Proceedings of ALGORITMY 2012
pp. 250–260

EXPLOITING LIMITED ACCESS DISTANCE OF ODE SYSTEMS FOR PARALLELISM AND LOCALITY IN EXPLICIT METHODS*

MATTHIAS KORCH†

Abstract. The solution of initial value problems of large systems of ordinary differential equations (ODEs) is computationally intensive and demands for efficient parallel solution techniques that take into account the complex architectures of modern parallel computer systems. This article discusses implementation techniques suitable for ODE systems with a special coupling structure, called *limited access distance*, which typically arises from the discretization of systems of partial differential equations (PDEs) by the method of lines. It describes how these techniques can be applied to different explicit ODE methods, namely embedded Runge–Kutta (RK) methods, iterated RK methods, extrapolation methods, and Adams–Bashforth (AB) methods. Runtime experiments performed on parallel computer systems with different architectures show that these techniques can significantly improve runtime and scalability. By example of Euler's method it is demonstrated that these techniques can also be applied to devise high-performance GPU implementations.

Key words. ordinary differential equations, initial value problems, parallelism, locality

AMS subject classifications. 65L05, 65Y05, 65Y20

1. Introduction.

This article considers the parallel solution of initial value problems (IVPs) of a special class of systems of ordinary differential equations (ODEs). General first order ODE IVPs can be defined by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad t \in [t_0, t_e] \tag{1.1}$$

with the right-hand-side function $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, the wanted solution function $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$, its initial value $\mathbf{y}_0$, and the integration interval $[t_0, t_e]$. The typical solution procedure consists in a step-by-step integration of the right-hand-side function $\mathbf{f}$ walking through the integration interval $[t_0, t_e]$ using a sequence of time steps $\kappa = 1, 2, \ldots$ such that at each time step an approximation $\mathbf{y}_{\kappa+1} \approx \mathbf{y}(t_{\kappa+1})$ is computed. There exist many different sequential and parallel solution methods, which are distinguished by the computations performed at each time step. Some of the parallel methods proposed are iterated Runge–Kutta (iterated RK, IRK) methods [14, 19], extrapolation methods [6], and peer two-step methods [18]. One important classification criterion is whether the computation of the new approximation $\mathbf{y}_{\kappa+1}$ involves $\mathbf{y}_{\kappa+1}$ itself and, hence, a non-linear system of equations has to be solved. Such methods are called *implicit*. Methods which use only $\mathbf{y}_\kappa$ or preceding approximations are called *explicit*. IVPs which are only tractable by implicit methods are called *stiff*; IVPs which can be solved by explicit methods are called *non-stiff*.

Most of the methods proposed support an arbitrary coupling between the equations of the ODE system, i.e., an evaluation of the right-hand-side function $\mathbf{f}$ for a component j, $f_j(t, \mathbf{y})$ may access all components of the argument vector $\mathbf{y}$. However, many ODE systems possess a special coupling structure that can be exploited to optimize (parallel) performance. In particular, many large ODE systems are sparse, i.e.,

*This work was supported by DFG Grants No.: Ra 524/7-1, Ra 524/7-2, Ra 524/17-1, and Ra 524/17-2.

†University of Bayreuth, Applied Computer Science 2, 95440 Bayreuth, Germany (korch@uni-bayreuth.de).

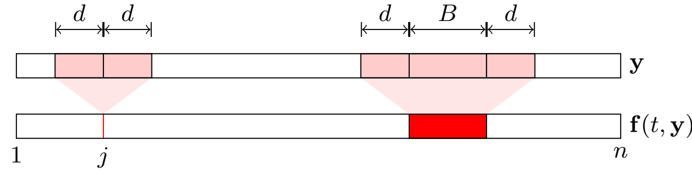

FIGURE 1.1. *Dependence pattern of ODE systems with limited access distance.*

$f_j(t, \mathbf{y})$ needs only a small number of components of the argument vector $\mathbf{y}$. This article considers the class of ODE systems with *limited access distance*, where $f_j(t, \mathbf{y})$ only uses components of $\mathbf{y}$ located nearby j. More precisely, the *access distance* $d(\mathbf{f})$ is the smallest value d, such that each component functions $f_j(t, \mathbf{y})$, $j = 1, \ldots, n$, accesses only the subset $\{y_{j-d}, \ldots, y_{j+d}\}$ of the components of the argument vector $\mathbf{y}$ (Fig. 1.1). The access distance of $\mathbf{f}$ is *limited* if $d(\mathbf{f}) \ll n$. Large ODE systems with limited access distance arise, for example, from the spatial semi-discretization of systems of partial differential equations (PDEs) by the method of lines.

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., [12, 20], 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.

This article discusses implementation techniques for explicit ODE methods which are applicable to ODE systems with limited access distance. Target architectures are modern shared- and distributed-memory parallel CPU-based computers, but also GPUs. The implementation techniques lead to a better utilization of the memory hierarchy and to lower communication overhead in parallel implementations, and thus to increased performance and scalability. As examples of explicit ODE methods, we consider embedded RK methods, iterated RK methods, extrapolation methods, and Adams–Bashforth (AB) methods on CPU-based parallel computers. All these implementations were developed using C as programming language and MPI or POSIX Threads (Pthreads) for parallelization. Euler's method is used as example to demonstrate the applicability of these techniques to GPUs. Detailed runtime experiments on several parallel systems with different architectures are shown to evaluate the resulting performance of the optimization techniques.

2. Optimized communication pattern. Parallel ODE methods can exploit three different types of parallelism or combinations of these types [4]. *Parallelism across the method* carries out calculations intrinsic to the method itself in parallel, e.g., independent stages as in [18]. Usually, the exploitation of parallelism across the method leads to only a small number of coarse-grained tasks. *Parallelism across time* (also called *parallelism across the steps*) refers to solving different parts of the time range at the same time. This type of parallelism can be exploited on a small scale by overlapping adjacent time steps as proposed, e.g., by Miranker and Lininger [13]. But it can also be exploited for massive parallelism [1]. In this article, we focus on *parallelism across the system*, which can naturally be exploited by any ODE method by distributing the equations of the ODE system to different processor cores. The precondition for this kind of parallelization is that the dimension of the ODE system

is large enough to provide a sufficient amount of work for each processor core.

General implementations of ODE methods, i.e., implementations suitable for ODE systems with arbitrary coupling, must provide to each evaluation of a component function $f_j(t, \mathbf{y})$ the entire argument vector $\mathbf{y}$. Therefore, in a system-parallel execution, all processor cores must have access to the entire argument vector or a copy of it. In case of a distributed address space, this requires a replicated storage of the argument vector, where each copy of the argument vector is automatically located in the local memory of the corresponding processor core. Typically, each processor core computes a subset, usually a block of the components of an argument vector, and the final replicated argument vector (i.e., all its copies) has to be assembled from the local subsets by a multibroadcast operation (e.g., `MPI_Allgatherv()`).

In case of a shared address space, we can choose between replicated and shared storage of the argument vector. While a replicated storage requires explicit memory copy operations which act as a multibroadcast operation, a shared storage allows that the function evaluations directly access the shared argument vector. To understand the possible impact on performance of the two storage strategies, we have to consider the physical layout of the memory. Most of today's shared-address-space computers have a NUMA (*non-uniform memory access*) architecture with physically distributed memory (e.g., each multi-core CPU has a locally attached memory), and memory pages are allocated by a *first touch policy* (a page is allocated in the local memory of the multi-core CPU that first writes to it). Consequently, copies of replicated vectors as well as the locally computed blocks of components of a shared vector can be located in the local memories of the corresponding processor cores. However, for a dense ODE system, either of the two storage strategies does involve a large number of expensive remote memory accesses.

If the ODE system has a limited access distance, we can exploit this property to apply a completely local storage strategy with scalable neighbor-to-neighbor communication involving only a small number of vector components as it is used in many domain decomposition approaches (Fig. 2.1). Each of the p processor cores stores locally its block of n/p components of the argument vector and, additionally, $d(\mathbf{f})$ components at each of the two borders of its range of components, which are copied from the adjacent neighbor processor core (*ghost cells*). The copying of these components can be realized by single transfer operations, and often it can be overlapped with computations (e.g., using the non-blocking operations `MPI_Isend()` and `MPI_Irecv()`). Due to the NUMA architecture of modern shared-address-space computers, this storage strategy is favorable also in case of a shared address space. Even though, in case of a shared address space, a shared storage would avoid the need for explicitly copying the components within the access distance $d(\mathbf{f})$ from the neighbor processor cores, these components would nevertheless be accessed by the function evaluations and thus be implicitly transferred to the neighboring processor cores. Moreover, page sharing may occur at adjacent borders and adversely affect the performance if the distribution of the components to the processor cores is not aligned at page granularity.

3. Pipelining and diamond-like tiling schemes. The application of the optimized communication pattern described in the previous section already leads to a high scalability since it replaces expensive global communication by local neighbor-to-neighbor communication so that the resulting communication costs are independent of the number of participating processor cores. For highest parallel efficiency, it is however necessary to also optimize the per-core performance.

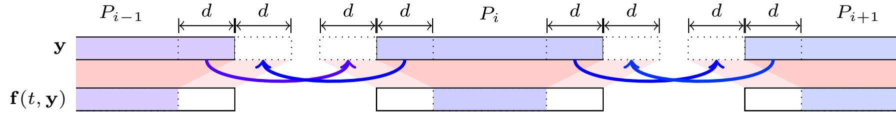

FIGURE 2.1. *Optimized communication pattern for ODE systems with limited access distance.*

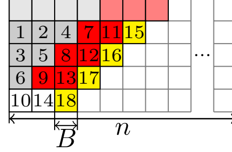

FIGURE 3.1. *Pipeline computation order.*

The per-core performance of data-intensive applications such as ODE methods applied to ODE systems of large dimension mainly depends on how well the memory hierarchy can be utilized to reduce the number of expensive memory accesses. A standard approach for improving cache efficiency is the use of *loop tiling* to break down the working spaces of loops iterating over large vectors or matrices to the size(s) of the cache(s). While general implementations often already allow some form of loop tiling, the assumption that a component function $f_j(t, \mathbf{y})$ may access all components of the argument vector $\mathbf{y}$ prevents that all loops across the system dimension can be fused and tiled, because all n components of $\mathbf{y}$ have to be computed in a loop before the function evaluations $f_j(t, \mathbf{y})$, $j = 1, \ldots, n$, can be performed in another loop. Since for large ODE systems the storage space of a n-vector is larger than the cache size, argument vector components and function results cannot be reused as desirable.

Often, more advanced loop transformations can be applied if the ODE system has a limited access distance. All these transformations are based on the observation that a function evaluation $f_j(t, \mathbf{y})$ can be started as soon as the $2d(\mathbf{f}) + 1$ components of $\mathbf{y}$ within the access distance of $f_j(t, \mathbf{y})$ have been computed. Assuming a block-based subdivision into $n_B = \lceil n/B \rceil$ blocks of size $B \geq d(\mathbf{f})$, the function evaluation of a block $J \in \{1, \ldots, n_B\}$ defined by

$$\mathbf{f}_J(t, \mathbf{y}) = (f_{(J-1)B+1}(t, \mathbf{y}), f_{(J-1)B+2}(t, \mathbf{y}), \ldots, f_{(J-1)B+\min\{B, n-(J-1)B\}}(t, \mathbf{y})) \tag{3.1}$$

can be started as soon as the blocks $J - 1$, J, and $J + 1$ of $\mathbf{y}$ are available.

For a sequence of steps consisting of the computation of an argument vector $\mathbf{y}$ and a function evaluation $\mathbf{f}(t, \mathbf{y})$, this block-based dependence pattern allows a reorganization of the loop structure such that the loop over the system dimension becomes the outermost loop by delaying the computation of the steps of the sequence by one block. This results in the pipeline-like computation scheme illustrated in Fig. 3.1. The advantage of this scheme is that it can process several steps of the sequence in a single sweep, thus creating a small working space that consists of the vector components accessed by computing one pipeline diagonal. If this working space is small enough to fit in the cache, this leads to a reuse of vector elements between successive pipeline diagonals and thus to a high cache efficiency.

For long sequences of steps, it may not be efficient to span the pipeline across

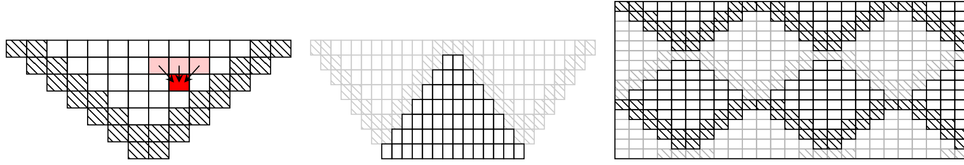

FIGURE 3.2. *Diamond-like tiling.*

all steps, because the resulting working space would exceed the cache size. Instead, one can span the pipeline across only a small number of steps and perform several sweeps across the system dimension. An even better performance might be obtained by a modification of the pipelining scheme resulting in diamond-like tile shapes as illustrated in Fig. 3.2, which have been proposed in [15, 16] to partition the two-dimensional problem domain of a finite-difference time-domain (FDTD) application.

4. Embedded RK methods. The computation scheme of one time step of an s-stage embedded RK method with method coefficients a_{li}, c_i, and b_l is given by

$$\begin{aligned} \mathbf{w}_l &= \mathbf{y}_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{w}_i), \quad l = 1, \ldots, s, \\ \mathbf{y}_{\kappa+1} &= \mathbf{y}_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{v}_l, \qquad \hat{\mathbf{y}}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{l=1}^{s} \hat{b}_l \mathbf{v}_l \ . \end{aligned} \tag{4.1}$$

The second, *embedded* approximation, $\hat{\mathbf{y}}_{\kappa+1} \approx \mathbf{y}(t_\kappa)$, provides an efficient means to estimate the local error for stepsize control.

In the general case, an implementation of the classical RK scheme needs to hold at least $s+1$ vectors of size n (so called *registers*) to store $\mathbf{y}_\kappa$, $\mathbf{w}_2, \ldots, \mathbf{w}_s$, and $\mathbf{y}_{\kappa+1}$, where n is the dimension of the ODE system. One additional register ($\hat{\mathbf{y}}_{\kappa+1}$ or an error estimate) is required for the implementation of a stepsize controller which can reject and repeat steps.

System-parallel implementations of embedded RK methods, which make no assumptions about the method coefficients or the coupling of the ODE system, have to compute the stages $l = 1, \ldots, s$ one after the other and have to exchange the current argument vector $\mathbf{w}_l$ between all participating processors at every stage. The scalability of general implementations is therefore often not satisfactory.

For ODE systems with limited access distance, we can apply the optimizations suggested in Sections 2 and 3 to the stage computations (cf. [10]). It is possible to replace the global multibroadcast operation to exchange $\mathbf{w}_l$ between the stages by neighbor-to-neighbor communication, and the sequence of stages can be processed using the pipelining scheme. Since the number of stages is usually small ($s = 13$ for DOPRI 8(7), often $s < 10$), the pipeline can in most scenarios be spanned across all s stages. The resulting working space illustrated in Fig. 4.1 (left) has a size of $\Theta(\frac{1}{2}s^2B)$. By overlapping the storage locations of the vectors (Fig. 4.1 (right)), the storage space can be reduced to $2n + \Theta(\frac{1}{2}s^2B)$. While other approaches to reduce the storage space to $2n$ or $3n$ such as [2, 3, 5, 8, 17] require special method coefficients and only some of these methods provide embedded solutions, the pipelining scheme does not impose restrictions on the choice of coefficients of the RK method and can thus be used with popular embedded RK methods such as the methods of Dormand and Prince, Verner, Fehlberg and others.

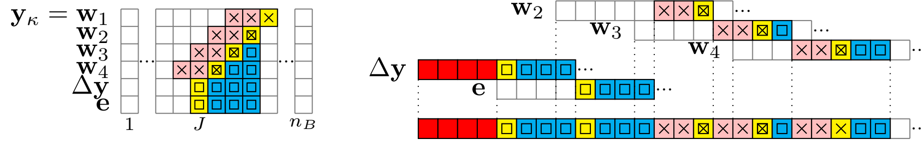

FIGURE 4.1. *Left: Illustration of the working space of one pipelining step of an embedded RK method. Blocks required for function evaluations are marked by crosses. Blocks updated using results of function evaluations are marked by squares. Right: Illustration of the overlapping of the vectors* $\Delta\mathbf{y}$, $\mathbf{e}$ *and* $\mathbf{w}_2, \dots, \mathbf{w}_s$ *in the low-storage implementation.*

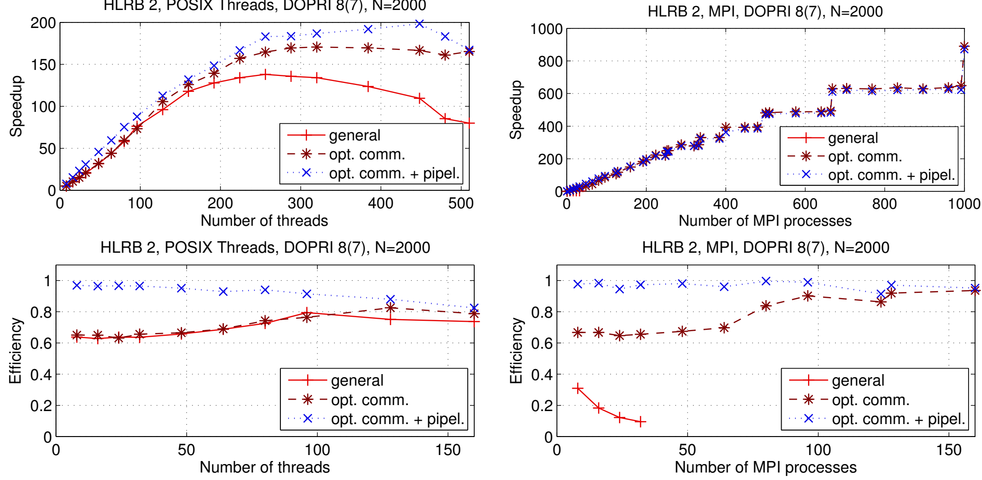

FIGURE 4.2. *Scalability of embedded RK implementation variants.*

Runtime experiments to evaluate the performance of shared- and distributed-address-space implementations of embedded RK methods optimized for limited access distance have been performed on several parallel computer systems with different architectures. In Fig. 4.2 we show parallel speedups and efficiency obtained on HLRB 2 at LRZ Munich. HLRB 2 is an SGI Altix 4700 system equipped with 9728 Intel Itanium 2 Montecito processors at 1.6 GHz. The test problem is BRUSS2D-MIX [4], a typical example of a PDE discretized by the method of lines, with limited access distance $d(\mathbf{f}) = 2N$ and system size $n = 2N^2$. As RK method, DOPRI 8(7) was used.

While the general Pthreads implementation can obtain reasonable speedups, the general MPI implementation does not scale due to the expensive `MPI_Allgatherv()` operation. Speedups and efficiency of the MPI implementations with optimized communication are very high. Pipelining improves the performance in particular for small numbers of processor cores, because for large numbers of processor cores the total amount of data processed per time step per processor core fits in the cache.

5. Iterated RK Methods. Based on the classical implicit RK methods, explicit IRK methods suitable for non-stiff equations introduce the iteration process

$$\mathbf{Y}_l^{(k)} = \mathbf{y}_\kappa + h_\kappa \sum_{i=1}^{s} a_{li}\mathbf{F}_i^{(k-1)}, \quad \mathbf{F}_i = \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{Y}_i), \quad l = 1, \dots, s, \quad k = 1, \dots, m. \tag{5.1}$$

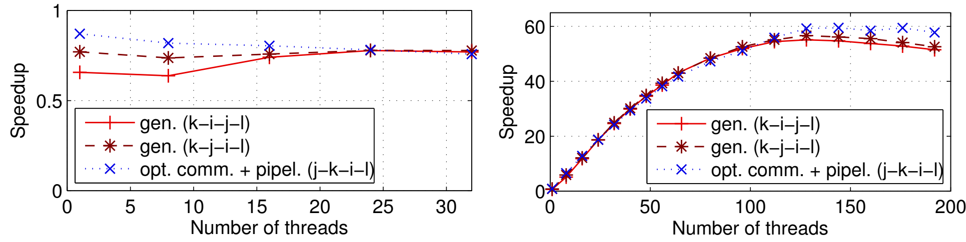

FIGURE 5.1. *Scalability of shared-address-space IRK implementations variants.*

We choose the 'trivial' predictor $\mathbf{Y}_l^{(0)} = \mathbf{y}_\kappa, l = 1, \ldots, s$, to start the iteration process and execute a fixed number of $m = p - 1$ corrector steps (5.1), where p is the order of the underlying implicit RK method (cf. [19]). Two approximations to $\mathbf{y}(t_{\kappa+1})$ of different order, $\mathbf{y}_{\kappa+1}$ and $\hat{\mathbf{y}}_{\kappa+1}$, are then computed by

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{F}_l^{(m)} \quad \text{and} \quad \hat{\mathbf{y}}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{F}_l^{(m-1)}. \tag{5.2}$$

In contrast to embedded RK methods, the stages of IRK methods can be computed in parallel. But here we focus on purely system-parallel implementations.

For general system-parallel implementations, the i-loop, the l-loop, and the j-loop across the system dimension, are fully permutable, i.e., they can be interchanged and loop tiling can be applied. But the m corrector steps have to be computed sequentially, and the s vectors $\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}$ have to be exchanged between the corrector steps.

If the access distance of the ODE system is limited, we can switch to scalable neighbor-to-neighbor communication and process the corrector steps in pipeline order. Fig. 5.1 shows speedups obtained by shared-address-space IRK implementations on HLRB 2 for the test problem BRUS2D-MIX with $N = 1000$ using Radau IA(5) as base method. The pipelining implementation with optimized communication outperforms the general implementations for small numbers of processor cores, because the working space of a pipelining step fits in the cache while the working space of a time step exceeds the cache size. It also delivers the best performance if the number of processor cores is large, because in this situation the optimized communication pattern with distributed storage is more efficient than the barrier synchronization combined with shared storage used in the general implementations.

6. Extrapolation methods. At each time step κ of an extrapolation method, r independent approximations $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ to $\mathbf{y}(t_{\kappa+1})$ of increasing accuracy are computed. Then, based on these approximations, $\mathbf{y}_{\kappa+1}$ is extrapolated using the Aitken–Neville algorithm [7]. To compute the r approximations, the IVP $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$, $\mathbf{y}_0 = \mathbf{y}_\kappa$ is solved r times on the same interval $[t_\kappa, t_{\kappa+1}]$ by a base method computing sequences of *micro-steps* with different constant stepsizes $h_\kappa^{(1)} > h_\kappa^{(2)} > \cdots > h_\kappa^{(r)}$. Here, we use the explicit Euler method as base method.

The micro-step sequences can be computed in parallel on different groups of processor cores. If the ODE system has a limited access distance, a micro-step sequence computed by a group of processor cores can be processed using the pipelining scheme and the optimized communication pattern can be applied. Fig. 6.1 shows speedups and efficiency of different shared- and distributed-address-space implementation vari-

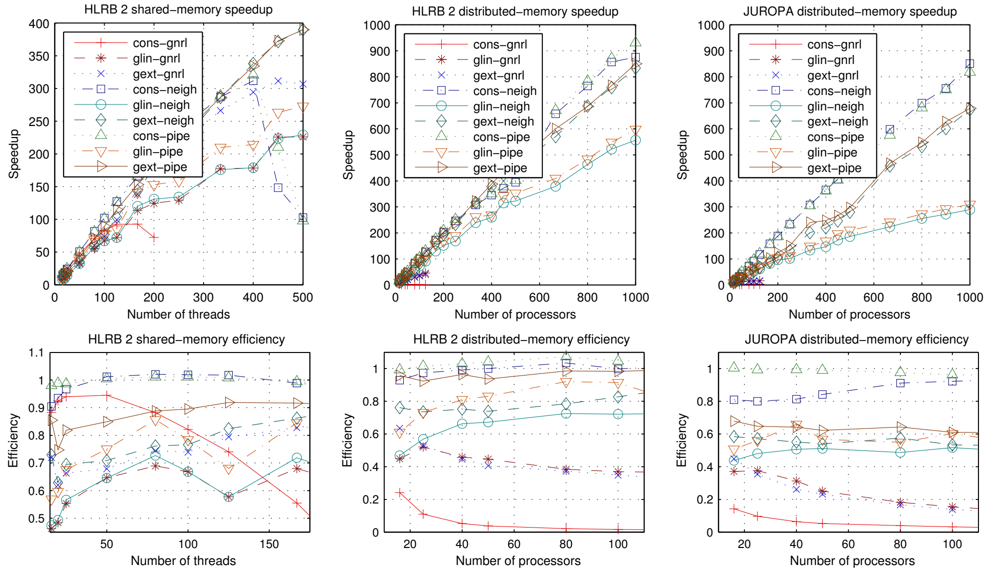

FIGURE 6.1. *Scalability of implementation variants of extrapolation methods.*

ants on HLRB 2 and JUROPA using $r = 16$ and the test problem BRUSS2D with $N = 2000$. JUROPA, operated by the Jülich Supercomputing Centre (JSC), consists of 2 208 nodes equipped with two quad-core Intel Xeon X5570 (Nehalem-EP) processors running at 2.93 GHz. For a detailed description of the implementation variants see [11]. As for embedded RK methods, we observe that only the MPI implementations with the optimized communication pattern (`*-neigh` and `*-pipe`) are scalable. The use of the pipelining scheme is most successful for small numbers of processor cores, where the total working space of a micro-step does not fit in the cache.

7. Adams-Bashforth methods. AB methods defined by

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^{k} \beta_l \mathbf{f}(t_{\kappa-l+1}, \mathbf{y}_{\kappa-l+1}) \tag{7.1}$$

belong to the class of explicit linear k-step methods. If a constant stepsize is used for all time steps, a limited access distance of the ODE system can be exploited to process the time steps in pipeline order and to apply the optimized communication pattern. Since the number of time steps can be very large, the pipeline should not be spanned across all time steps. Instead, we introduce an implementation parameter which determines the pipeline length. Runtime experiments on HLRB 2 and JUROPA (Fig. 7.1) confirm that the optimized communication pattern is needed to obtain scalable MPI implementations. Also, pipelining outperforms the other implementations for small numbers of processor cores where the working space of a pipelining step fits in the cache, but the working space of a time step is larger than the cache.

8. GPU implementation of the explicit Euler method. As a first step to investigate the potential scalability of ODE solvers on GPUs, we investigate OpenCL implementations of the explicit Euler method

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \cdot \mathbf{f}(t_\kappa, \mathbf{y}_\kappa) \tag{8.1}$$

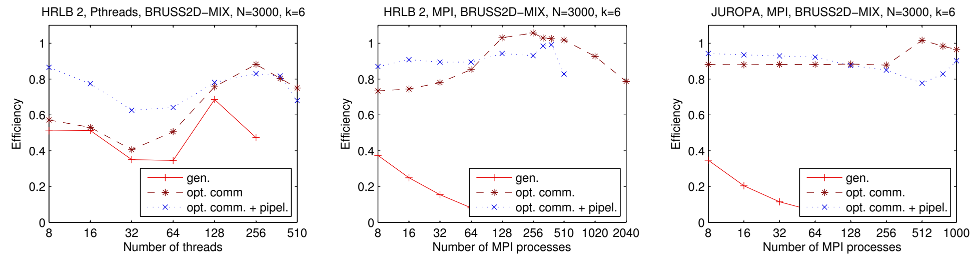

FIGURE 7.1. *Scalability of AB implementation variants.*

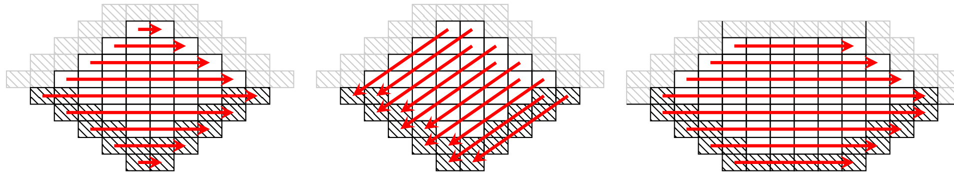

FIGURE 8.1. *From left to right: row-diamond, diagonal-diamond, and hexagon tiling scheme.*

with constant stepsize. For best performance on a GPU it is crucial to utilize the small but fast local memories of the multiprocessors of the GPU, in particular if the GPU has no cache. For ODE systems with limited access distance, we investigate different diamond-like shapes of tiles (Fig. 8.1,) where the computation of a tile can be performed completely in local memory (cf. [9]).

Runtime experiments on an NVIDIA GTX 580 GPU with BRUSS2D-MIX and a second test problem (vibrating string [7]), a discretized 1D PDE with access distance $d(\mathbf{f}) = 3$, show that it depends on the ODE system if diamond tiling leads to better performance than a general implementation (Fig. 8.2). For BRUSS2D-MIX with a large access distance of $d(\mathbf{f}) = 2N$, the general implementation is most efficient, because the L2 cache of the GTX 580 reduces the benefit of the use of local memories and the computations of the diamond tiling implementations are not perfectly balanced. For the vibrating string test problem, the diamond tiling implementations can outperform the general implementation, because the smaller access distance enables a more efficient utilization of the local memories. The hexagon implementation can often deliver the best performance, in particular for large system sizes.

9. Conclusions. Optimization techniques for explicit parallel ODE solvers specialized in ODE systems with limited access distance have been presented and discussed. If the ODE system has a limited access distance, scalable neighbor-to-neighbor communication can be used and cache efficiency can be increased for sequences of steps consisting of the computation of an argument vector and a function evaluation by processing the steps in a pipeline- or diamond-like fashion. These techniques have been applied to embedded and iterated RK methods, extrapolation methods and AB methods for CPU-based parallel computer systems. Additionally, GPU implementations of the explicit Euler method with diamond-like tile shapes have been discussed. Runtime experiments have shown that the techniques described can significantly improve the performance of the methods considered on different architectures.

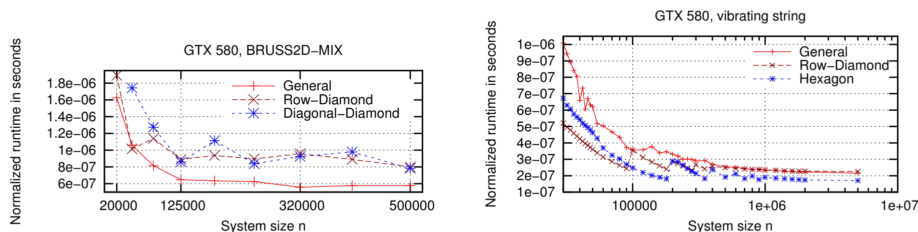

FIGURE 8.2. *Normalized runtime of explicit Euler implementations on a GPU.*

Acknowledgments. We thank the Jülich Supercomputing Centre (JSC) and the Leibniz Computing Centre (LRZ) Munich for providing access to their supercomputer systems JUROPA and HLRB 2.

REFERENCES

[1] P. Amodio and L. Brugnano, *Parallel solution in time of ODEs: some achievements and perspectives*, Appl. Numer. Math., 59 (2009), pp. 424–435. Selected papers NUMDIFF-11.

[2] 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, 2004, pp. 1–34. AIAA Paper 2004-2814.

[3] ———, *Low-dissipation and low-dispersion fourth-order Runge–Kutta algorithm*, Computers & Fluids, 35 (2006), pp. 1459–1463.

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

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

[6] 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, Elsevier, 1998, pp. 517–524.

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

[8] 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 (2000), pp. 177–219.

[9] M. Korch, J. Kulbe, and C. Scholtes, *Diamond-like tiling schemes for efficient explicit euler on GPUs*, in The 11th Int. Symp. on Parallel and Distributed Computing (ISPDC 2012), IEEE, June 2012.

[10] M. Korch and T. Rauber, *Parallel low-storage Runge-Kutta solvers for ODE systems with limited access distance*, Int. J. High Perf. Comput. Appl., 25 (2011), pp. 236–255.

[11] M. Korch, T. Rauber, and C. Scholtes, *Scalability and locality of extrapolation methods on large parallel systems*, Concurrency Computat.: Pract. Exper., 23 (2011), pp. 1789–1815.

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

[13] W. L. Miranker and W. Liniger, *Parallel methods for the numerical integration of ordinary differential equations*, Mathematics of Computation, 21 (1967), pp. 303–320.

[14] S. P. Nørsett and H. H. Simonsen, *Aspects of parallel Runge–Kutta methods*, in Numerical Methods for Ordinary Differential Equations, no. 1386 in LNM, 1989, pp. 103–117.

[15] D. Orozco and G. Gao, *Mapping the FDTD application to many-core chip architectures*, in The 38th Int. Conf. on Parallel Processing (ICPP-2009), IEEE, 2009.

[16] D. Orozco, E. Garcia, and G. Gao, *Locality optimization of stencil applications using data dependency graphs*, in Proceedings of the 23rd Int. Conf. on Languages and Compilers for Parallel Computing (LCPC'10), Berlin, Heidelberg, 2011, Springer, pp. 77–91.

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

[18] B. A. Schmitt, R. Weiner, and S. Jebens, *Parameter optimization for explicit parallel peer*

two-step methods, Appl. Numer. Math., 59 (2008), pp. 769–782.

[19] 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 (1990), pp. 111–127.

[20] 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 (2009), pp. 29–38.