Proceedings of the Federated Conference on Computer Science and Information Systems pp. 685–694
DOI: 10.15439/2016F464
ACSIS, Vol. 8. ISSN 2300-5963

Influence of Locality on the Scalability of Method- and System-Parallel Explicit Peer Methods

Matthias Korch,
Thomas Rauber,
Matthias Stachowski
and Tim Werner
Department of Computer Science
University of Bayreuth
Email: {korch, rauber, matthias.stachowski, tim.werner}@uni-bayreuth.de

***Abstract*—Because the numerical solution of initial value problems (IVPs) of systems of ordinary differential equations (ODEs) can be computationally intensive, several parallel methods have been proposed in the past. One class of modern parallel IVP methods are the peer methods proposed by Schmitt and Weiner, some of which are publicly available in the software package EPPEER released in 2012. Since they possess eight independent stages, these methods offer natural parallelism across the method suitable for the typical numbers of CPU cores in modern multicore workstations. EPPEER is written in FORTRAN95 and uses OpenMP as parallel programming model.**

In this paper, we investigate the influence of the locality of memory references on the scalability of method- and system-parallel explicit peer methods. In particular, we investigate the interplay between the linear combination of the stages and the function evaluations by applying different program transformations to the loop structure and by evaluating their performance in detailed runtime experiments. These experiments point out that loop tiling is required to improve cache utilization while still allowing the compiler to vectorize along the system dimension.

To show that for certain classes of right-hand-side functions a stage-parallel execution is not optimal, and to enhance the scalability of the peer methods to core numbers larger than the number of stages of a method, system-parallel implementations have been derived. Runtime experiments show that there are IVPs for which these new implementations outperform stage-parallel implementations on numbers of cores less than or equal to the number of stages. Moreover, by exploiting the ability to utilize higher core numbers, higher speedups than the number of stages have been reached.

I. Introduction

THIS paper considers a class of parallel solution methods for initial value problems (IVPs) of systems of ordinary differential equations (ODEs), defined by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad t \in [t_0, t_e], \qquad (1)$$

where $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is the *right-hand-side function* defining the ODE system, $t \in \mathbb{R}$ is the *independent variable*, usually denoting "*time*", $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is the *solution function* to be computed within the *integration interval* $[t_0, t_e] \subset \mathbb{R}$, and $\mathbf{y}_0$ is the initial value, i.e., the value of $\mathbf{y}$ at time t_0.

Many ODE IVPs do not have an analytical solution and must be solved numerically. The classical numerical approach, which is also used by the class of methods considered in this paper, applies a time-stepping procedure that starts at t_0 and walks through the integration interval, computing a new approximation value $\mathbf{y}_m \approx \mathbf{y}(t_m)$ at each time step $m = 1, 2, \ldots$ until t_e is reached. A detailed treatment of the subject can be found in [1].

Because the numerical solution of ODE IVPs often is computationally intensive, several methods with potential for parallelism have been proposed. Usually, these methods are classified as exploiting *parallelism across the method*, *parallelism across the system*, and *parallelism across time* (also called *parallelism across the steps*), see [2] for an overview of classical parallel ODE methods.

This paper considers the explicit parallel two-step peer methods provided as part of the EPPEER software package [3]. Peer methods have been introduced by Schmitt and Weiner in 2004 [4]. The explicit methods included in EPPEER, which has been released in 2012, are described in [5], [6], [7]. These methods possess up to 8 independent stages, which can be computed in parallel on different cores. Hence, they are an example for methods providing parallelism across the method, which can be exploited without a (possibly difficult) parallelisation of the right-hand-side function. However, as the authors of EPPEER note, additional parallelisation across the system is possible for larger numbers of cores, but this functionality is currently not part of the EPPEER package and has not been investigated yet.

In their tests of the parallel performance of the explicit two-step peer methods, the authors of EPPEER notice that near-optimal speedups are possible for expensive right-hand-side functions, while the speedups observed for cheap right-hand-side functions were less satisfactory [6], [8]. They attribute this behavior to a part of the time step where a linear combination of vectors is computed [6], [8].

In this paper, we investigate the reasons for the lower performance of cheap right-hand-side functions in detail and identify the locality of memory references of the linear combination but also the interplay between the locality of the linear combination and the locality of the function evaluations as the main performance limiters. As part of this investigation, we apply several different program transformations to the loop structure of the peer methods and evaluate their effect on locality and scalability using runtime experiments on two

978-83-60810-90-3/$25.00

different hardware architectures. In particular, we make use of loop tiling to exploit both temporal and spatial locality while still enabling the compiler to vectorize the loops over the system dimension. In addition to the parallelization over the stages, we also investigate system-parallel implementations. The goal of this is not only to enable the use of larger numbers of processor cores, but primarily to compare the locality behavior and the scalability of a system-parallel and a stage-parallel execution on small numbers of cores. In particular, we are interested in the question whether a system-parallel or a stage-parallel execution is more efficient for IVPs with a cheap right-hand-side function.

The rest of the paper is structured as follows: Section II discusses related work. Section III describes the mathematical structure of peer methods. Section IV investigates the influence of the loop structure on locality and scalability. Section V considers the interplay of the function evaluations and the system-parallel execution. The last two sections conclude the article.

II. Background and Related Work

A. Parallel ODE Methods

The numerical solution of ODE IVPs can be computationally intensive. Therefore, many solution methods with potential for parallelism have been proposed. Most of the fundamental work on parallel ODE methods dates back to the 1980s and 1990s; an overview and further references can be found in [2].

Parallelism across time is generally difficult to obtain for IVPs because there needs to be an information flow from the initial value at t_0 to the end of the integration interval t_e. There are, however, promising approaches based on the Picard iteration, e.g., Parareal methods [9]. Most of the parallel ODE methods proposed concentrate on *parallelism across the method*, i.e., they provide a small number of independent coarse-grained computational tasks inherent in the computational structure of the method, for example, independent stages. Examples are Parallel Adams–Bashforth (PAB) and Parallel Adams–Moulton (PAM) methods [10], which belong to the class of *general linear methods* [1], parallel iterated RK (PIRK) methods [2], and extrapolation methods [1], [2]. Basically, all IVP methods possess a natural potential for *parallelism across the ODE system*, because usually the equations of the ODE system can be distributed to different processor cores. Of course, this approach is only feasible for reasonably large ODE systems.

One implementation strategy aiming at parallelism across the system is the use of parallel linear algebra operations which are parallelized along the system dimension. The project Odeint [11], for example, which is part of the Boost C++ library, contains several IVP methods and allows the use of different sequential or parallel state vector types. Another example of this type of parallelism is the PETSc library [12], which targets partial differential equations (PDEs), but also contains several ODE IVP solvers. One disadvantage of this approach is that parts of the code outside the linear algebra operations are not parallelized, and parallel performance tuning cannot be applied to the IVP solver as a whole.

System-parallel implementations covering all parts of a time step have been investigated in [13] for embedded RK methods and in [14] for PIRK methods.

Waveform relaxation methods [2] are specially tailored for a system-parallel execution of large ODE systems as they arise in the simulation of electrical circuits. They use the Picard iteration to decouple the equations of the ODE system and thus avoid synchronization and communication between cores. However, similar to Parareal methods, they have to deal with a possibly slow convergence of the Picard iteration.

The subject of this paper are explicit parallel two-step peer methods, which show similarities in their computational structure to PIRK methods and Parallel Adams methods. It investigates the locality and scalability of these methods when exploiting either method or system parallelism.

B. Performance Optimization

Computer systems are becoming more and more complex and make use of parallelism at various levels. To hide the increasing complexity, modern CPUs and compilers contain many mechanisms and techniques that aim at providing most of the available performance of the hardware in a transparent way to the application programmer. For example, CPUs contain multiple execution units, which can work in parallel, and they use dynamic instruction scheduling and *simultaneous multithreading* (SMT) to increase the utilization of the execution units. Modern compilers try to automatically unroll and vectorize loops to make use of SIMD (*single instruction multiple data*) extensions such as SSE and AVX [15].

Practically all modern CPUs used in high-performance computing possess a deep memory hierarchy with usually two or three levels of cache to temporarily store and reuse data read from or written to the external, slower main memory, thus hiding most part of the latency time that would otherwise be needed to access the main memory. In multi-core CPUs, the higher cache levels are often shared between several cores. To obtain maximum performance on a hardware platform with memory hierarchy, it is crucial to optimize the locality of memory references.

In the field of compiler design, efficient use of the memory hierarchy is tried to be achieved by reordering compute intensive loops in the source code. Loop tiling is considered as one of the most successful techniques. An important analytical model are *cache miss equations* [16], which can be used to estimate the effects of loop transformations. To analyze the dependencies of loops and to perform loop transformations, often the polyhedral model is used, e.g., [17]. A simple, but insightful visual model for the performance of computer programs on modern multi-core architectures is the *roofline model* [18]. A generalization of the roofline model to cover deep memory hierarchies is the *execution cache memory (ECM) model* [19].

To overcome the limits of static code analysis, some compiler-based approaches introduce source code annotations

[15] or propose domain specific languages (DSLs), which are then extended by *autotuning* techniques, e.g., [20]. Autotuning tries to determine the best configuration from a search space of possible code variants and parameters (e.g., block sizes for loop tiling, loop unroll factors, or number of threads). In [21], an online autotuning approach for system-parallel PIRK methods has been investigated.

III. Explicit Parallel Peer Methods in EPPEER

In this section, we will explain the computational structure of the explicit parallel two-step peer methods and describe their implementation in EPPEER. Then, we will investigate locality and scalability of the unmodified EPPEER code by several runtime experiments.

A. Computational Structure

Similar to classical RK methods, the explicit two-step peer methods in EPPEER use a time-stepping procedure to solve the IVP and compute s stages $\mathbf{Y}_{m,i}$, $i = 1, \ldots, s$, at each time step from t_{m-1} to t_m with $t_m = t_{m-1} + h_{m-1}$ for $m = 1, 2, \ldots$ [7]. However, in contrast to classical RK methods, all s stages have the same accuracy and stability properties and, to compute the stages, the s stages and the s function values of the previous time step are used:

$$\mathbf{Y}_{m,i} = \underbrace{\sum_{j=1}^{s} b_{ij} \mathbf{Y}_{m-1,j}}_{\mathbf{S_Y}} + \underbrace{h_m \sum_{j=1}^{s} a_{ij} \mathbf{f}(t_{m-1,j}, \mathbf{Y}_{m-1,j})}_{\mathbf{S_f}}, \quad (2)$$

where $t_{m-1,j} = t_{m-1} + h_{m-1} c_j$ for $j = 1, \ldots, s$, and a_{ij}, b_{ij}, and c_j, $i, j = 1, \ldots, s$ with $c_s = 1$ are the coefficients of the particular peer method. Therefore, these methods are a subclass of general linear methods (GLMs) [1].

Because only values from the previous time step are used, the s stages of the current time step do not depend on each other and can be computed in parallel on different cores, thus exhibiting parallelism across the method. In EPPEER, three methods with $s = 4, 6, 8$ are included which can use up to s cores. These methods have been introduced in [6]. In addition, four FSAL methods (first same as last: last stage of the previous time step is reused as the first stage of the current time step) with $s = 3, 5, 7, 9$, which have been introduced in [5], are included, which can use up to $s - 1$ cores.

Since all s stages have the same properties, in particular the same order $O(h_m^s)$, there is no determined value for $\mathbf{y}_m \approx \mathbf{y}(t_m)$. Instead, any stage value can be used as new approximation to the solution function.

Due to the two-step character, a starting procedure is required to generate $s-1$ stages in addition to the initial value $\mathbf{y}_0$ before the first time step can be performed with the peer method to compute $\mathbf{Y}_{1,i}$. Currently, EPPEER uses an explicit RK method (DOPRI 5(4)) to compute the start values. However, a parallel starting procedure has been proposed in [22].

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO PRIVATE(stg,ic) SCHEDULE(STATIC)
     do stg = ist1,stages
       pyy(:,idx(new+stg)) = 0.D0
       do ic = 1,stages
         pyy(:,idx(new+stg)) = pyy(:,idx(new+stg))+
          & pa(stg,ic)*ff(:,idx(ic))
       end do
     end do
!$OMP END DO
!$OMP DO PRIVATE(stg,ic) SCHEDULE(STATIC)
     do stg = ist1,stages
       do ic = 1,stages
         pyy(:,idx(new+stg)) = pyy(:,idx(new+stg))+
          & pb(stg,ic)*pyy(:,idx(ic))
       end do
     end do
!$OMP END DO
!$OMP DO SCHEDULE(STATIC)
     do stg = ist1,stages
        call fcn(t+phs*pc(stg),pyy(:,idx(new+stg)),
          &ff(:,idx(new+stg)),cpar)
     end do
!$OMP END DO
!$OMP END PARALLEL
```

Listing 1. Ver-0: loop structure used in EPPEER.

B. Implementation

The following section introduces the implementation of the original EPPEER package. The source code of this implementation can be seen in Listing 1. Each time step of a peer method consists of two basic parts: The linear combination and the function evaluation.

EPPEER implements those two basic parts by three consecutive loop nests: The first and the second loop nest perform the linear combination. For this purpose the first loop nest initializes an $s \times n$ accumulation matrix for the computation of $\mathbf{Y}_{m,i}$ with zeroes, i.e.,

$$\text{for } i = 1, \ldots, s: \quad \mathbf{Y}_{m,i} \leftarrow \mathbf{0} \quad (3)$$

and uses this matrix to accumulate and add the second sum of Eq. (2), $\mathbf{S_f}$:

$$\text{for } i = 1, \ldots, s: \quad \mathbf{Y}_{m,i} \leftarrow \mathbf{Y}_{m,i} + \underbrace{h_m \sum_{j=1}^{s} a_{ij} \mathbf{F}_{m-1,j}}_{\mathbf{S_f}}, \quad (4)$$

where

$$\mathbf{F}_{m-1,j} = \mathbf{f}(t_{m-1,j}, \mathbf{Y}_{m-1,j}) \quad (5)$$

are stored function values computed in the previous time step. After that the second loop nest accumulates and adds the first sum of Eq. (2), $\mathbf{S_Y}$:

$$\text{for } i = 1, \ldots, s: \quad \mathbf{Y}_{m,i} \leftarrow \mathbf{Y}_{m,i} + \mathbf{S_Y}. \quad (6)$$

The third and final loop nest performs the function evaluation to compute $\mathbf{F}_{m,j}$, needed in the next time step, using Eq. (5).

The outermost loop of each loop nest iterates over the stages of the peer method, where the ith iteration of the outer loop computes the argument vector $\mathbf{Y}_{m,i}$ for stage i. Thus, we will refer to these loops as "stage loops". The iterations of the stage loops are independent of each other. That is why the stages

can be computed in parallel for each loop nest. This is a major advantage of the peer methods. The EPPEER package takes this advantage by parallelizing each stage loop with OpenMP.

The stage loops of the first and the second loop nest contain inner loops, which also iterate over the stages in order to compute the linear combinations $\mathbf{S_f}$ and $\mathbf{S_Y}$, respectively, for the stage i corresponding to the current iteration of the outer stage loop. Thus, we will refer to these loops as "combination loops".

The combination loops perform operations on vectors of dimension n and, thus, contain an innermost loop, which iterates over the system dimension and which we therefore call "system loop". Instead of explicitly implementing these system loops as FORTRAN `do` loops, EPPEER uses the FORTRAN vector notation, thus making it easy for the compiler to generate vectorized code for these loops.

C. Performance

In this section, we use the results of runtime experiments to analyze the scalability of the original EPPEER package with the number of cores. The first target system for these experiments is an 8 core Intel Xeon E5-2630V3 (Haswell-EP) CPU. However, the Haswell-EP CPU has two features which might alter the scalability in an unexpected way: Turbo Boost (on-demand increase of clock frequency) and Hyper Threading (2-way simultaneous multithreading (SMT) per core). To avoid potential negative influences of these features on our measurement results, we disabled both features for all measurements. Our second target system is an Intel Xeon Phi 31S1 coprocessor. The Xeon Phi does not have a turbo mode, but it provides 4-way SMT per core, which cannot be disabled. However, to ensure that each thread runs on a separate physical core, we make use of the OpenMP runtime environment to distribute the threads evenly among the cores until the number of threads exceeds the total number of cores. On both target systems, the Intel FORTRAN compiler in version 16.0.2 was used with optimization level 2 (`-O2`) to compile the EPPEER package. All computations were performed using double precision. For profiling purposes we use PAPI in Version 5.4.1. PAPI is a powerful profiling library, which can read the values of CPU-internal performance counters, which can count, for example, the number of cache misses or the number of load/store operations.

We chose two different ODE systems as test problems: The first one is the 2D Brusselator *brus*. It models an oscillating chemical reaction-diffusion system using a two dimensional grid as spatial discretization. The resulting access pattern to the grid points is a five-point 2D stencil. Therefore, *brus* is a sparse problem and its right-hand-side function $\mathbf{f}$ has a time complexity of $\Theta(n)$. For all measurements with *brus*, we chose a vector size of 500 000 elements, where each vector element is a double precision floating point number, requiring 8 bytes of storage space. Thus, a vector corresponds to a grid of 500×500 cells, where each cell contains two double values, so that one vector requires 3.81 MB of storage space. The second test problem is the N-body problem *mbod*, which is also known as the many body problem. It simulates the movement of particles, which interact with each other by the gravitational force. Since the gravitational field of a particle influences every other particle, the N-body problem is a dense problem and its right-hand-side function $\mathbf{f}$ has a time complexity of $\Theta(n^2)$. To obtain the experimental results shown in the following, a scene consisting of 2000 particles was used, where each particle added 6 equations to the ODE system, resulting in a vector size of 12 000 elements (93.75 KB). Table I shows the speedups observed in our experiments.

TABLE I
SPEEDUPS MEASURED WITH THE EPPEER PACKAGE.

		Speedups for different #threads				
Hardware	**Problem**	1	2	3	4	8
Haswell-EP	*brus*	1	1.839	2.006	2.232	2.311
Haswell-EP	*mbod*	1	1.961	2.582	3.779	7.047
Xeon Phi	*brus*	1	1.941	2.619	3.609	6.702
Xeon Phi	*mbod*	1	1.966	2.594	3.804	7.133

Similar to the authors of the EPPEER package, our measurements with the *mbod* problem show a very good scalability. The use of 8 threads yields an almost ideal speedup of 7.0 on Haswell-EP and a speedup of 7.1 on the Xeon Phi. In contrast, the best speedups measured for the *brus* problem are only 2.3 and 6.7 on the Haswell-EP and the Xeon Phi, respectively (on both systems obtained using 8 threads). Thus, obviously, the scalability of the peer methods is influenced by the IVP to be solved.

Interestingly, Table I shows a smaller efficiency when 3 threads are used, which is caused by the following: It is not possible to distribute s equally sized tasks among p threads evenly if $s \bmod p \neq 0$. In our case it is impossible to distribute the 8 stages equally among 3, 5, 6 and 7 threads. This load imbalance forces some of the threads to idle while all other threads complete their last remaining task.

One apparent difference between the two test problems is the access pattern of the right-hand-side function $\mathbf{f}(t_{m,j}, \mathbf{Y}_{m,j})$ to the argument vector $\mathbf{Y}_{m,j}$ and the resulting time complexity. While the dense *mbod* problem has a time complexity of $\Theta(n^2)$, the time complexity of the sparse *brus* problem is only $\Theta(n)$. We can therefore expect that for *brus* the time needed to perform the function evaluations constitutes a smaller fraction of the runtime of a time step than for the *mbod*. In fact, further measurements (Figure 1) have shown that on average evaluating the right hand side of *mbod* takes about 97 % of the total runtime of a time step, whereas the linear combination only takes the remaining 1 to 2 %. In contrast, for *brus*, less than 18 % of the total runtime of a time step is required to compute the function evaluations, but nearly 75 % are needed to perform the linear combination to compute the argument vectors $\mathbf{Y}_{m,j}$.

In total, the results of the runtime experiments show that the scalability depends on the IVP to be solved. For sparse and computationally inexpensive systems like *brus*, computing the linear combinations takes a major part of the runtime (see Figure 1 (b)) and can therefore be expected to have

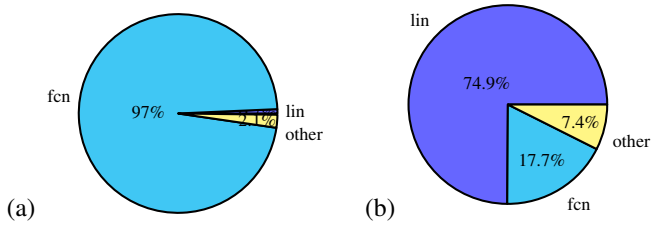

Fig. 1. Average runtime fractions of the linear combination (lin), the function evaluation (fcn) and other parts of the code for (a) *mbod* and (b) *brus*.

a significant influence on scalability. Since the arithmetic intensity, i.e., the ratio of the number of arithmetic instructions to the number of memory instructions, of this part of the time step is low, we can expect the performance of the linear combination to be bound by the performance of the cache and memory subsystem. Moreover, the working set of one time step, i.e., the amount of data accessed in a time step, amounts to four $s \times n$ matrices, which corresponds to 122.1 MB in our experiments and, thus, is significantly larger than the cache size of typical workstations today. We can therefore expect that many expensive accesses to the main memory are required in the experiments with *brus*.

However, for dense and computationally intensive systems like *mbod*, the evaluation of the right-hand-side function of the ODE system dominates the runtime. Since the function evaluations are independent of each other, this leads to the high scalability observed. For the problem size we used, the working set of one time step amounts to 2.92 MB, that means all data accessed during one time step fits completely in the 20 MB L3 cache of the Haswell-EP. The 512 KB L2 cache of a Xeon Phi core can not store the whole working set of a time step, but, to perform the stage-parallel function evaluation efficiently, it is only important that it can store two n-vectors (93.75 KB each) to hold the argument of the function evaluation and its result. The reason for this is that to compute the gravitational force acting on one body the interacting forces with all other $N-1$ bodies are accumulated. Thus, to compute all entries of the function result $\mathbf{f}(t_{m,j}, \mathbf{Y}_{m,j})$, half of the argument vector $\mathbf{Y}_{m,j}$ is read $n/2$ times.[1] Since all important data fits in the cache and the arithmetic intensity is higher than that of *brus*, we can expect the performance of the *mbod* problem to be bound by the compute performance of the processor used. Therefore, improving the locality of memory references further would probably not improve the already high scalability.

In the following, we therefore focus on the sparse case, where an improvement of the scalability is more important and where we can expect to be able to improve the scalability by an improvement of the locality behavior. As test problem we consider only *brus* from now on.

[1] *mbod* is a first order system derived from a second order system by substitution.

IV. Loop Structure of the Linear Combination

A. Variants of the Loop Structure

After the identification of the linear combination as the major fraction of the runtime for sparse ODE systems in the previous section, this section focuses solely on the improvement of the loop structure of the linear combination, possible loop transformations, and their influence on the resulting locality and scalability. The interplay with the evaluation of the right-hand-side function (the third loop nest in the original EPPEER package) will be considered afterwards in Section V.

The data access pattern of the original stage-parallel EPPEER loop structure (referred to as "Ver-0" in the following) is illustrated in Figure 2 for a method with 8 stages and one thread per stage. Only the first two threads are shown, because the data access pattern is similar for all threads: to compute the argument vector $\mathbf{Y}_{m,i}$ of their stage i, the whole two $s \times n$ matrices Y_{m-1} and F_{m-1} are read, and after each loop nest the result of the computations within the loop nest are written back to memory. Hence, in total there are $2s+1$ write accesses to each element of the matrix Y_m: initialization with zero, accumulation of sum $\mathbf{S_Y}$ and accumulation of sum $\mathbf{S_Y}$. Though the stages are computed in parallel by different threads, in most current shared-memory computers several threads share parts of the memory subsystem (e.g., shared higher level caches, main memory modules connected to a socket) so that they compete for these limited resources and may quickly reach their limits.

Since the system dimension is the stride-1 dimension, the innermost loops of Ver-0 iterate over the stride-1 dimension, which leads to high spatial locality and allows the compiler to vectorize the loads and stores using sequential SIMD load/store instructions.

The first improvement of the loop structure we consider can be seen in Listing 2 and will be referred to as "Ver-1" in the following. Ver-1 adopts the parallelization across the stages from Ver-0, but it contains two modifications. First, it eliminates the zero initialization of Y_m by peeling off the first iteration of the combination loop. This saves one pass over the matrix Y_m and, thus, many expensive memory accesses. Moreover, the first and the second loop nest of Ver-0 are fused, which both contribute to the computation of $\mathbf{Y}_{m,i}$, to a single loop nest. This loop fusion is legal, because the only dependencies between those two loop nests are the write accesses to the matrix Y_m accumulating the sums in Eq. 2. That is why changing the order of the loops, and thus the order of the accumulation, only influences the round-off error. An advantage of this new fused loop structure is that only s write accesses to each element of Y_m are necessary.

Trying to overcome some disadvantages of Ver-0 and Ver-1, Ver-2 (Listing 3) parallelizes the linear combinations across the system dimension. This version is based on an earlier version of the EPPEER package. It is obtained by interchanging the loops of each loop nest of Ver-0 so that the outermost loop iterates over the system dimension and the two inner loops iterate over the stages. To enable this interchange,

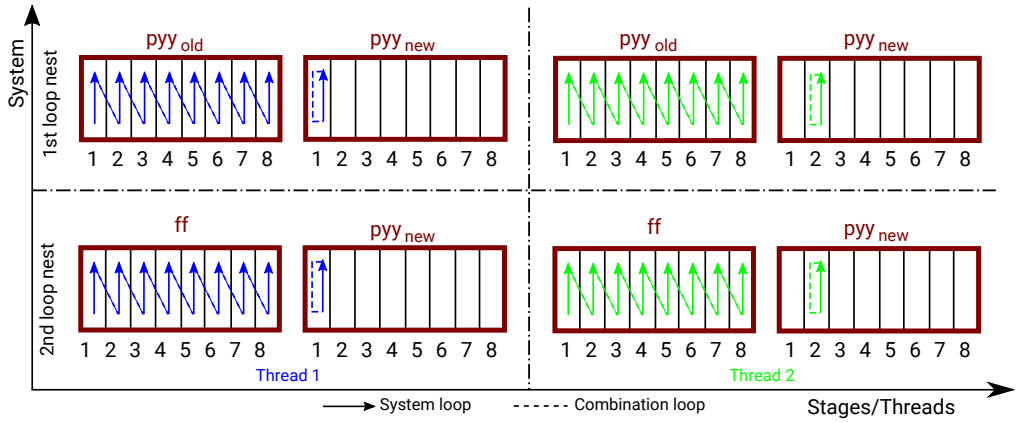

Fig. 2. Data access pattern of the original EPPEER package (Ver-0).

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO PRIVATE(stg,ic) SCHEDULE(STATIC)
    do stg = ist1,stages
      pyy(:,idx(new+stg)) = pa(stg,1)*ff(:,idx(1)) +
         & pb(stg,1)*pyy(:,idx(1))
      do ic = 2,stages
        pyy(:,idx(new+stg)) = pyy(:,idx(new+stg)) +
           & pa(stg,ic)*ff(:,idx(ic)) + pb(stg,ic) *
           & pyy(:,idx(ic))
      end do
    end do
!$OMP END DO
!$OMP END PARALLEL
```

Listing 2. Ver-1: Ver-0 improved by loop peeling and fusion.

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO PRIVATE(id,stg,ic) SCHEDULE(STATIC)
    do id = 1,nprob
      do stg = ist1,stages
        pyy(id,idx(new+stg)) = 0.D0
        do ic = 1,stages
         pyy(id,idx(new+stg)) = pyy(id,idx(new+stg))+
           &  pa(stg,ic)*ff(id,idx(ic))
        end do
      end do
    end do
!$OMP END DO
!$OMP DO PRIVATE(id,stg,ic) SCHEDULE(STATIC)
      do id = 1,nprob
        do stg = ist1,stages
          do ic = 1,stages
           pyy(id,idx(new+stg)) = pyy(id,idx(new+stg))+
             & pb(stg,ic)*pyy(id,idx(ic))
          end do
        end do
      end do
!$OMP END DO
!$OMP END PARALLEL
```

Listing 3. Ver-2: System parallel execution.

the previously innermost system loops, which were defined implicitly using vector notation, had to be transformed into an explicit FORTRAN do loop. As a result, a higher temporal locality for the write accesses to the elements of the matrix Y_m is generated. The compiler can now keep the temporary partial sums in a register during the s^2 iterations over the stages by the inner loops of each loop nest, so that now each element of Y_m needs only be written to main memory twice.

Even more important, the amount of data accessed by each thread is reduced significantly: While in Ver-0 and Ver-1 each thread needs to read all elements of the matrices Y_{m-1} and F_{m-1} to compute the stage(s) assigned to it, in Ver-2 it reads only the range of elements of Y_{m-1} and F_{m-1} assigned to it and it writes only the corresponding range of elements of Y_m. Hence, if there are p threads, each thread only reads at most $2s\lceil\frac{n}{p}\rceil$ elements and writes at most $s\lceil\frac{n}{p}\rceil$ elements, whereas in Ver-0 and Ver-1 each thread reads $2sn$ elements and writes at most $\lceil\frac{s}{p}\rceil$ elements.

While Ver-1 can use only as many threads as the peer method used has stages, an additional benefit of the system parallel execution is that more threads can participate in the computation of the linear combination.

Unfortunately, the new loop structure of Ver-2 also has disadvantages. Since the iteration over the stride-1 dimension is performed by the outermost loop, the reads of the matrix elements in the innermost loops have a large stride of n. Hence, the higher temporal locality comes at the expense of lower spatial locality. Further, the compiler cannot vectorize the loads and stores using sequential SIMD load/store instructions. However, there are SIMD gather or scatter instructions, which make a vectorization of strided memory accesses possible. Although all modern Intel CPUs since Haswell support AVX2 gather instructions, our Haswell-EP still emulates those instructions by microcode. Thus, gather instructions still have a low throughput on Haswell-EP.

In order to derive a loop structure with high temporal as well as spatial locality, we can make use of loop tiling (Ver-3, Listing 4). Ver-3 also makes use of both optimizations from Ver-1 (elimination of the zero initialization phase and loop fusion) yielding a similar loop structure as in Ver-1. However, in contrast to Ver-1, it has an outer parallel tile loop, which iterates over the system dimension in larger steps dividing the system into tiles of a user-defined size. Inside the tile loop run the stage loop and the combination loop. The innermost loop is the intra-tile loop, which iterates over the elements of the

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO PRIVATE(stg,ic, idi) SCHEDULE(STATIC)
do id = 1,nprob, tile_size
  do stg = ist1,stages
    do idi = id, min(id+tile_size-1,nprob)
      pyy(idi,idx(new+stg)) = pb(stg,1)*pyy(idi,idx(1))+
        & pa(stg,1)*ff(idi,idx(1))
    end do
    do ic = 2,stages
      do idi = id, min(id+tile_size-1,nprob)
        pyy(idi,idx(new+stg)) = pyy(idi,idx(new+stg))+
          & pb(stg,ic)* pyy(idi,idx(ic)) +
          & pa(stg,ic) * ff(idi,idx(ic))
      end do
    end do
  end do
end do
!$OMP END DO
!$OMP END PARALLEL
```

Listing 4. Ver-3: Ver-2 improved by loop peeling, fusion, and tiling.

TABLE II
RUNTIME COMPARISON FOR 100 TIME STEPS ON HASWELL-EP AND XEON PHI WITH GRID SIZE 500×500

		Runtime for different #threads				
Hardware	**Version**	1	4	8	60	120
Haswell-EP	Ver-0	10.73	4.98	5.02	-	-
Haswell-EP	Ver-1	8.64	4.00	3.45	-	-
Haswell-EP	Ver-3	2.53	0.95	0.76	-	-
Xeon Phi	Ver-0	50.01	13.06	6,69	-	-
Xeon Phi	Ver-3	19.54	4.90	2.47	0.39	0.29

corresponding tile, i.e., the stride-1 dimension. The resulting memory access pattern is illustrated in Figure 3. Only the same amount of data needs to be accessed as in Ver-2, but, since the tile loop now contains the stage loop and the combination loop, an iteration of the tile loop computes the linear combination for one tile for all the stages. This provides a blocking effect across the stages: Assuming the tile size is small enough so that the data accessed in one iteration of the tile loop fits in the cache, the final values of the elements of Y_m have to be sent to main memory only once. Since the innermost loop iterates over the stride-1 dimension, there also is a high spatial locality, so that cache lines can be reused once they have been loaded into the cache, and efficient SIMD vectorization is possible. Because of that, choosing a good tile size for Ver-3 is important.

B. Performance of the Loop Structure Variants

In this section we will analyze how the different versions so far scale with the amount of active cores. For this purpose we have measured the runtimes of the different versions on the Haswell-EP and on the Xeon Phi. The runtimes measured are given in Table II. The scaling behavior is shown in Figures 4 and 6 in terms of the inverted runtime (reciprocal of the runtime) as a function of the number of threads. As in a speedup diagram, a linear growth of the inverted runtime corresponds to a good scalability, but, in contrast to a speedup diagram, there is no need to chose a sequential reference runtime.

We can divide the versions so far into two groups: The first group consists of the original EPPEER version (Ver-0) and Ver-1, which both only utilize stage parallelism, while the second group consists of Ver-2 and Ver-3, which both only utilize system parallelism for the linear combination.

Because the peer methods part of EPPEER possess at most 8 stages, the stage-parallel variants use up to 8 cores only. Furthermore, there is a load imbalance when 3, 5, 6, or 7 threads are used. Ver-1 of the first group, which improves the original EPPEER version by removing the zero initialization and fusing the two loop nests, is about 1.6 times faster than the original EPPEER version.

In contrast to the stage-parallel variants, the system-parallel variants can potentially use more than 8 cores and are not affected by the load imbalance. The cause of this is the system dimension offering a higher degree of parallelism. Ver-2 uses system parallelism without loop tiling. That is why it eventually became up to 3 times faster than the best stage-parallel variant on the Haswell-EP. However, when Ver-2 was used on less than 4 cores, it was slower than the stage-parallel variants.

Unlike Ver-2, Ver-3 achieves system parallelism with an efficient memory access pattern by utilizing loop tiling. Preparatory runtime measurements had shown that a tile size of about 128 was best for almost all problem sizes. For the following experiments, therefore 128 was used as tile size for Ver-3. Figure 5 shows the normalized runtime of Ver-3 with different tile sizes for a grid size of 500×500 and 700×700 and for one core and eight cores on the Haswell-EP. For both problem sizes and both core numbers there is a runtime minimum between tile size 64 and 256.

Because of its more efficient memory access pattern, Ver-3 was about three times faster than Ver-2 running on the same amount of cores on the Haswell-EP. Ver-3 was also about 8 times faster than the best stage-parallel variant, if both used the same amount of cores. This suggests Ver-3 also having a more cache friendly memory access pattern than the stage-parallel variants.

On the Xeon Phi also Ver-3 obtains the best runtime. On this processor we could observe a reduction of the runtime for up to 120 threads, where for large numbers of threads Ver-3 clearly outperforms Ver-2.

To measure the locality behavior, we measure the normalized runtime (Figures 7 and 8), i.e., the runtime per time step divided by the number of equations, n, on the Haswell-EP. Since the right-hand-side of *brus* has costs $\Theta(n)$, an increase in the normalized runtime usually indicates working sets falling out of a cache level. As we can see, the normalized runtime of Ver-3 is significantly lower than that of the other versions, and it does not increase as strongly when the problem size exceeds a certain threshold, which depends on the number of threads.

In addition, we measured the L3 cache misses and the total amount of store and load operations. The measurements confirm that Ver-3 has a smaller L3 cache miss rate in relation to the total load/store operations than both stage-parallel versions (see Figure 9). Furthermore, this plot shows also that the L3 cache miss rate is much lower on small problem sizes.

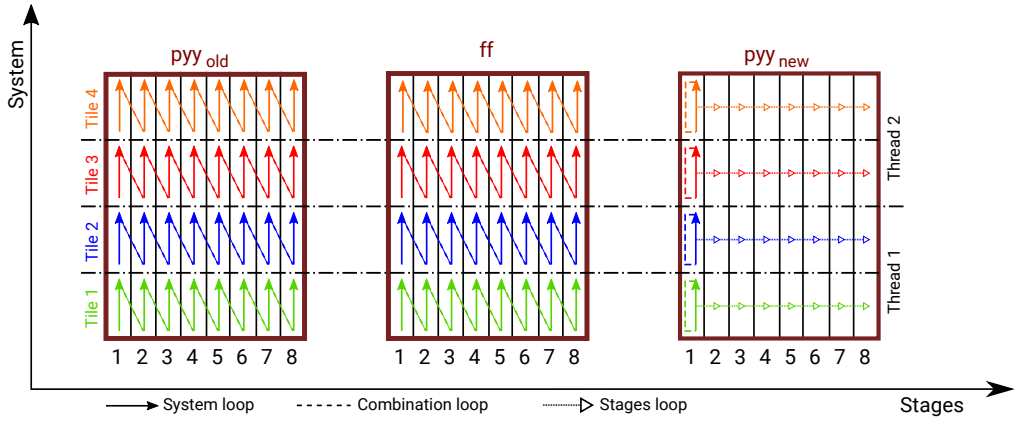

Fig. 3. Data access pattern of the optimized system-parallel loop structure (Ver-3).

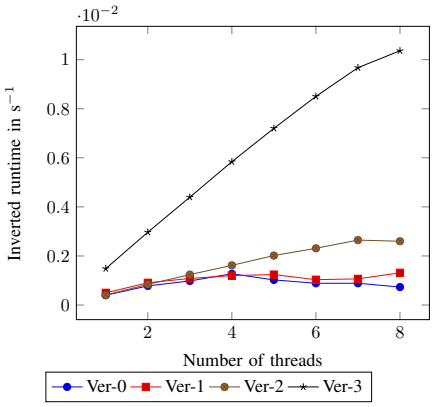

Fig. 4. Inverted runtime of different versions of the linear combination of *brus* on Haswell-EP in seconds.

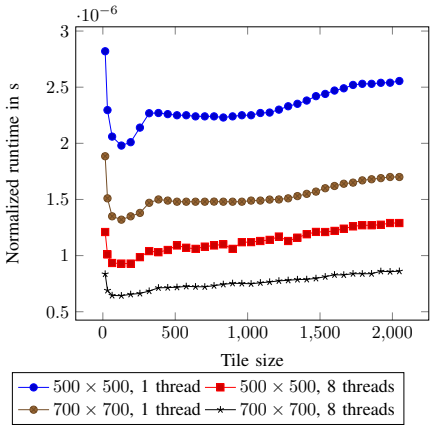

Fig. 5. Normalized runtime of Ver-3 in seconds for different tile sizes (x-axis), different grid sizes and numbers of threads on Haswell-EP.

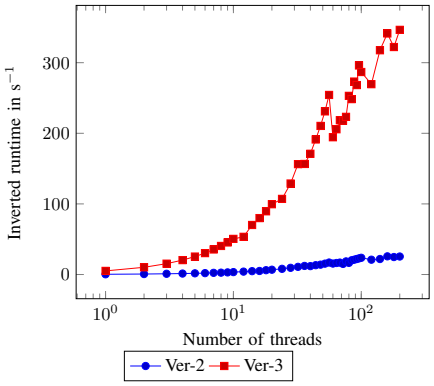

Fig. 6. Inverted runtime of system-parallel versions of the linear combination of *brus* on Xeon Phi for one time step in seconds.

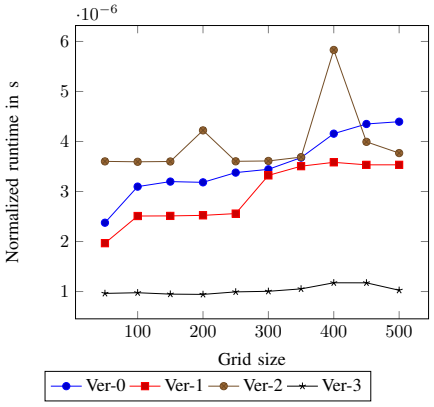

Fig. 7. Normalized runtime of the linear combination of *brus* on Haswell-EP in seconds for 1 thread.

V. Interplay with the Function Evaluation and Fully System-Parallel Execution

In the last section we have only focused on optimizing the linear combination without modifying the loop nest, which evaluates the problem function. In this section, however, we will improve the problem function evaluation and the

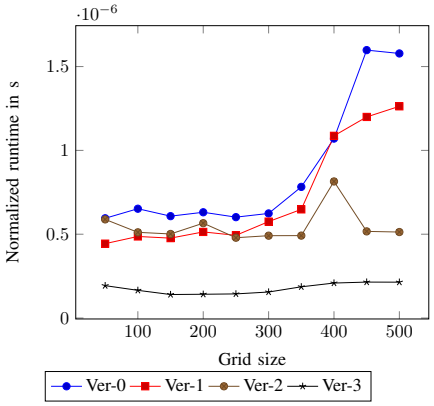

Fig. 8. Normalized runtime of the linear combination of *brus* on Haswell-EP in seconds for 8 thread.

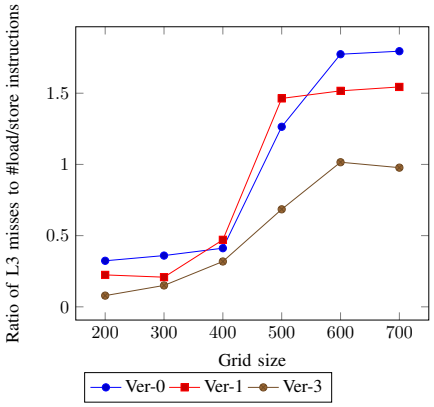

Fig. 9. Ratio of L3 misses to #load/store instructions for 8 threads and variable grid size.

interaction of the problem function with the linear combination.

The runtime experiments of the previous section have confirmed that an optimized system-parallel tiled loop variant of the linear combination such as Ver-3 outperforms stage parallel variants, because it requires less memory references and can efficiently exploit temporal and spatial locality. However, up until now all implementation variants evaluated the problem function using parallelism across the stages.

Unfortunately, combining the system-parallel linear combination with a stage-parallel function evaluation has several disadvantages: The stage-parallel function evaluation can use as many cores as there are stages only. Furthermore both a system parallel linear combination and method parallel function evaluation have different memory access patterns, which causes a data redistribution and reduces locality. That is why we modify the loop nest containing the problem function to adopt system parallelism. However, up to now the problem function has been evaluated by a single function call that computed the temporal derivatives for every element of a given vector, which is not suited for system parallelism.

Therefore, we implemented two new versions of the *brus*

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO PRIVATE(stg) SCHEDULE(STATIC)
 do id = 1,nentries,tile_size
   do stg = ist1,stages
     do idi = id, min(id+tile_size,nentries)
       call fcn(idi,t+phs*pc(stg),pyy(:,idx(new+stg)),
            & ff(:,idx(new+stg)),cpar)
     end do
   end do
 end do
!$OMP END DO
!$OMP END PARALLEL
```

Listing 5. Elementwise function evaluation.

```
do stg = ist1,stages
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO SCHEDULE(STATIC)
  do i = 1,nentries,block_size
     call fcn(i,block_size,t+phs*pc(stg),
       & pyy(:,idx(new+stg)), ff(:,idx(new+stg)),cpar)
  end do
!$OMP END DO
!$OMP END PARALLEL
end do
```

Listing 6. Blockwise function evaluation.

problem with a modified signature and a modified internal structure: The first new version (*element* version) calculates the temporal derivatives for one single element of the problem vector per call. Yet again, this design has two major disadvantages: The *element* version has to perform the boundary checks of the stencil for each call, and the compiler is not able to vectorize this function efficiently. Moreover the temporal derivatives of the two substances contained in a grid cell have some computations in common, which the *element* version has to compute redundantly twice. We can avoid all those inefficiencies if we do not call the right-hand-side function once per element, but once for a range of elements, and optimize the internal structure of the function accordingly. This results in the implementation of the *block* version. Finally, we have adapted the loop nest evaluating the problem functions to the *element* version in Listing 5 and to the *block* version in Listing 6. Since the *element* version cannot provide any blocking itself, we have added a simple 1D-blocking scheme to the loop nest.

Inverted runtimes measured for the two new variants of *brus* are shown in Figures 10 and 11. Here, the grid size used is 500×500, the number of stages is set to 8, and 200 is used as block size for the *block* version. As expected, both new versions introduced in this section are faster. The *block* version has the best runtime. The *element* version first starts slower than the original implementation, because this version invokes the function evaluation n times, each time performing a test whether the current index corresponds to a boundary point or not. This leads to a high overhead. But for large numbers of cores, the higher locality of the *element* version can compensate this. In particular, it can obtain speedups higher than the number of stages available.

Hence, all in all, the best runtime for *brus* is obtained by the system-parallel linear combination with loop tiling, loop

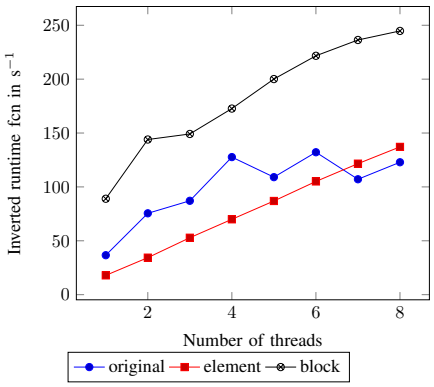

Fig. 10. Inverted runtime for different function evaluations (row, element, block) on Haswell-EP for one function invocation using Ver-3.

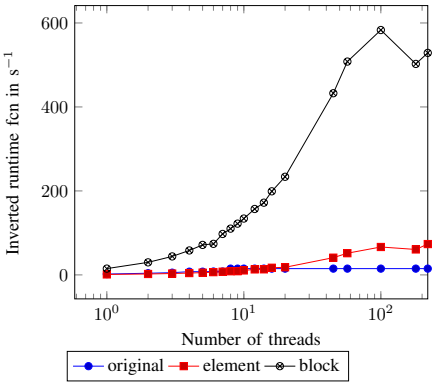

Fig. 11. Inverted runtime for different function evaluations (row, element, block) on Xeon-Phi for one function invocation using Ver-3.

fusion and loop peeling combined with the system-parallel blockwise function evaluation.

VI. Conclusions

In this paper, we have considered the influence of locality on the scalability of method- and system-parallel explicit peer methods. In particular, we have focused on improving the scalability for large sparse ODE systems with cheap function evaluation costs. After confirming that the linear combination to compute the argument vectors requires the major part of the runtime for such ODE systems, we have considered several transformations of the loop structure, analyzed their memory access pattern and measured the resulting runtimes and scalability on a Haswell-EP and on a Xeon Phi processor. As expected, the results of the runtime experiments confirmed that a system-parallel computation of the linear combination leads to a better performance than a stage-parallel computation because less memory references are required and a more cache-efficient data access pattern can be employed. A further performance improvement is possible when also the function evaluation can be performed in a system-parallel way, so that no data-redistribution is necessary.

Acknowledgment

We would like to thank Bernhard Schmitt for providing the EPPEER package and Simon Melzner for hardware support.

References

[1] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed. Berlin: Springer, 2000.
[2] K. Burrage, *Parallel and Sequential Methods for Ordinary Differential Equations*. New York: Oxford University Press, 1995.
[3] B. A. Schmitt, "Peer methods for ordinary differential equations," http://www.mathematik.uni-marburg.de/~schmitt/peer/, last checked 2015-08-17.
[4] B. A. Schmitt and R. Weiner, "Parallel two-step W-methods with peer variables," *SIAM J. Numer. Anal.*, vol. 42, no. 1, pp. 265–282, 2004.
[5] B. A. Schmitt, R. Weiner, and S. Beck, "Two-step peer methods with continuous output," *BIT Numer. Math.*, vol. 53, pp. 717–739, 2013.
[6] B. A. Schmitt, R. Weiner, and S. Jebens, "Parameter optimization for explicit parallel peer two-step methods," *Appl. Numer. Math.*, vol. 59, pp. 769–782, 2009.
[7] R. Weiner, K. Biermann, B. A. Schmitt, and H. Podhaisky, "Explicit two-step peer methods," *Comput. Math. Appl.*, vol. 55, no. 609–619, 2008.
[8] B. A. Schmitt and R. Weiner, *Manual for explicit parallel peer code EPPEER*, Aug. 2012.
[9] Y. Maday and G. Turinici, "A parareal in time procedure for the control of partial differential equations," *C.R.A.S. Sér. I Math*, vol. 335, pp. 387–391, 2002.
[10] P. J. van der Houwen and E. Messina, "Parallel Adams methods," *J. Comput. Appl. Math.*, vol. 101, pp. 153–165, Jan. 1999.
[11] K. Ahnert, D. Demidov, and M. Mulansky, "Solving ordinary differential equations on gpus," in *Numerical Computations with GPUs*, V. Kindratenko, Ed. Springer, 2014, ch. 7, pp. 125–157.
[12] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.
[13] M. Korch and T. Rauber, "Parallel low-storage Runge-Kutta solvers for ODE systems with limited access distance," *Int. J. High Perf. Comput. Appl.*, vol. 25, no. 2, pp. 236–255, 2011. doi: 10.1177/1094342010384418
[14] ——, "Locality optimized shared-memory implementations of iterated Runge-Kutta methods," in *Euro-Par 2007*, ser. LNCS, vol. 4641. Springer, 2007, pp. 737–747.
[15] R. Karrenberg, "Automatic SIMD vectorization of SSA-based control flow graphs," Dissertation, Universität des Saarlandes, Saarbrücken, Jul. 2014.
[16] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: A compiler framework for analyzing and tuning memory behavior," *ACM Trans. Prog. Lang. Syst. (TOPLAS)*, vol. 21, no. 4, pp. 703–746, 1999.
[17] D. Feld, T. Soddemann, M. Jünger, and S. Mallach, "Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation," in *Proc. of the 3rd International Workshop on Polyhedral Compilation Techniques*, A. Größlinger and L.-N. Pouchet, Eds., Berlin, Germany, Jan. 2013, pp. 45–54.
[18] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
[19] G. Hager, J. Treibig, J. Habich, and G. Wellein, "Exploring performance and power properties of modern multi-core chips via simple machine models," *Concurrency and Computation: Practice and Experience*, vol. 28, pp. 189–210, 2016.
[20] J. Ansel, "Autotuning programs with algorithmic choice," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, Feb. 2014.
[21] N. Kalinnik, M. Korch, and T. Rauber, "Online auto-tuning for the time-step-based parallel solution of ODEs on shared-memory systems," *Journal of Parallel and Distributed Computing*, vol. 74, no. 8, pp. 2722–2744, 2014. doi: 10.1016/j.jpdc.2014.03.006
[22] B. A. Schmitt and R. Weiner, "Parallel start for explicit parallel two-step peer methods," *Numerical Algorithms*, vol. 53, no. 2-3, pp. 363–381, 2010.