily be obtained with only a few adaptations. Because in the low-storage implementation the working space of the pipelining steps is embedded into

[1] A similar sweep approach is possible for other L^p norms $||x||_p = \left(\sum_{j=1}^n |x_j|^p\right)^{\frac{1}{p}}$.

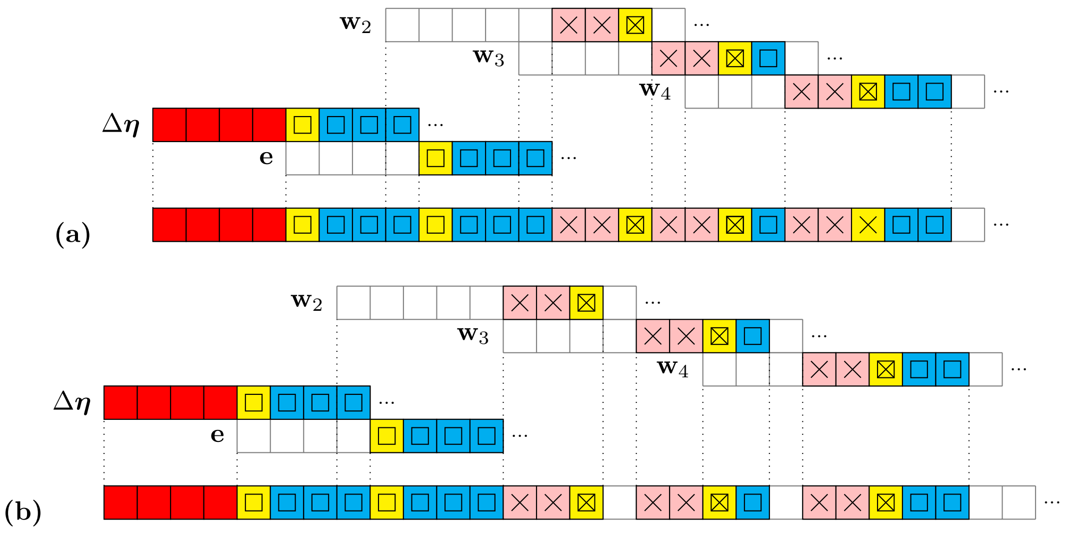

Fig. 4: Illustration of the overlapping of the vectors $\Delta\boldsymbol{\eta}$, $\mathbf{e}$ and $\mathbf{w}_2, \ldots, \mathbf{w}_s$ in the low-storage implementation. This overlapping allows us to embed $\Delta\boldsymbol{\eta}$ and the working space of the pipelining steps into a single vector of size $n + \Theta\left(\frac{1}{2}s^2B\right)$. **(a)** Sequential implementation and parallel implementation without support for overlapping communication with computations. **(b)** Parallel implementation with support for overlapping communication with computations.

Vector:	$\Delta\boldsymbol{\eta}$	$\mathbf{e}$	$\mathbf{w}_2$	$\ldots$	$\mathbf{w}_i$	$\ldots$	$\mathbf{w}_s$
Offset (blocking/sequential):	0	s	$s+3$	$\ldots$	$s+\frac{1}{2}i^2+\frac{3}{2}i-2$	$\ldots$	$\frac{1}{2}s^2+\frac{5}{2}s-2$
Offset (non-blocking):	0	s	$s+3$	$\ldots$	$s+\frac{1}{2}i^2+\frac{5}{2}i-4$	$\ldots$	$\frac{1}{2}s^2+\frac{7}{2}s-4$

Tab. 1: Offsets (in number of blocks) of the overlapping of $\Delta\boldsymbol{\eta}$ with $\mathbf{e}$ and $\mathbf{w}_2, \ldots, \mathbf{w}_s$

the register holding $\Delta\boldsymbol{\eta}$ and each processor must keep the data of its working space in memory, it is not possible to realize a shared-address-space implementation that uses only one global copy of this register to which all participating processors have shared access. Instead, each processor allocates a chunk of memory of size $\lceil \frac{n_B}{p} \rceil + \left(\frac{1}{2}s^2 + \frac{5}{2}s - 1\right) B$ locally where it can hold the working space of its partition of the outermost loop and ghost cells copied from neighboring processors. The local allocation can be advantageous on shared-memory systems with non-uniform memory access times (NUMA systems). It can, however, be a disadvantage on shared-memory systems with uniform memory access times (UMA systems), because blocks which are required by neighboring processors have to be copied between the local data structures of the threads and there is no gain in faster access times to the elements of the local copy.

Since the embedded working space has to be moved forward block by block, the computation order illustrated in Fig. 3 (a) can not be applied. Moreover, the computation order illustrated in Fig. 3 (b) can not be implemented efficiently because the data to be transferred are overwritten in subsequent pipelining steps and, therefore, the data transfer can not be overlapped with computations. However, parallel implementations with reduced storage space can be realized using the computation order illustrated in Fig. 3 (c), i.e., using the same computation order as (PipeD4) or (PipeD5). In the experimental evaluation in Section 5, we consider a low-storage implementation derived from (PipeD4) referred to as (PipeD4ls).

Parallel implementations based on blocking communication, such as our shared-address-space implementations, can be realized using the same vector displacements as the sequential implementation (Fig. 4 (a)). But in order to support the overlapping of communication with computations, the overlapping memory layout of the argument vectors needs to be splayed by one block as illustrated in Fig. 4 (b). This slightly enlarges the register which stores $\Delta\boldsymbol{\eta}$ and the elements of $\mathbf{e}$ and $\mathbf{w}_2, \ldots, \mathbf{w}_s$ which belong to the working space of the current pipelining step to the size $n + \left(\frac{1}{2}s^2 + \frac{7}{2}s - 3\right) B$. The resulting new offsets are included in Table 1.

4.3 Comparison of Data Set Sizes of Semi-discretized PDEs

The spatial discretization of a d-dimensional PDE system by the method of lines [41] uses a d-dimensional spatial grid. In the following, we assume that the same number of grid points is used in each dimension, and refer to it as N. The ODE system which results from this semi-discretization consists of $n_v \cdot N^d$ equations, where n_v is the number of dependent variables in the PDE system. The spatial derivatives at the grid points are approximated using difference operators which use grid points in a bounded environment of the grid point considered. Hence, one spatial dimension can be selected for a subdivision of the ODE system into blocks such that each block represents an N^{d-1}-dimensional slice of the grid and the ODE system has a limited access distance $d(\mathbf{f}) = n_v \cdot r \cdot N^{d-1}$, where r is the maximum distance in the selected dimension between the grid point to which the discretization operator is applied and the grid points used by the discretization operator.

Since the working space of the pipelining scheme requires a storage space of only $\Theta\left(\frac{1}{2}s^2 d(\mathbf{f})\right) = \Theta\left(\frac{1}{2}s^2 \cdot n_v r N^{d-1}\right)$ while the system size n grows with $\Theta\left(N^d\right)$, the additional storage space needed to store the working space becomes less significant when N increases. As an example for the case of sequential execution, Table 2 shows a comparison of data set sizes for an ODE of size $n = 2N^2$ with access distance $d(\mathbf{f}) = 2N$, as it may result from a 2D PDE with two dependent variables discretized using a five-point stencil (e.g., BRUSS2D [10], see also Section 5). The function $S(x)$ used in this table represents the space needed to store x vector elements using double precision (8 bytes per vector element). P_s^B is the number of vector elements in the working space of the pipelining scheme that have to be stored in addition to $\boldsymbol{\eta}_\kappa$ (and $\Delta\boldsymbol{\eta}$) if an s-stage method and blocksize B is used, i.e.,

$$P_s^B = \left(\frac{1}{2}s^2 + \frac{5}{2}s - 2\right) B \ . \tag{9}$$

Column I shows the size of the working space $S(P_s^B)$ in the case that a 7-stage method is used. We chose $s = 7$ because the popular embedded RK method DOPRI 5(4) that we used in the runtime experiments presented in Section 5 has 7 stages. Column II shows the size of a single register holding n vector elements. Columns III and V represent the total amount of memory needed when using the low-storage pipelining implementation with (V) and without (III) stepsize control. In comparison to this, column IV shows the amount of memory needed when using the most space-efficient RK methods that have been proposed in related works, which make use of special coefficient sets and require at least two n-registers. Finally, column VI shows the storage space required when using a conventional embedded RK method with 7 stages and stepsize control or a conventional RK method with 8 stages without stepsize control.

Concerning data set sizes, the parallel implementation requires a larger amount of memory than the sequential implementation because every processor must provide sufficient space to store the data accessed in one pipelining step. Thus, the total amount of memory required increases with the number of processors and may even exceed the total amount of memory required by other implementations as the number of processors gets close to the number of blocks $n_B = \lceil n/B \rceil$, where $n_B = N$ in the case of semi-discretized PDEs.

As an example, we consider the case of non-blocking communication, where the working space of one pipelining step contains

$$Q_s^B = \left(\frac{1}{2}s^2 + \frac{7}{2}s - 3\right) B$$

vector elements. Including one additional register to perform stepsize control, our parallel low-storage implementation must store a total amount of $2n + p \cdot Q_s^B$ vector elements. On the other hand, the conventional implementations must store $(s+2)n + 2s(p-1)B$ vector elements. Hence, the low-storage implementation justifies its name if (assuming that n is an integer multiple of B)

$$\begin{aligned}
2n + p \cdot Q_s^B &< (s+2)n + 2s(p-1)B \\
2n + p\left(\frac{1}{2}s^2 + \frac{7}{2}s - 3\right) B &< (s+2)n + 2s(p-1)B \\
2n_B + p\left(\frac{1}{2}s^2 + \frac{7}{2}s - 3\right) &< (s+2)n_B + 2s(p-1) \\
p\left(\frac{1}{2}s^2 + \frac{3}{2}s - 3\right) &< s(n_B - 2) \\
p &< \frac{n_B - 2}{\frac{1}{2}s + \frac{3}{2} - \frac{3}{s}} \ .
\end{aligned} \tag{10}$$

In other words, storage space can be saved if we choose the number of blocks per processor, $\lfloor n_B/p \rfloor$, such that

$$\left\lfloor \frac{n_B}{p} \right\rfloor > \frac{1}{2}s + \frac{3}{2} - \frac{3}{s} + 1 \geq \frac{1}{2}s + \frac{3}{2} - \frac{3}{s} + \frac{2}{p} \tag{11}$$

for $p \geq 2$, i.e., if we assign more than $\frac{1}{2}s + \frac{3}{2} - \frac{3}{s} + 1$ blocks to each processor. Table 3 shows the minimum number of blocks to be assigned to each processor for RK methods with up to 17 stages.

Table 4 continues Table 2 by showing exemplary data set sizes for the case of a non-blocking parallel implementation with $s = 7$ (e.g., DOPRI 5(4)) and $B = 2N$ (e.g., BRUSS2D). For each grid size N, a number of processors, p (VIII), has been chosen as an example, which is high enough to benefit from parallelism, but small enough to save a significant amount of storage space. Column IX shows the size of the storage space required for each processor to store the working space of one pipelining step. In column X, this size has been multiplied by the number of processors, p, thus representing the total amount of storage space required to store the working spaces of the pipelining steps on all processors. The total amount of storage space including the two n-registers required by the non-blocking low-storage implementation is shown in column XI. For comparison, column XII shows the total amount of storage space required by the conventional parallel implementations that exploit the limited access distance.

5 Experimental Evaluation

The implementation variants described in Section 3 and 4 have been evaluated using runtime experiments on several modern parallel computer systems with different architectures. The computer systems considered are two off-the-shelf multi-processor servers equipped with Intel Xeon and AMD Opteron quad-core processors, one commodity cluster system with three different communication networks and two supercomputer systems, the Jülich Multiprocessor (JUMP) at the Jülich Supercomputing Centre (JSC) and the High End System in Bavaria 2 (HLRB 2) at the Leibniz Supercomputing Centre (LRZ) in Munich. Table 5 summarizes the target platforms, the compilers and the MPI versions used. Table 6 shows the implementation variants considered.

		I	II	III	IV	V	VI
N	n	$S\left(P_7^{2N}\right)$	$S(n)$	$S\left(n+P_7^{2N}\right)$	$S(2n)$	$S\left(2n+P_7^{2N}\right)$	$S((7+2)n)$
10^2	$2\cdot10^4$	62.5 KB	156.2 KB	218.8 KB	312.5 KB	375.0 KB	1.4 MB
10^3	$2\cdot10^6$	625.0 KB	15.3 MB	15.9 MB	30.5 MB	31.1 MB	137.3 MB
10^4	$2\cdot10^8$	6.1 MB	1.5 GB	1.5 GB	3.0 GB	3.0 GB	13.4 GB
10^5	$2\cdot10^{10}$	61.0 MB	149.0 GB	149.1 GB	298.0 GB	298.1 GB	1.3 TB
10^6	$2\cdot10^{12}$	610.4 MB	14.6 TB	14.6 TB	29.1 TB	29.1 TB	131.0 TB

Tab. 2: Comparison of data set sizes of different RK integrators occurring in the solution of an ODE system of size $n = 2N^2$ with access distance $d(\mathbf{f}) = 2N$ (sequential execution). See explanation in the text.

Stages (s):	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Blocks per processor:	3	4	4	5	6	6	7	7	8	8	9	9	10	10	11	11

Tab. 3: Minimum number of blocks to be assigned to each processors in an implementation with non-blocking communication in order to save storage space according to (11).

As an example problem we use BRUSS2D [10], a typical example of a 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. [29, 31]), this problem has a limited access distance of $d(\mathbf{f}) = 2N$, where the system size is $n = 2N^2$. As weight of the diffusion term we use $\alpha = 2 \cdot 10^{-3}$. The experiments presented in the following have been performed using the 3-stage embedded RK method RKF 2(3) [43, p. 64], the 7-stage embedded RK method DOPRI 5(4) [16], and the 13-stage embedded RK method DOPRI 8(7) [37].

5.1 Sequential Implementations

Figure 5 shows a comparison of the sequential runtimes of the three sequential implementations (D), (PipeD) and (PipeDls) on the five target systems. The runtimes have been normalized with respect to the number of steps and the number of ODE components, n. Thus, if machine instructions had constant execution times, the normalized runtime would be independent of n, except for slight differences caused by a different number of rejected steps. But, actually, the execution times of machine instructions vary, in particular those of memory instructions, for which the execution times depend on the location of the accessed data within the memory hierarchy.

In the diagrams in Fig. 5, the normalized runtime is plotted against the dimension of the ODE system, n, thus making visible the changes of the execution times of the memory instructions for increasing amounts of data to be processed. Usually, the highest increase of the normalized runtime occurs when relevant working sets grow larger than the highest-level cache, which is largest in size. In this case, the number of main memory accesses increases, which take a multiple of the time of a cache hit. Hence, for our application, it is often less significant, *which* cache level is hit, but primary important *that* the cache is hit.

5.1.1 4×4 *Xeon Server*

The largest differences in the runtimes of the three implementations have been observed on the Xeon processor (Hydra). If the system size is small, e.g., $n \leq 80\,000$, several n-vectors can be stored in the 3 MB L2 cache. For RKF 2(3), nearly the complete working set of one time step fits in the L2 cache. When n or the number of stages, s, increases, more memory access operations lead to cache misses and the normalized runtime increases. For n larger than approximately 10^6, the normalized runtime stays relatively constant. Only in the experiment using DOPRI 8(7) with 13 stages, the normalized runtime of the two pipelining implementations slowly increases. This behavior corresponds to the ratio between the size of the working space of a pipelining step and the size of the L2 cache. While this working set occupies only at most 10 % of the L2 cache in the experiment with RKF 2(3), it grows up to about half the size of the L2 cache in the experiment with DOPRI 5(4) and even grows larger than the L2 cache for $n > 5.8 \cdot 10^6$ in the experiment with DOPRI 8(7).

Since in our experiments n ranges from $8\cdot10^5$ to $8\cdot10^6$, the diagrams showing the normalized runtime on Hydra illustrate the situation where working sets representing one or several n-vectors do not fit in the cache. Only the loop performing the pipelining steps and the innermost loop over the succeeding stages can run inside the cache. In this situation, the pipelining implementations clearly outperform the general implementation (D). The relation between the normalized runtimes of the pipelining implementation (PipeD) and its low-storage variant (PipeDls) is unique on this machine. Here, (PipeDls) is about 1.5 to 2 times faster than (PipeD). Such high speedup had not been expected since the computation order and the size of the working space are the same in both implementations.

On the other hand, (PipeDls) stores the data accessed in the pipelining steps in a very compact memory region. Even more important is the number of cache lines that have to be replaced when the implementations ad-

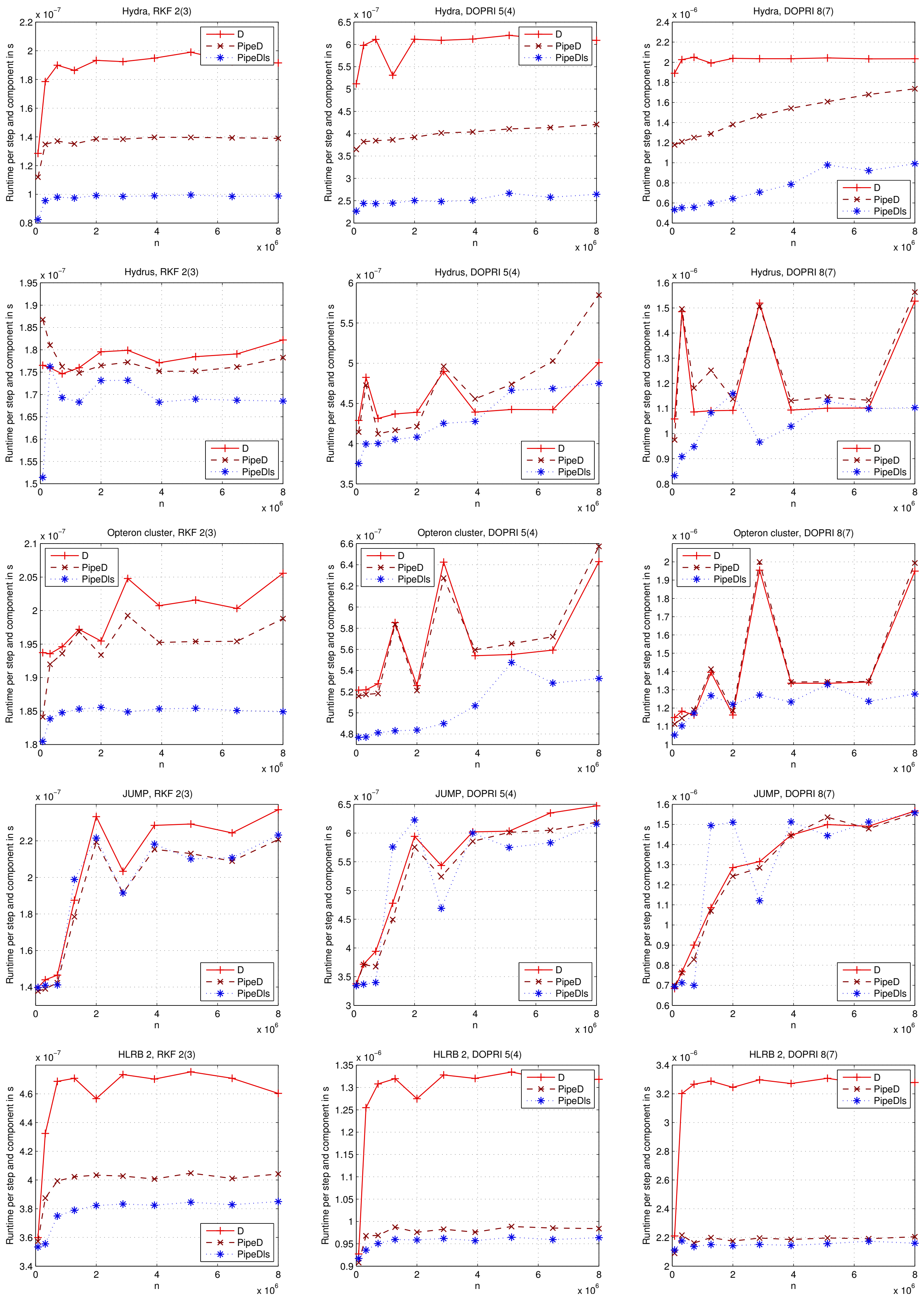

Fig. 5: Sequential runtime per step and component in seconds.

		VIII	IX	X	XI	XII
N	n	p	$S\left(Q_7^{2N}\right)$	$S\left(p \cdot Q_7^{2N}\right)$	$S\left(2n + p \cdot Q_7^{2N}\right)$	$S((7+2)n + 2s(p-1)2N)$
10^2	$2 \cdot 10^4$	5	71.9 KB	359.4 KB	671.9 KB	1.5 MB
10^3	$2 \cdot 10^6$	25	718.8 KB	17.5 MB	48.1 MB	142.5 MB
10^4	$2 \cdot 10^8$	100	7.0 MB	701.9 MB	3.7 GB	13.6 GB
10^5	$2 \cdot 10^{10}$	1000	70.2 MB	68.5 GB	366.6 GB	1.3 TB
10^6	$2 \cdot 10^{12}$	10000	701.9 MB	6.7 TB	35.8 TB	133.0 TB

Tab. 4: Comparison of data set sizes of different RK integrators occurring in the solution of an ODE system of size $n = 2N^2$ with access distance $d(\mathbf{f}) = 2N$ (parallel execution with non-blocking communication). See explanation in the text.

vance to the next pipelining step. (PipeD) stores all n-vectors ($\boldsymbol{\eta}_\kappa$, $\mathbf{w}_2, \ldots, \mathbf{w}_s$, $\Delta\boldsymbol{\eta}$, and $\mathbf{e}$) in separate n-registers. Thus, assuming that the working space of a pipelining step fits in the cache, $s+2$ blocks containing B vector elements must be loaded into the cache, where they replace $s+2$ other blocks of this size. Except for the block of $\boldsymbol{\eta}_\kappa$, all replaced blocks contain updated values and have to be written back to main memory.

By contrast, (PipeDls) uses only one register of size n to store $\boldsymbol{\eta}_\kappa$ and one larger register to store $\Delta\boldsymbol{\eta}$ and the working space of the pipelining steps. When (PipeDls) moves on to the next pipelining step, only two blocks of size B have to be loaded into the cache and one block needs to be written back to main memory. The blocks which are not needed in future pipelining steps, are overwritten and thus reused in the next pipelining step.

The storage size of a block containing B vector elements grows from 3.13 KB to 31.25 KB in our experiments. Thus, in the worst case, (PipeDls) loads only 62.50 KB in each pipelining step into the cache while (PipeD) has to load 156.25 KB in case of RKF 2(3), 281.25 KB in case of DOPRI 5(4), and 468.75 KB in case of DOPRI 8(7). Since the difference in normalized runtime between (PipeD) and (PipeDls) is higher if the number of stages is larger, this presumably is the reason for the higher performance of (PipeDls) on Hydra.

However, the blocks of the $s+2$ n-registers in implementation (PipeD) which are required in the next pipelining step could be prefetched in parallel to the computation of the current pipelining step. But this can only work if the prefetch logic of the processor recognizes the access pattern of the implementation and if the memory-to-cache transfer is fast enough.

5.1.2 Opteron-based Systems

On the other target platforms the difference in the performance of (PipeD) and (PipeDls) is not as large as on Hydra. The two systems Hydrus and the Opteron cluster are both equipped with AMD Opteron processors. Even though the processors of the two systems represent different generations of the Opteron processor (K8 and K10) our measurement results show many similarities between the two systems. Differences between the two systems mainly result from different cache sizes. While the K8 processors in the Opteron cluster have an L2 cache of 1024 KB, the K10 processors of Hydrus have a smaller L2 cache of only 512 KB but possess an additional shared L3 cache of 2 MB L3 cache of 2 MB (non-inclusive victim cache [1]).

The first similarity observed is that the normalized runtime curves of (D) and (PipeD) show peaks while the curve of (PipeDls) is significantly smoother. The peaks are sharper if the number of stages is higher. The implementations (D) and (PipeD) have in common that the innermost loops iterate over the stages while the argument vectors are allocated separately. In contrast to this, the innermost loop of (PipeDls) also iterates over the stages, but the argument vectors are embedded into a single register such that the vector elements accessed by the innermost loop are located inside a small consecutive memory region. Most importantly, the distance between the memory locations of the elements of succeeding argument vectors accessed in the innermost loop is not a constant amount.

Therefore, the innermost loops of (D) and (PipeD) are more likely to provoke cache conflict misses than the innermost loop of (PipeDls) if the associativity of the cache is smaller than the number of vectors accessed in the innermost loop. Since the L1 cache of the Opteron processors is only 2-way associative, this leads to the conclusion that the peaks in the normalized runtime curves of (D) and (PipeD) are caused by an increased number of conflict misses due to an unfortunate alignment of the argument vectors. Additional tests with a randomized allocation policy support this conclusion as they led to different shapes of the normalized runtime curves.

Comparing (D) and (PipeD), (PipeD) can outperform (D) only if the working space of one pipelining step fits in the L2 cache. This is the case in the experiment with RKF 2(3). In the experiment with DOPRI 5(4), the working space of one pipelining step grows larger than the cache for $N \gtrsim 1.3 \cdot 10^6$ on Hydrus and for $N \gtrsim 5.4 \cdot 10^6$ on the Opteron cluster. In the experiment with DOPRI 5(4), this happens for $N \gtrsim 1.6 \cdot 10^5$ on Hydrus and for $N \gtrsim 6.5 \cdot 10^5$ on the Opteron cluster.

The low-storage implementation (PipeDls) is clearly faster than (PipeD) in nearly all experiments, but its performance also depends on whether the working space of one pipelining step fits in the L2 or L3 cache. Therefore, (D) can be faster than (PipeDls) for large system sizes such as $n \gtrsim 5.0 \cdot 10^6$ in the experiment with DOPRI 5(4) on Hydrus.

Name	HLRB 2	JUMP	Hydra	Hydrus	Opteron cluster
Institution	LRZ Munich	JSC	University of Bayreuth	University of Bayreuth	University of Bayreuth
HW Platform	SGI Altix 4700	IBM p6 575	Intel ‘Fox Cove’ S7000FC4URE	MEGWARE Saxonid C16	MEGWARE Saxonid C2
CPU	Intel Itanium 2 (Montecito)	IBM Power 6	Intel Xeon E7330 (Core, Tigerton)	AMD Opteron 8350 (K10, Barcelona)	AMD Opteron 246 (K8, Sledgehammer)
Clock (GHz)	1.6	4.7	2.4	2.0	2.0
L1 Cache (KB)	16[1]	$64 + 64$[2]	$32 + 32$[2]	$64 + 64$[2]	$64 + 64$[2]
L2 Cache (KB)	$1024 + 256$[2]	4096	2×3072[3]	512[4]	1024[4]
L3 Cache (MB)	9	32[5,6]	–	2[5,7]	–
#Cores/CPU	2	2 (4)[8]	4	4	1
#Cores/Node	512	32 (64)[8]	16	16	2
#Cores (total)	9728	448 (896)[8]	16	16	64
Interconnect(s)	NUMAlink 4	Infiniband	n/a	n/a	Infiniband, Gigabit Ethernet, Fast Ethernet
Compiler	GCC 4.3	IBM XL C/C++ V 10.1	GCC 4.3	GCC 4.3	GCC 4.3
MPI Version	SGI MPT 1.16	IBM POE 4.3.2	MVAPICH2 1.2	MPICH 1.2.7	MPICH 1.2.5

[1] Not used for floating point data.
[2] Instructions + data.
[3] Two cores share a 3 MB L2 cache.
[4] Exclusive cache (does not store data present in L1 cache).
[5] Shared by all cores.
[6] Off-die cache.
[7] ‘Mostly exclusive’ victim cache running at northbridge clock.
[8] Number of virtual cores in simultaneous multi-threading (SMT) mode.

Tab. 5: Overview of the target platforms considered in the experimental evaluation.

Name	**Exploits Limited Access Distance**	**Pipe-lining**	**Low Storage**	**Distr. Data Structures**	**Non-bl. Comm.**	**Figures**
Sequential Implementations						
D	–	–	–	n/a	n/a	
PipeD	✓	✓	–	n/a	n/a	2 (a), 2 (b)
PipeDls	✓	✓	✓	n/a	n/a	2 (a), 2 (b), 4 (a)
Pthreads Implementations for Shared Address Space						
D	–	–	–	–	–[1]	
Dbc	✓	–	–	–	–[2]	1
PipeD	✓	✓	–	–	–[2]	2 (b), 3 (a)
PipeD4	✓	✓	–	✓	–[2,3]	2 (b), 3 (c)
PipeD4ls	✓	✓	✓	✓	–[2,3]	2 (b), 3 (c), 4 (a)
MPI Implementations for Distributed Address Space						
D	–	–	–	✓	–[4]	
Dbc	✓	–	–	✓	✓[5]	1
PipeD	✓	✓	–	✓	✓[5]	2 (b), 3 (a)
PipeD2	✓	✓	–	✓	✓[5]	2 (b), 3 (b)
PipeD4	✓	✓	–	✓	✓[5]	2 (b), 3 (c)
PipeD5	✓	✓	–	✓	✓[5]	2 (b), 3 (c)
PipeD4ls	✓	✓	✓	✓	✓[5]	2 (b), 3 (c), 4 (b)

[1] The computation of the argument vector $\mathbf{w}$ and the function evaluation $\mathbf{f}(t, \mathbf{w})$ are separated by barrier operations.

[2] Blocks of the argument vectors accessed by multiple processors are protected by locks. At the beginning of the time step, the locks are held by the processors which will compute the corresponding block. The locks are released as soon as the computation of the corresponding block has been completed.

[3] Blocks are copied using `memcpy()`.

[4] The argument vector $\mathbf{w}$ is made available to all processors using a multibroadcast operation (`MPI_Allgather()`).

[5] Blocks are send to/received from neighboring processors by non-blocking single transfer operations (`MPI_Isend()`/`MPI_Irecv()`).

Tab. 6: Overview of the implementation variants considered in the experimental evaluation.

5.1.3 IBM p6 575 Cluster

JUMP stands out by its huge 32 MB L3 cache. Also its 4 MB L2 cache is the largest among the five target systems considered. Thus, even in the experiment with DOPRI 8(7) where $s = 13$, the working space of one pipelining step fits in the L2 cache. For $n = 80\,000$, the entire working space of one time step fits in the L3 cache.

As n grows from $8 \cdot 10^5$ to $8 \cdot 10^6$, the working sets corresponding to one or more n-vectors drop out of the L3 cache and the normalized runtimes increase and stabilize when the size of one n-vector becomes larger than the L3 cache. This happens for $n \approx 4.2 \cdot 10^6$. Comparing the three RK methods, the scaling of the diagrams gives the visual impression that the slope of the curves is steeper if the number of stages is smaller. In fact, in the interval $n = 8 \cdot 10^5, \dots, 2.1 \cdot 10^6$, where the number of n-vectors that fit in the L3 cache decreases to 2, the normalized runtime grows by $\approx$57–67 % for RKF 2(3), by $\approx$71–86 % for DOPRI 5(4), and by $\approx$77–115 % for DOPRI 8(7). That means, in this interval, the slopes are steeper if s is larger, what can be explained by the larger portion of the working space of one time step that no longer fits in the L3 cache. In the range $n = 2.1 \cdot 10^6, \dots, 4.2 \cdot 10^6$, where the number of n-vectors that fit in the L3 cache decreases from 2 to 1, a significant increase of the normalized runtime has been observed only for DOPRI 8(7).

5.1.4 SGI Altix 4700

The Itanium 2 processor used in the HLRB 2 system contains a 9 MB L3 cache, which is large enough to store the working space of a pipelining step even in the experiment with the 13-stage method DOPRI 8(7). But for $n \geq 1.18 \cdot 10^6$ the storage space required for one n-vector is larger than the L3 cache. Thus, for $n = 8 \cdot 10^5, \dots, 1.28 \cdot 10^6$ we observe that the normalized runtimes quickly increase and then stay almost constant for $n \geq 1.28 \cdot 10^6$. The pipelining implementations are considerably faster than the general implementation (D). Moreover, due to its higher locality resulting from the more compact storage of the working space, the low-storage variant of the pipelining scheme, (PipeDls), runs slightly faster than the conventional pipelining implementation (PipeD).

5.2 Shared-Address-Space Implementations

In this section, we compare the speedups and the efficiencies of the parallel Pthreads implementations for the grid sizes $N = 1000$ (Figs. 6 and 7) and $N = 2000$ (Figs. 8 and 9) on the four target systems Hydra, Hydrus, JUMP and HLRB 2 described in Table 5.

Since our intention is to compare the new low-storage implementation of the pipelining scheme with the pre-existing pipelining implementation (PipeD), we use the sequential execution time of (PipeD) as the reference for the speedup and efficiency computation.

5.2.1 4 × 4 Xeon Server

On Hydra, which is equipped with four quad-core Xeon (Tigerton) processors, the scalability of the Pthreads implementations is generally not as good as on the other systems. Only the low-storage pipelining implementation (PipeD4ls) can obtain speedups above 7. While all other shared-memory systems considered rely on a NUMA architecture, where each processor can access its local memory very fast without interfering with local memory accesses of other processors, Hydra uses a centralized memory architecture. All processor chips access the main memory through a central 'Memory Controller Hub' (Northbridge), which turns out to be a bottleneck for memory-intensive parallel applications.

The comparison of the sequential runtimes already showed that the performance on this machine is sensitive to changes of the locality behavior. The parallel execution amplifies these performance differences. Thus, the scalability of the general implementation (D) and the implementation (Dbc), which uses the same loop structure as (D), is very poor. Their speedups are below 4. The conventional pipelining implementations (PipeD) and (PipeD4) obtain noticeably better speedups up to ≈ 7 but cannot take profit of all cores of the machine.

In our experiments, (PipeD4ls) is the only implementation which can exploit all cores of the machine, but only if the amount of data to be processed is not too large. The highest speedups with respect to the sequential execution time of (PipeD) observed for (PipeD4ls) was 16.8 in the experiment using DOPRI 5(4) and $N = 1000$. However, the self-reflexive speedup in this case is only 10.4. If the number of stages or the size of the ODE system is increased, the maximum speedup is reached on less than 16 cores (see experiments with DOPRI 8(7), $N = 1000$ and DOPRI 5(4), $N = 2000$).

5.2.2 4 × 4 Opteron Server

On the second system equipped with four quad-core processors, Hydrus, all Pthreads implementations – the specialized implementations, but also the general implementation (D) – scale well and can exploit all cores of the machine. In contrast to Hydra, the Opteron processors of Hydrus have local memories and are connected by point-to-point links ('HyperTransport').

The speedups measured on this machine using 16 threads range from 10.2 to 18.3. The sequential runtimes of the different implementations already were very similar, and the same applies to the parallel speedups. The only exception is the experiment with DOPRI 8(7) and $N = 2000$, where (PipeD4ls) is significantly faster than the other implementations.

5.2.3 IBM p6 575 Cluster

The nodes of JUMP are equipped with 32 Power 6 cores. Each core can execute two threads in hardware (two-way SMT). Among the systems considered, JUMP has the fastest processor clock (4.7 GHz) and the largest cache

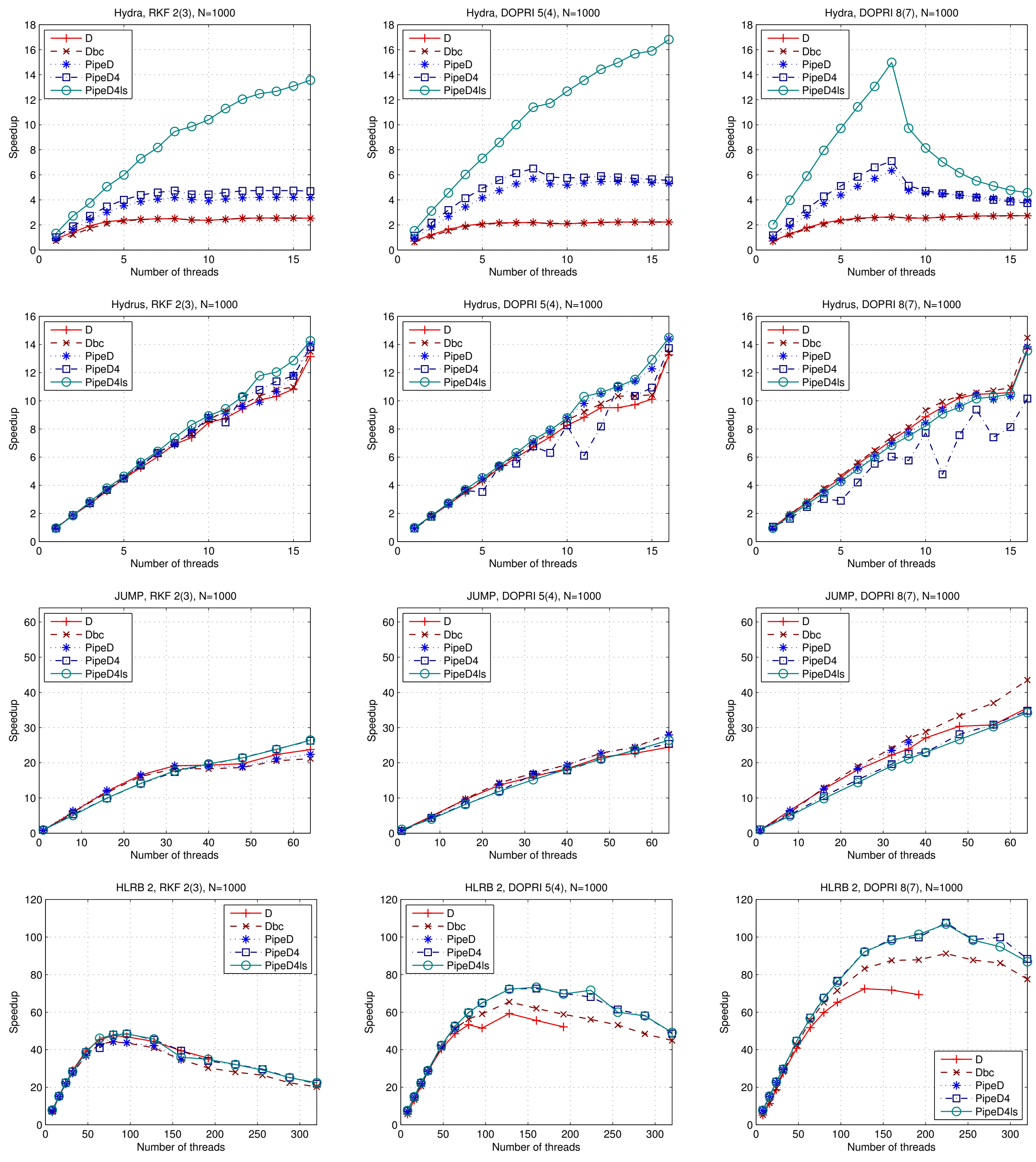

Fig. 6: Parallel speedup of the shared-address-space implementations ($N = 1000$).

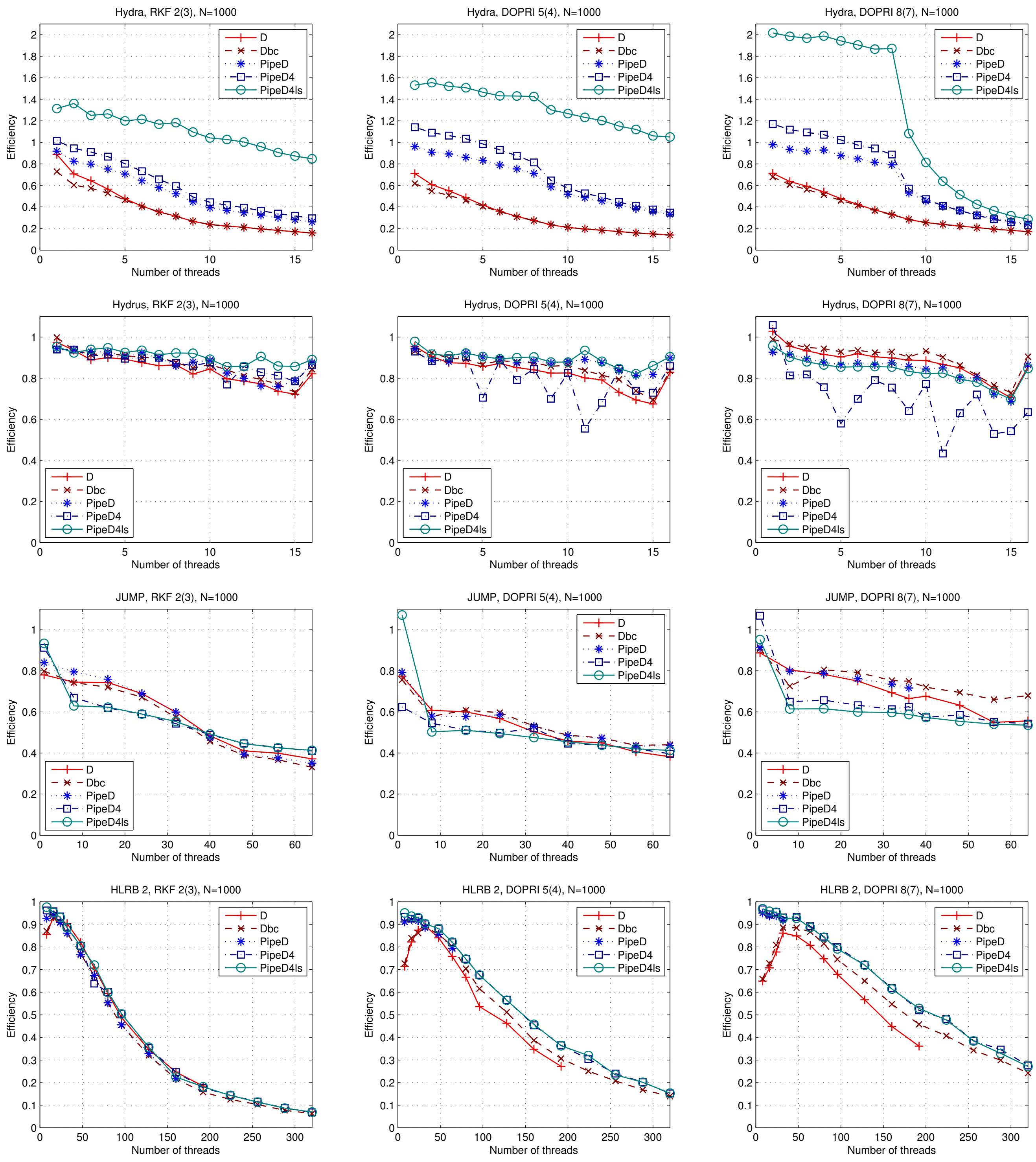

Fig. 7: Parallel efficiency of the shared-address-space implementations ($N = 1000$).

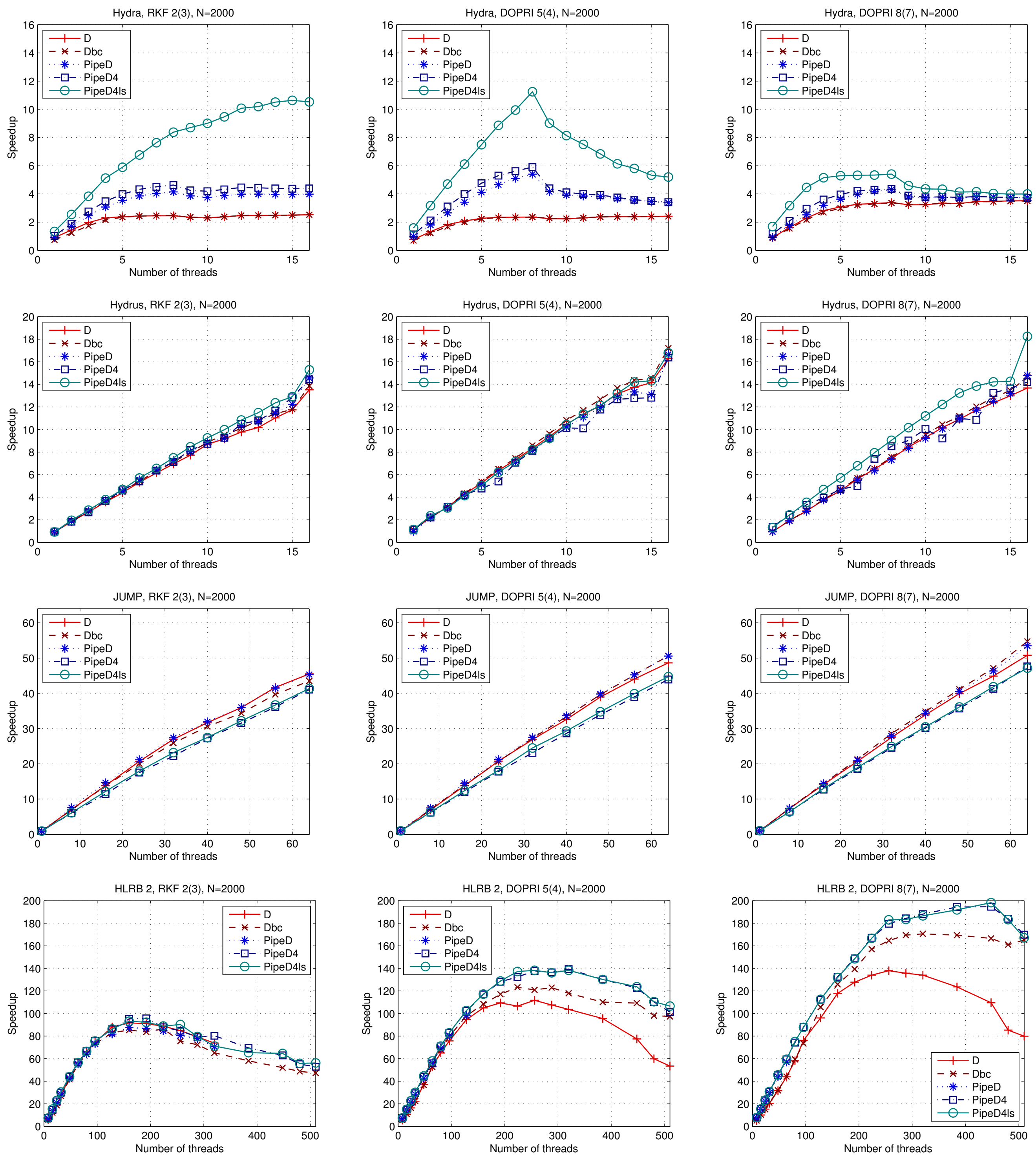

Fig. 8: Parallel speedup of the shared-address-space implementations ($N = 2000$).

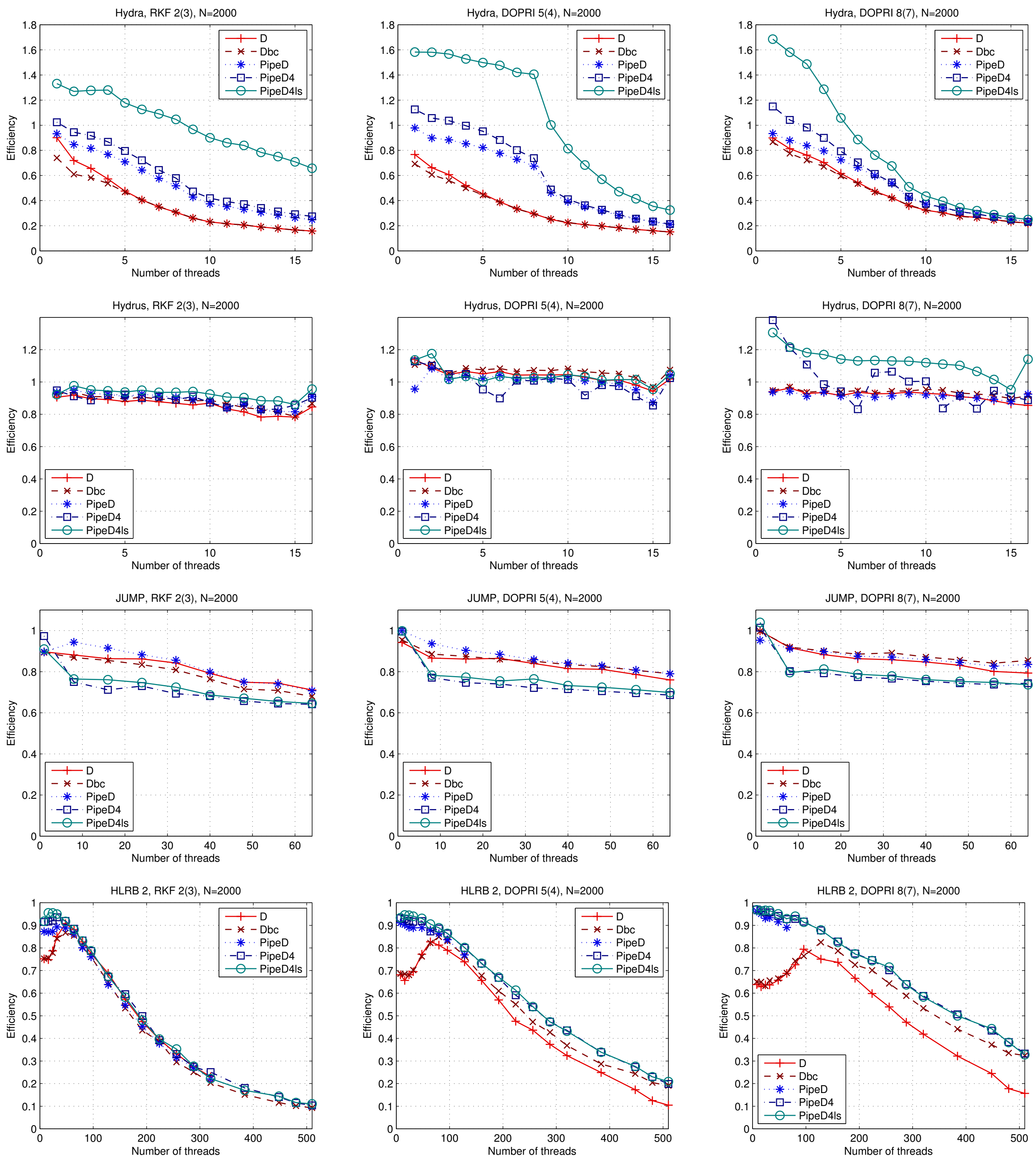

Fig. 9: Parallel efficiency of the shared-address-space implementations ($N = 2000$).

(32 MB). In the experiments with $N = 2000$, the scalability of all implementations is very good. The speedups for 32 threads (ST mode) lie between 22.2 and 27.4 for RKF 2(3), between 23.1 and 27.5 for DOPRI 5(4), and between 24.5 and 28.5 for DOPRI 8(7).

It is particularly remarkable that in SMT mode (i.e., using more than 32 threads) the efficiency with respect to the number of threads decreases only slightly, and with respect to the number of physical cores it nearly doubles. Apparently, the two threads of instructions executed on each Power 6 core, which mainly consist of floating-point instructions, can be scheduled to the two binary floating-point units available on each core such that the two threads are not significantly delayed by contention for processor resources, and the memory hierarchy can satisfy the higher rate of data requests generated by the second thread without delaying the data required by the first thread. Thus, for $N = 2000$, speedups between 41.0 and 45.5 have been obtained for RKF 2(3), between 43.9 and 50.6 for DOPRI 5(4), and between 47.1 and 54.7 for DOPRI 8(7). In the experiments with $N = 1000$, speedups and efficiencies are lower, but the implementations can still exploit all cores of the machine and even take profit of the SMT capability.

Comparing the implementation variants, the differences are not very large. We observe, however, that (PipeD), (Dbc) and (D) are often noticeably faster than (PipeD4) and (PipeD4ls). (PipeD4) and (PipeD4ls) use the same computation order but a different memory layout of the data structures. Both use distributed data structures and copy blocks between neighboring threads using `memcpy()`. By contrast, (PipeD), (Dbc) and (D) use shared (central) data structures and do not need to copy blocks. Thus, it seems likely that the overhead resulting from the copying of blocks has a negative effect on the overall performance. Further, the direction of the iteration over the dimension of the ODE system has an influence on the performance, c.f. Section 5.3.3.

5.2.4 SGI Altix 4700

HLRB 2 has a globally shared address space based on a NUMAlink network with a dual fat tree topology. Parallel thread-based jobs can use up to 510 Itanium 2 cores. Our Pthreads implementations, however, run efficiently only on small numbers of cores. The highest speedups have been measured using DOPRI 8(7) and $N = 2000$. In this experiment, the speedups on 96 cores are in the range from 73.3 to 88.0. The highest speedups of 198.3 and 194.7 are reached by the implementations (PipeD4ls) and (PipeD4) using 448 threads. Both implementations, (PipeD4) and (PipeD4ls), deliver very similar parallel runtimes, which means that the low-storage variant neither improves nor deteriorates the performances.

Also in most other experiments these two implementations deliver the highest speedups. This means that on this system the higher locality of the distributed data structures appears to be an advantage even though it entails the overhead of block transfers. However, for large numbers of threads, (PipeD4) and (PipeD4ls) can only be compared to (Dbc) and (D) since (PipeD) supports at most $\lfloor n_B/(2s) \rceil$ threads. But the comparison with (PipeD) for small numbers of threads already confirms that on this machine, where memory access times are significantly more non-uniform than on the other systems, distributed data structures lead to better performance. The reason is that, to access a shared data structure, all threads which run on processors not directly connected to the memory section in which the data structure resides have to perform expensive remote memory accesses.

(D) and (Dbc) differ in the synchronization mechanisms employed, but both use the same loop structure. For small numbers of threads, the influence of the synchronization mechanisms is less important, and the performance of (D) and (Dbc) does not differ significantly. In this situation, the loop structure of (D) and (Dbc) is not competitive to the pipelining scheme. But as the number of threads increases, the amount of data to be processed by the individual processors decreases and a larger part of this data fits in the L3 cache. Thus, the number of main memory accesses is reduced and the performance improves. The best efficiency is obtained if the amount of data to be processed by each processor has approximately the size of the L3 cache. If the number of threads is increased further, this behavior stops taking effect since all data fit in the local caches and no more main memory accesses are saved. Consequently, the increasing synchronization overhead of the larger number of threads leads to a decreasing efficiency as it is not compensated by sufficiently strong positive cache effects. Regarding the influence of the synchronization operations, the comparison of (D) and (Dbc) for large numbers of threads shows that the lock-based synchronization strategy of (Dbc), which makes use of the limited access distance of the test problem, leads to a better scalability than the barrier-based general strategy of (D) in most experiments.

Compared to the MPI implementations (see Section 5.3.4), the general scalability of the Pthreads implementations is far below the potential of the machine. As our comparison shows, scalable implementations require the use of distributed data structures because only they can reflect the physically distributed organization of the globally addressable main memory. We assume that for the Pthreads implementations which use distributed data structures, (PipeD4) and (PipeD4ls), the overhead of the block transfers limits the scalability since the Pthreads implementations – unlike the MPI implementations – do not overlap the block transfers by computations.

5.3 Distributed-Address-Space Implementations

In this section, we compare the speedups and the efficiencies of the MPI implementations for the grid sizes $N = 1000$ (Figs. 10 and 11) and $N = 2000$ (Figs. 12 and 13) on the five target systems described in Table 5.

As in Section 5.2 where we considered the Pthreads implementations, we use the sequential execution time of (PipeD) as the reference for the speedup and efficiency computation. In addition to the MPI variants of (D), (Dbc), (PipeD), (PipeD4) and (PipeD4ls), we also consider the implementation variants (PipeD2) and (PipeD5) (cf. Section 3.4).

5.3.1 4×4 *Servers*

In general, on both 4×4 servers the speedups and the efficiency of the MPI implementations are similar to those of the Pthreads implementations. The exchange of messages between processes via the MPI library does not lead to a high overhead compared to the performance of the Pthreads implementations, which can communicate using shared variables. Instead, the performance on Hydra is determined by the locality of the implementations and their ability to save main memory accesses. Thus, the low-storage pipelining implementation, (PipeD4ls), clearly outperforms the other implementations. On Hydrus almost all implementations scale well, and their performance does not differ significantly. Only in the experiment with DOPRI 8(7) and $N = 2000$ the low-storage implementation (PipeD4ls) shows a significantly higher efficiency than the other specialized implementations.

The only MPI implementation which delivers an evidently worse performance than its Pthreads counterpart is the general implementation (D), because it requires global communication. While the Pthreads version of (D) makes use of shared data structures and can synchronize the threads by executing one barrier operation at each stage, the MPI version requires the execution of a multibroadcast operation (`MPI_Allgather()`) at each stage, which assembles the current argument vector from its locally computed parts and makes it available to all participating MPI processes. This communication operation is very expensive; its costs increase with the number of participating processes. The MPI version of implementation (D) therefore shows a poor scalability in all experiments performed – not only on Hydra and Hydrus, but also on the other systems considered. Its speedup grows slowly or even decreases when the number of processes is increased, and its efficiency declines rapidly.

The two implementations (PipeD2) and (PipeD5), which have not been included in the comparison of the Pthreads implementations, deliver a similar performance like the other two conventional pipelining implementations, (PipeD) and (PipeD4). The performance of (PipeD5) is nearly identical to that of (PipeD4). (PipeD2) has approximately the same performance as (PipeD) or is slightly slower.

5.3.2 *Opteron Cluster*

The Opteron cluster consists of 32 dual-processor nodes connected by three separate communication networks at different speeds. This allows us to investigate the influence of the communication speed on the performance of the RK implementations.

Figures 10–13 show the parallel speedups and efficiencies measured using the fastest network, an Infiniband network with a nominal bandwidth of 10 Gbit/s. Using this network, all specialized implementations scale well, but the rate at which the efficiency decreases as more processors are used varies depending on the grid size N. The efficiency of the specialized implementations is 0.6 or better in the experiments with $N = 1000$, and it is 0.8 or better in the experiments with $N = 2000$. In contrast to this, the general implementation (D) hardly profits from a parallel execution because it requires expensive global communication.

Taking a closer look at the efficiency curves of the specialized implementations for the Infiniband network, we first notice a sharp drop of the efficiencies of some implementations from ≈ 0.9 to ≈ 0.7 in the experiment with DOPRI 8(7) and $N = 1000$ when the number of processes reaches 24 or 32, respectively. A similar but smaller drop of the efficiencies of these implementations occurs in the experiment with DOPRI 5(4) and $N = 1000$. The only specialized implementation not affected is (PipeD4ls), whose efficiency is 0.94 or better. No such drops of the efficiencies have been observed in the experiment with RKF 2(3) and $N = 1000$ and in all experiments with $N = 2000$. We conclude that this effect is caused by an increased number of conflict misses, because its magnitude increases with the number of stages and the innermost loops of our implementations iterate over the stages, thus touching elements of each of the s argument vectors which have the same index. (PipeD4ls) is the only implementation which does not store the argument vectors separately. Instead, (PipeD4ls) contracts the data which are part of the working space of the outermost loop into a small consecutive memory region, and the access patterns of the inner loops are less likely to produce conflict misses. However, no similar effect has been observed using the Gigabit Ethernet or Fast Ethernet network. This suggests that the way data are stored and managed by the network protocol (OpenIB or TCP/IP) influences the cache access pattern and the data layout in the cache, and this leads to a worse cache performance of some implementations when OpenIB is used instead of TCP/IP.

Even though this sharp drop of the efficiencies makes an evaluation of the experiment with $N = 1000$ and DOPRI 8(7) difficult, a clearer comparison of the implementations is possible for the other experiments. Using the Infiniband network, $N = 1000$ and RKF 2(3) or DOPRI 5(4), (PipeD4ls) obtains the highest efficiency. (PipeD4) and (PipeD5) are slightly slower, but there is no significant difference in the performance of these two implementations. Similarly, (PipeD) and (Dbc) show an equal performance, and both are slightly slower than (PipeD4) and (PipeD5), but clearly faster than (PipeD2). Thus, the most successful implementations in these experiments are those which use distributed data structures. Using the Infiniband network and $N = 2000$, the performance improvement of (PipeD4ls) over the other implementations is larger than for $N = 1000$, particularly if DOPRI 8(7) is used, but the differences of the

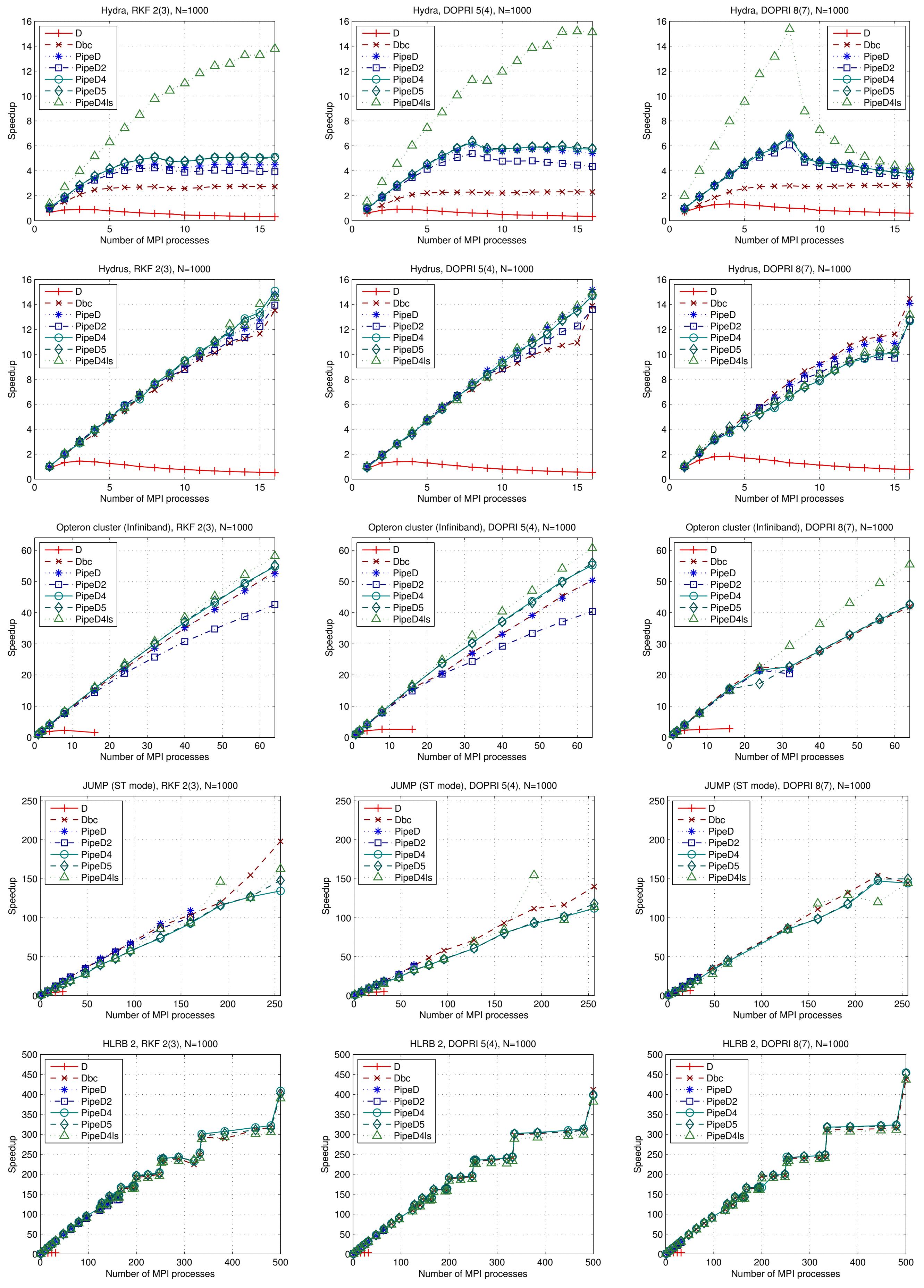

Fig. 10: Parallel speedup of the distributed-address-space implementations ($N = 1000$).

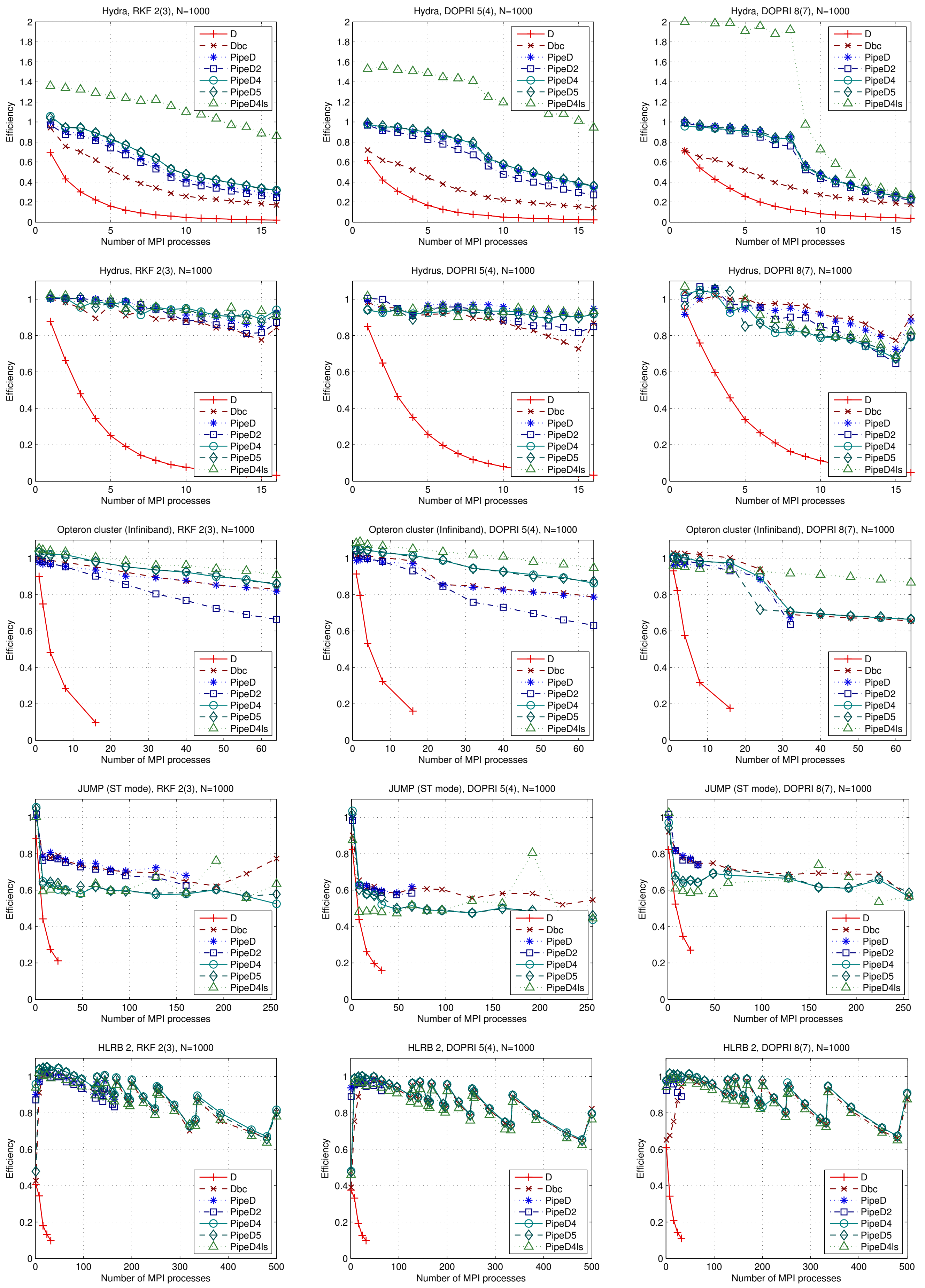

Fig. 11: Parallel efficiency of the distributed-address-space implementations ($N = 1000$).

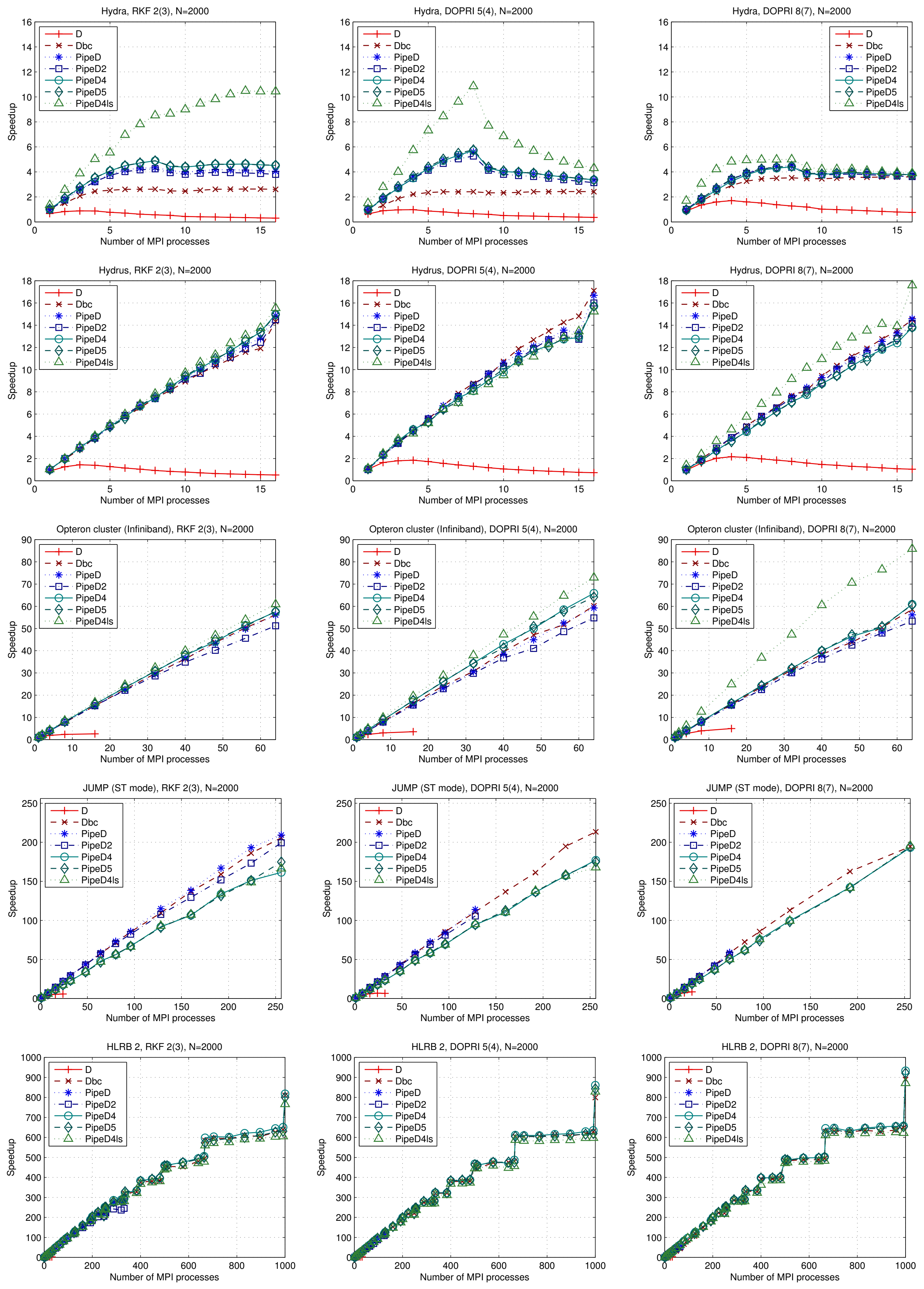

Fig. 12: Parallel speedup of the distributed-address-space implementations ($N = 2000$).

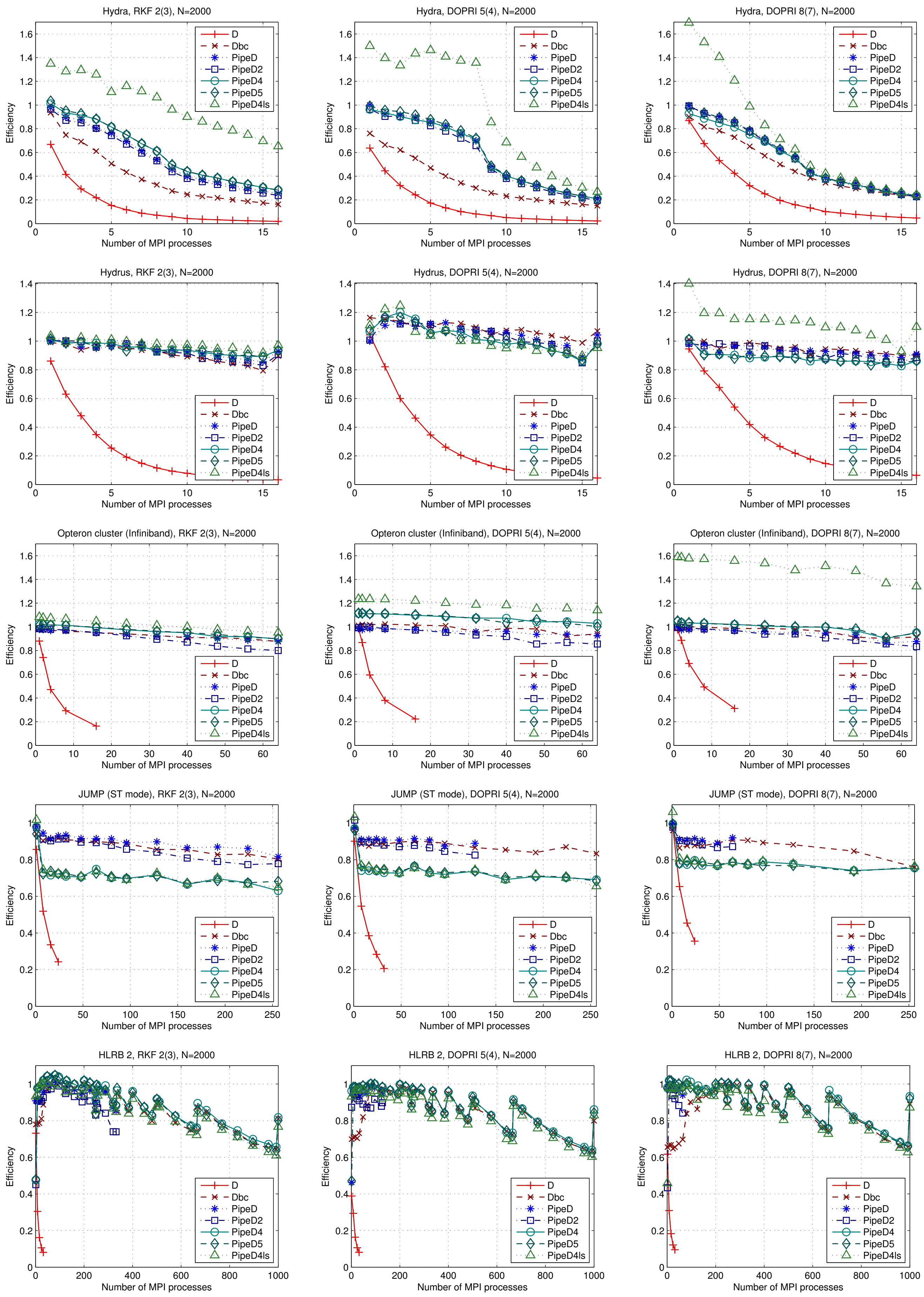

Fig. 13: Parallel efficiency of the distributed-address-space implementations ($N = 2000$).

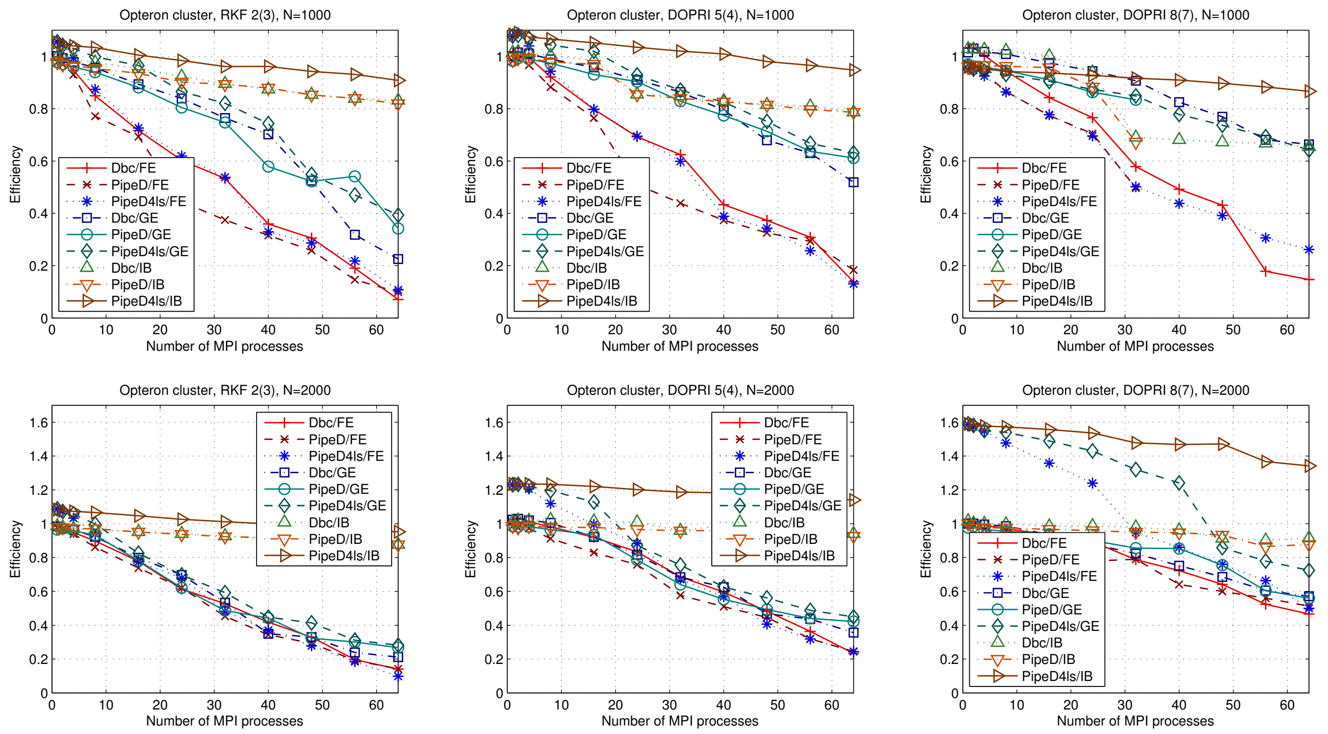

Fig. 14: Comparison of the parallel efficiency of the distributed-address-space implementations on the Opteron cluster using different networks (FE = Fast Ethernet, GE = Gigabit Ethernet, IB = Infiniband).

performances of the other specialized implementations compared with each other are smaller. Only (PipeD2) can always be identified as the slowest specialized implementation. Thus, for this larger system dimension, which requires more data and larger working sets to be processed, the reduction of the storage space and the simultaneous increase of spatial locality achieved by the low-storage implementation, (PipeD4ls), has a significantly larger influence on the execution time than the variations of the computation and communication pattern developed for the other implementations.

Figure 14 shows a comparison of the parallel efficiencies of the implementations (Dbc), (PipeD), and (PipeD4ls) on the three different interconnection networks. The speed of the interconnection network has a deciding influence on the scalability of our implementations. As one expects, the scalability generally improves with the network speed because communication costs decrease. Thus, usually the performance on the Gigabit Ethernet network (1 Gbit/s) is better than the performance on the Fast Ethernet network (100 Mbit/s), and the best performance can be obtain on the Infiniband network (10 Gbit/s). Exceptions are the experiments with $N = 1000$ and DOPRI 5(4) or DOPRI 8(7) where the efficiency curves of some implementations on the Infiniband network show a sharp drop when the number of processors exceeds some value as discussed above. However, the impact of the network speed on the performance depends on the dimension of the ODE system and the number of stages of the RK method. When the number of processors is increased on the Infinband network, the efficiencies stay nearly constant, i.e., they decrease only slowly. Compared to this, the efficiencies on the other networks degrade relatively quickly. Using $N = 2000$, i.e., the larger system dimension, the difference in the performance of the Fast Ethernet network and the Gigabit Ethernet network is only small. Using $N = 1000$, the difference in the performance of the networks depends to a greater extent on the number of stages of the RK method. Thus, while both the Fast Ethernet network and the Gigabit Ethernet network are significantly slower than the Infiniband network for RKF 2(3), for DOPRI 8(7) the performance of the Gigabit Ethernet network gets closer to the performance of the Infiniband network, whereas the performance of the Fast Ethernet network does not improve noticeably when the number of stages is increased.

5.3.3 IBM p6 575 Cluster

While the Pthreads implementations are confined to the shared address space of a single cluster node, the MPI implementations can communicate across node boundaries. We can choose ST or SMT mode by running up to or more than 32 MPI processes per node. Figures 10–13 show the speedups and the efficiencies observed in ST mode for up to 256 MPI processes.

The performance of the general implementation (D) is as bad as on the other systems considered because it requires global communication. This behavior can already be observed using small numbers of processes. We therefore took measurements with this implementation using only at most 32 processes on a single node.

In both ST and SMT mode, the specialized implementations fall into two groups. The group which obtains a higher efficiency consists of (Dbc), (PipeD) and (PipeD2). These implementations have in common that all processors process the pipelines towards increasing array indices. The second group consists of the implementations (PipeD4), (PipeD5) and (PipeD4ls), which apply the computation order illustrated in Fig. 3 (c). A comparison with an additional implementation variant of (PipeD) which iterates over the system dimension in opposite direction, i.e., towards decreasing array indices, indicates that the iteration direction is the deciding factor why the implementations in the second group show a lower efficiency.

All in all, the efficiency observed on JUMP is lower than on Hydrus and on the Opteron cluster using the Infiniband network. In our experiments with up to 256 processors, the efficiency does, however, remain almost constant or decreases only slowly if the number of processors is increased, which represents a high scalability. The different strategies of (PipeD4) and (PipeD5) to handle the blocks of $\boldsymbol{\eta}_\kappa$ that are required to start the initialization of the pipelines are not reflected in significant runtime differences. But the alternative finalization strategy used by (PipeD2) shows a tendency to result in worse runtimes than the finalization strategy of (PipeD). In most of the experiments performed on JUMP, the low-storage implementation (PipeD4ls) is competitive with its base implementation (PipeD4). Only in the experiments using DOPRI 5(4) and DOPRI 8(7) with $N = 1000$ the performance of (PipeD4ls) is slightly worse than that of (PipeD4).

Figure 15 compares the performance of the specialized implementations (Dbc), (PipeD), (PipeD4) and (PipeD4ls) in ST and SMT mode. In Section 5.2.3, we saw that the performance of the Pthreads implementations can nearly be doubled by executing two threads on each physical core. We observe a similar improvement for the MPI implementations. But while the job size of the Pthreads implementations was limited to one node, i.e., 32 threads in ST mode and 64 threads in SMT mode, MPI jobs can span multiple nodes. In our experiments with up to 256 processes (8 nodes/256 cores in ST mode, 4 nodes/128 cores in SMT mode), the performance of SMT jobs nearly matches the performance of the corresponding ST jobs with the same number of processes but twice the number of physical cores. That means, the performance our implementations gain from using SMT mode can be sustained for multi-node jobs.

5.3.4 SGI Altix 4700

Among the target systems considered, HLRB 2 has the largest number of processor cores. On this system we could investigate the scalability of our implementations with up to $n_B/2 = N/2$ processor cores, which is the maximum number of processes supported by our current implementations.[2]

On HLRB 2 the performance of the specialized implementations is not influenced by the direction of the iteration over the dimension of the ODE system. Due to their higher locality, the pipelining implementations clearly outperform (Dbc) if the number of processors is small. But as the number of processors grows, (Dbc) becomes more competitive in the experiments using RKF 2(3) or DOPRI 5(4) and may even outperform the pipelining implementations in some experiments because (Dbc) can overlap a larger part of the communication time and the share of the overall data set to be processed by the individual processors decreases reciprocally proportional with the number of processors p and, thus, locality aspects become less significant. However, if the number of processors is increased further such that $n_B/p - 2 \approx s$ the advantage of (Dbc) being able to overlap a larger part of the communication time is no longer applicable and the performance is no longer better than that of the pipelining implementations.

As on JUMP, (PipeD4) and (PipeD5) show similar execution times. The alternative finalization strategy of (PipeD2) is not successful on HLRB 2. But the other implementations obtain a high efficiency above 0.6 even for large numbers of processors. Using DOPRI 8(7) and $N = 2000$, speedups between 870 and 933 have been measured on 1000 processor cores. Steps observed in the speedup curves and the corresponding saw teeth observed in the efficiency curves reflect load imbalances resulting from the block-based blockwise data distribution. For example, when 896 MPI processes are started using $N = 2000$ as parameter, the data range of 208 processes consists of 3 blocks while the data range of 688 processors consists of only 2 blocks. In most experiments, the performance of the low-storage implementation (PipeD4ls) is slightly lower than that of the corresponding conventional implementation (PipeD4). The general implementation (D), which requires global communication, shows a similarly poor performance as on the other systems.

6 Conclusions

In this paper, we have proposed an implementation strategy for RK methods which is based on a pipelined processing of the stages of the method. This implementation strategy is suitable for ODE systems with limited access distance. It has been derived from the pipelining scheme presented in [29, 30, 31] by an overlapping of vectors. The implementation strategy proposed herein provides a higher locality of memory references and requires less storage space than our previous implementations and other implementations proposed in related works. If no stepsize control is required, only $n + \Theta\left(\frac{1}{2}s^2 d(\mathbf{f})\right)$ vector elements need to be stored. Efficient stepsize control based on an embedded solution is possible using only one additional n-vector. In contrast to low-storage

[2] Using $n_B/2$ processors, the data range assigned to each processor spans two blocks. It is possible to modify (Dbc) such that the data range of each processor can be reduced to one block, but no pipelining of the RK stages is possible with a data range of only one block since at least two pipeline stages are required to build a feasible pipeline.

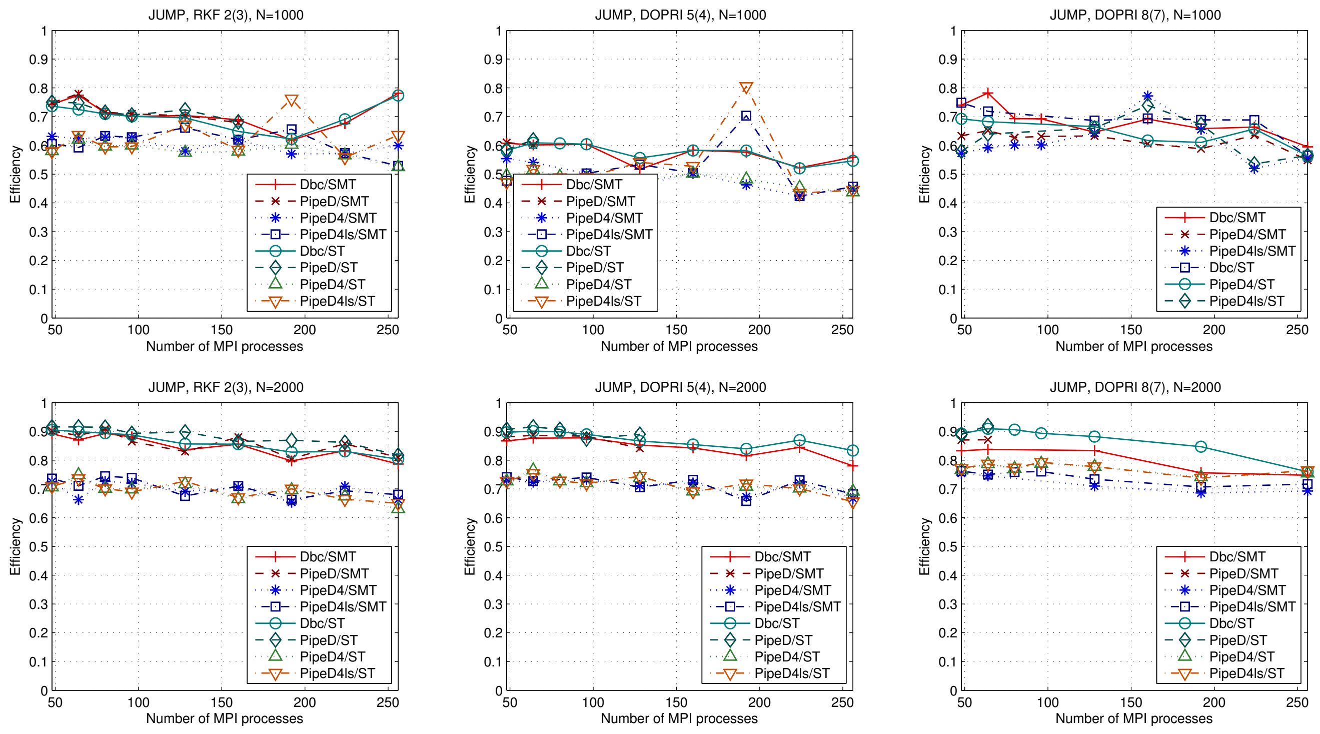

Fig. 15: Comparison of the parallel efficiency of the distributed-address-space implementations on the IBM p6 575 Cluster using SMT and ST mode.

RK algorithms proposed by other authors, the presented implementation strategy does not impose restrictions on the choice of coefficients of the RK method and can thus be used with popular embedded RK method such as the methods of Dormand and Prince, Verner, Fehlberg and others. Moreover, parallel implementations of the proposed strategy for shared as well as distributed address space have been described, and an experimental evaluation has shown a high scalability on five different parallel computer systems.

Download

The implementations investigated in this paper are available for download from our project website `http://www.ai2.uni-bayreuth.de/ode-adaptivity/`.

Acknowledgments

We thank the Jülich Supercomputing Centre and the Leibniz Supercomputing Centre Munich for providing access to their supercomputer systems. This work was supported by the German Research Foundation (DFG).

References

[1] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 10h Processors*, May 2009. Publication No. 40546, Revision 3.11.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach.* Morgan Kaufmann, 2002.

[3] W. Augustin, V. Heuveline, and J.-P. Weiss. Optimized stencil computation using in-place calculation on modern multicore systems. In *Euro-Par 2009. Parallel Processing*, LNCS 5704, pages 772–784. Springer, 2009.

[4] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[5] J. Berland, C. Bogey, and C. Bailly. Optimized explicit schemes: matching and boundary schemes, and 4th-order Runge–Kutta algorithm. In *10th AIAA/CEAS Aeroacoustics Conference, 10–12 May, Manchester, UK*, pages 1–34, 2004. AIAA Paper 2004-2814.

[6] J. Berland, C. Bogey, and C. Bailly. Low-dissipation and low-dispersion fourth-order Runge–Kutta algorithm. *Computers & Fluids*, 35(10):1459–1463, 2006.

[7] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *11th ACM Int. Conf. on Supercomputing*, 1997.

[8] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw. (TOMS)*, 28(2):135–151, 2002.

[9] R. W. Brankin, I. Gladwell, and L. F. Shampine. RKSUITE: A suite of Runge–Kutta codes for the initial value problem of ODEs. Softreport 92-S1, Department of Mathematics, Southern Methodist University, Dallas, Texas, USA, 1992.

[10] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations.* Oxford University Press, New York, 1995.

[11] D. R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[12] M. Calvo, J. M. Franco, and L. Rández. Short note: A new minimum storage Runge–Kutta scheme for computational acoustics. *J. Comp. Phys.*, 201(1):1–12, 2004.

[13] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. *Sci. Prog.*, 5:173–184, 1996.

[14] N. H. Cong and L. N. Xuan. Twostep-by-twostep PIRK-type PC methods with continuous output formulas. *J. Comput. Appl. Math.*, 221:165–173, 2008.

[15] P. Deuflhard. Recent progress in extrapolation methods for ordinary differential equations. *SIAM Rev.*, 27:505–535, 1985.

[16] J. R. Dormand and P. J. Prince. A family of embedded Runge–Kutta formulae. *J. Comput. Appl. Math.*, 6(1):19–26, 1980.

[17] R. Ehrig, U. Nowak, and P. Deuflhard. Massively parallel linearly-implicit extrapolation algorithms as a powerful tool in process simulation. In *Parallel Computing: Fundamentals, Applications and New Directions*, pages 517–524. Elsevier, 1998.

[18] E. Fehlberg. Classical fifth–, sixth–, seventh– and eighth order Runge–Kutta formulas with step size control. *Computing*, 4(2):93–106, 1969.

[19] K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Proc. of Supercomputing'99 Conference*, 1999.

[20] C. W. Gear and X. Hai. Parallelism in time for ODEs. *Appl. Numer. Math.*, 11:45–68, 1993.

[21] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems.* Springer, Berlin, 2nd rev. edition, 2000.

[22] T. E. Hull, W. H. Enright, and K. R. Jackson. User's guide for DVERK - A subroutine for solving non-stiff ODEs. Technical Report 100, Department of Computer Science, University of Toronto, 1976.

[23] K. R. Jackson and S. P. Nørsett. The potential for parallelism in Runge–Kutta methods. Part 1: RK formulas in standard form. *SIAM J. Numer. Anal.*, 32(1):49–82, Feb. 1995.

[24] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing temporal locality with skewing and recursive blocking. In *SC2001: High Performance Networking and Computing.* ACM Press and IEEE Computer Society Press, Nov. 2001. CD-ROM.

[25] M. Kandemir, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. *J. Par. Distr. Comp.*, 60:924–965, Aug. 2000.

[26] M. Kappeller, M. Kiehl, M. Perzl, and M. Lenke. Optimized extrapolation methods for parallel solution of IVPs on different computer architectures. *Applied Mathematics and Computation*, 77(2–3):301–315, July 1996.

[27] C. A. Kennedy and M. H. Carpenter. Third-order $2N$-storage Runge–Kutta schemes with error control. Technical Report NASA TM-109111, National Aeronautics and Space Administration, Langley Research Center, Hampton, VA, 1994.

[28] C. A. Kennedy, M. H. Carpenter, and R. M. Lewis. Low-storage, explicit Runge–Kutta schemes for the compressible Navier-Stokes equations. *Appl. Numer. Math.*, 35(3):177–219, 2000.

[29] M. Korch. *Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität.* Doctoral thesis, University of Bayreuth, Bayreuth, Germany, Dec. 2006.

[30] M. Korch and T. Rauber. Scalable parallel RK solvers for ODEs derived by the method of lines. In *Euro-Par 2003. Parallel Processing*, LNCS 2790, pages 830–839. Springer, Aug. 2003.

[31] M. Korch and T. Rauber. Optimizing locality and scalability of embedded Runge–Kutta solvers using block-based pipelining. *J. Par. Distr. Comp.*, 66(3):444–468, Mar. 2006.

[32] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures.* Doctoral thesis, Universität Erlangen-Nürnberg, Erlangen, Germany, July 2004.

[33] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 1:131–145, July 1982.

[34] R. Martí, V. Campos, and E. Piñana. A branch and bound algorithm for the matrix bandwidth minimization. *European Journal of Operational Research*, 186(2):513–528, 2008.

[35] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Trans. Par. Dist. Syst.*, 9(8):769–787, Aug. 1998.

[36] S. P. Nørsett and H. H. Simonsen. Aspects of parallel Runge–Kutta methods. In *Numerical Methods for Ordinary Differential Equations*, number 1386 in LNM, pages 103–117, 1989.

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

[38] T. Rauber and G. Rünger. Parallel execution of embedded and iterated Runge–Kutta methods. *Concurrency: Practice and Experience*, 11(7):367–385, 1999.

[39] T. Rauber and G. Rünger. Improving locality for ODE solvers by program transformations. *Sci. Prog.*, 12(3):133–154, 2004.

[40] S. J. Ruuth. Global optimization of explicit strong-stability-preserving Runge–Kutta methods. *Math. Comp.*, 75(253):183–207, 2005.

[41] W. E. Schiesser. *The Numerical Method of Lines.* Academic Press, Inc., 1991.

[42] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference.* MIT Press, Cambridge, Mass., 2nd edition, 1998.

[43] K. Strehmel and R. Weiner. *Numerik gewöhnlicher Differentialgleichungen.* Teubner, Stuttgart, 1995.

[44] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2009.

[45] P. J. van der Houwen and B. P. Sommeijer. Parallel iteration of high-order Runge–Kutta methods with stepsize control. *J. Comput. Appl. Math.*, 29:111–127, 1990.

[46] Q. Wang, Y. C. Guo, and X. W. Shi. An improved algorithm for matrix bandwidth and profile reduction in finite element analysis. *Progress In Electromagnetics Research Letters*, 9:29–38, 2009.

[47] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Par. Comp.*, 27(1–2):3–35, 2001.

[48] J. White, A. Sangiovanni-Vincentelli, F. Odeh, and A. Ruehli. Waveform relaxation; Theory and practice. *Transactions of the Society for Computer Simulation*, 2:95–133, 1985.

[49] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.