Locality Improvement of Data-Parallel Adams–Bashforth Methods through Block-Based Pipelining of Time Steps

Matthias Korch

University of Bayreuth
Applied Computer Science 2
korch@uni-bayreuth.de

Abstract. Adams–Bashforth methods are a well-known class of explicit linear multi-step methods for the solution of initial value problems of ordinary differential equations. This article discusses different data-parallel implementation variants with different loop structures and communication patterns and compares the resulting locality and scalability. In particular, pipelining of time steps is employed to improve the locality of memory references. The comparison is based on detailed runtime experiments performed on parallel computer systems with different architectures, including the two supercomputer systems JUROPA and HLRB II.

1 Introduction

Many time-dependent processes can be modeled by initial value problems (IVPs) of ordinary differential equations (ODEs):

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

where $\mathbf{y}(t) \in \mathbb{R}^n$ is the solution function to be computed for the interval $t \in [t_0, t_e]$, $\mathbf{y}_0$ is the given *initial value*, i.e., the initial state of the process to be simulated at time t_0, and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is the given *right-hand-side function*, which describes the rates of change of the process to be simulated.

This article considers Adams–Bashforth (AB) methods [2,5,6] on an equidistant grid. At each time step $\kappa = 0, 1, 2, \ldots$, these methods compute an approximation $\mathbf{y}_{\kappa+1}$ to the solution function at time $t_{\kappa+1}$, $\mathbf{y}(t_{\kappa+1})$, using function results of the last k preceding time steps and weights $\beta_1, \ldots, \beta_k$ according to the scheme

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{l=1}^{k} \beta_l \mathbf{f}(t_{\kappa-l+1}, \mathbf{y}_{\kappa-l+1}). \tag{2}$$

AB methods belong to the class of *explicit linear k-step* or, more generally, *multi-step* methods and are suitable for *nonstiff* IVPs (see [2,5,6] for a discussion of stiffness and explicit vs. implicit methods).

Many parallel IVP solution methods have been proposed. An overview of the fundamental work and further references can be found in [1]. Recent work on

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 563–576, 2012.

parallel ODE methods includes variants of iterated Runge–Kutta methods [4] and peer two-step methods [12]. Many 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 [13], which belong to the class of *general linear methods* [2]. Different parallel execution schemes for these methods are investigated and discussed in [11].

This article focuses on the *parallelism across the ODE system* available in classical k-step AB methods (2), i.e., the computation of the components $y_{\kappa+1,1}, \ldots, y_{\kappa+1,n}$ of $\mathbf{y}_{\kappa+1}$ is distributed across the processing elements and performed in data-parallel style. Different loop structures for (2) and corresponding communication patterns for ODE systems with arbitrary coupling and for ODE systems with a special coupling structure called *limited access distance* are described, and the influence on locality and scalability is discussed. Double-precision implementations have been written in C for shared and distributed address space using POSIX Threads (Pthreads) and MPI, respectively. Starting point were Pthread implementations [10], which have, for this article, been optimized for NUMA (*non-uniform memory access*) architectures and been complemented by MPI implementations. Scalability and locality have been investigated using runtime experiments on several computer systems with different architectures, including the two supercomputer systems HLRB II and JUROPA.

2 Parallel Implementation of General AB Solvers

2.1 Possible Loop Structures

Equation (2) leads to a doubly nested, fully permutable loop structure, since one iteration over the summands $\beta_l \mathbf{f}(t_{\kappa-l+1}, \mathbf{y}_{\kappa-l+1})$ for $l = 1, \ldots, k$ and one iteration over the system dimension $j = 1, \ldots, n$ is required. It is sufficient to compute one evaluation of the right-hand-side function $\mathbf{f}(t_\kappa, \mathbf{y}_\kappa)$ per time step if the function results of the previous $k-1$ time steps are kept in memory. These considerations lead to the following three loop structures:

j–l. The j-loop over the large system dimension is chosen as outer loop. The l-loop to compute the j-th component of the vector $\mathbf{y}_{\kappa+1}$, $y_{\kappa+1,j} = y_{\kappa,j} + h \sum_{l=1}^{k} \beta_l F_{l,j}$ with $F_{l,j} := f_j(t_{\kappa-l+1}, \mathbf{y}_{\kappa-l+1})$ runs inside the j-loop. This results in a high temporal locality for this vector component since its storage location is reused in the partial sum updates.

l–j. The l-loop over the k steps of the AB method is chosen as outer loop. In each iteration of the l-loop, a j-loop over the system dimension is executed, which accesses the two vectors $\mathbf{F}_l := \mathbf{f}(t_{\kappa-l+1}, \mathbf{y}_{\kappa-l+1})$ and $\mathbf{y}_{\kappa+1}$. This results in a high spatial locality since cache lines of these two vectors can be reused for subsequent vector components once they have been loaded into the cache. Moreover, this loop structure can benefit from hardware prefetching, and its access pattern is easily predictable by the hardware prefetcher.

Tiling. Since the two loops are fully permutable, they can also be tiled to create an additional working space that fits in the cache. This leads to a triply nested loop structure, where the outermost loop (j-loop) iterates over the system dimension with stride B (*tile size* or *block size*). Inside the j-loop runs the l-loop, which iterates over the k steps. The innermost loop (jj-loop) again iterates over the system dimension and accesses the components $j, \ldots, j+B-1$ of the two vectors $\mathbf{F}_l$ and $\mathbf{y}_{\kappa+1}$. Thus, a high spatial locality results from the innermost loop iterating over two vectors with stride 1, but also a high temporal locality results from the reuse of a block of size B of the vector $\mathbf{y}_{\kappa+1}$ in successive iterations of the l-loop.

2.2 Parallelization

In data-parallel implementations, the system dimension $1, \ldots, n$ is partitioned among the processing elements. For highest spatial locality, a blockwise distribution is appropriate, such that each of the p processing elements is assigned a block of n/p consecutive components. As data structures, $k-1$ function results and the two approximation vectors $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ have to be stored in memory. In a sequential implementation, the function results can be stored in a $(k-1) \times n$ 2D array that is used in a cyclic fashion such that $\mathbf{f}(t_{m+k-1}, \mathbf{y}_{m+k-1})$ overwrites $\mathbf{f}(t_m, \mathbf{y}_m)$. Similarly, $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ can be stored in two 1D arrays of size n, the pointers to which are swapped at each time step.

The 2D array holding the function results can be distributed to the processing elements such that each processing element stores locally a partition of size $(k-1) \times n/p$ of this 2D array. This is necessary in an MPI implementation because of the separate address spaces. But even for shared address space, a distributed storage of the function results in separate, thread-local $(k-1) \times n/p$ 2D arrays often is preferable, because it ensures that all function results associated with a thread can be stored in local memory of the processing element on which the thread is executed. This is particularly important on NUMA architectures. Though the "first touch" policy applied by modern operating systems to support NUMA will also move memory pages of shared arrays to local memory of the thread that first writes to the page, shared data structures may lead to sharing of memory pages and thus to remote memory accesses at the borders of the data ranges of the threads. At a finer grained level, sharing of cache lines may decrease performance even on UMA (*uniform memory access*) architectures.

For the two vectors $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$, distributed storage is not always possible or preferable, because the function $\mathbf{f}$ has to be evaluated for $\mathbf{y}_\kappa$ and, in the general case, this function evaluation may use all components of $\mathbf{y}_\kappa$. Therefore, the general shared-address-space implementations considered in this article implement these two vectors as shared data structures. One barrier operation is required per time step to prevent that threads start the function evaluation before all other threads have computed their share of the argument vector. The MPI implementations require a replicated storage of the argument vector. Since each of the MPI processes computes n/p components of the argument vector, it must be gathered by all processes using a multibroadcast operation (`MPI_Allgatherv()`). The

```
for (i = k - 1; i < steps; i++)
{
  MPI_Allgatherv(Y_cur + first_elem, num_elems, MPI_DOUBLE,
                 Y_arg, counts, offsets, MPI_DOUBLE, MPI_COMM_WORLD);

  for (j = first_elem; j <= last_elem; j += B)
  {
    for (jj = j; jj < j + B; jj++) Y_new[jj] = b[k - 1] * F[i % (k - 1)][jj];

    for (l = 1; l < k - 1; l++)
      for (jj = j; jj < j + B; jj++) Y_new[jj] += b[j] * F[(i - l) % (k - 1)][jj];

    for (jj = j; jj < j + B; jj++) F[i % (k - 1)][jj] = f(jj, t0 + i * h, Y_arg);
    for (jj = j; jj < j + B; jj++) Y_new[jj] += b[0] * F[i % (k - 1)][jj];
    for (jj = j; jj < j + B; jj++) Y_new[jj] = Y_cur[jj] + h * Y_new[jj];
  }

  swap_vectors(&Y_new, &Y_cur);
}
```

Listing 1. General parallel MPI implementation with tiled loop structure

implementations considered in this article therefore store in each process n/p components of $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ and one additional array of size n, in which $\mathbf{y}_\kappa$ is gathered and which is then used as argument vector for the function evaluation.

Listing 1 shows a code fragment of a general parallel implementation of one time step of an AB method with tiled loop structure that uses MPI as programming environment. The loop structures j–l and l–j can be interpreted as special cases of the tiled loop structure using $B = 1$ and $B = n$, respectively.

3 Reducing Parallel Overhead through Specialization

There are many sparse ODE systems where the components of $\mathbf{y}_\kappa$ accessed by a component function $f_j(t_\kappa, \mathbf{y}_\kappa)$ are located nearby the index j. Examples are ODE systems resulting from a spatial discretization of PDE systems by the method of lines. This property is measured by the *access distance* $d(\mathbf{f})$, which is the smallest value b, such that all component functions $f_j(t_\kappa, \mathbf{y}_\kappa)$ access only the components $\{y_{\kappa,j-b}, \ldots, y_{\kappa,,j+b}\}$. We say $d(\mathbf{f})$ is *limited* if $d(\mathbf{f}) \ll n$.

For ODE systems with limited access distance, data-parallel implementations with a blockwise data distribution only need to exchange $d(\mathbf{f})$ components of $\mathbf{y}_\kappa$ at the left and at the right border of their data range. Using MPI as programming environment, the expensive multibroadcast operation `MPI_Allgatherv()` (cf. Fig. 1) can be replaced by non-blocking single transfer operations (`MPI_Isend()` and `MPI_Irecv()`), thus potentially overlapping communication with computations. Replicated storage of the argument vector for the function evaluation using an array of size n is no longer required. Instead, the two arrays holding the local parts of $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ of size n/p are enlarged by $d(\mathbf{f})$ at each border to store the data received from the neighbor processes (*ghost cells*).

In shared-address-space implementations, it is desirable to avoid the high costs of the global barriers (cf. Fig. 2). If the ODE system has a limited access

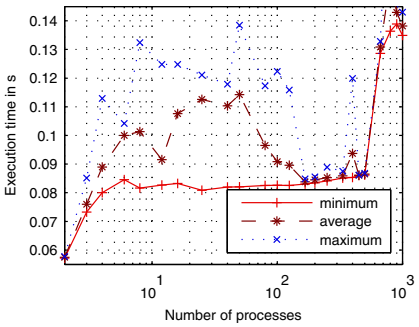

Fig. 1. Execution time of the communication operation `MPI_Allgatherv()` for $8 \cdot 10^6$ vector elements on HLRB II

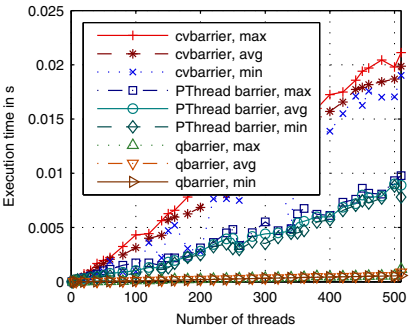

Fig. 2. Comparison of the execution time of barrier operations on HLRB II (barrier based on condition variables, Pthread barrier, barrier based on busy waiting)

distance, only data from neighbor threads are required for the function evaluation. Hence, it is sufficient to use locks for synchronization between neighbors. Similar to the way communication and computation could be overlapped in an MPI implementation, no waiting times for acquiring the locks occur if all threads process the ODE components synchronously at the same speed.

While shared data structures are needed in general implementations where the function evaluation may access all components of its argument vector, implementations specialized in a limited access distance can store the vectors $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ in a distributed fashion similar to the MPI implementations, thus avoiding page and cache line sharing. In this case, the data required from neighbor threads have to be copied to ghost cells, which consumes CPU time.

4 Pipelining of Time Steps

The loop structures considered in Section 2.1 allow that the evaluation of the right-hand-side function $\mathbf{f}(t_\kappa, \mathbf{y}_\kappa)$ accesses all components of $\mathbf{y}_\kappa$. Next, a pipeline-like loop structure covering several time steps of the AB method is described that can be used for ODE systems with limited access distance to increase locality. A similar approach has been proposed for the stages of embedded Runge–Kutta methods [8], the corrector steps of iterated Runge–Kutta methods [7], and the micro-steps of extrapolation methods [9].

The pipeline-like loop structure is based on a subdivision of all n-vectors into $n_B = \lceil n/B \rceil$ blocks of size B. This subdivision is similar to the subdivision for loop tiling, but, for the pipelining scheme to work, the block size must be larger than the access distance, i.e., $B \geq d(\mathbf{f})$, and the number of blocks, n_B, must be at least as large as the pipeline length L.

Given this subdivision, the function evaluation of a block $J \in \{1, \ldots, n_B\}$ of $\mathbf{y}_\kappa$ uses only components of the blocks $J-1$, J, and $J+1$ of $\mathbf{y}_\kappa$ if these blocks

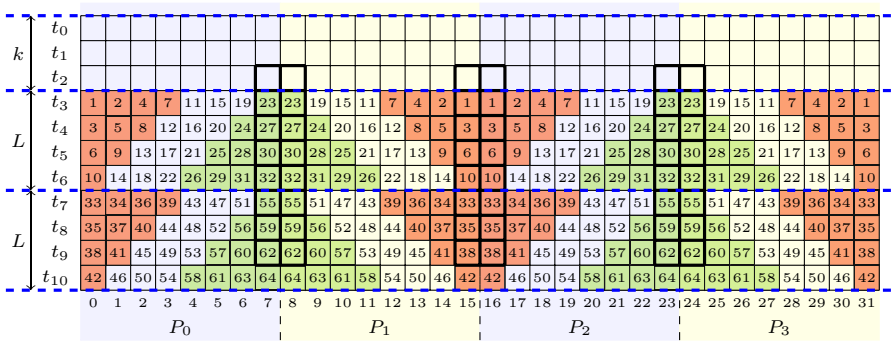

Fig. 3. Illustration of the pipeline-like processing of time steps in a data-parallel implementation for $n_B = 32$, $p = 4$, $k = 3$, and $L = 4$. Blocks that need to be exchanged between processing elements are highlighted by thick borders.

exist, i.e., if $1 < J < n_B$. Now, due to the dependence pattern of the blocks, a sequence of L successive time steps can be computed in a single sweep over the system dimension using the block computation order displayed in Fig. 3. In a data-parallel implementation, neighbor processing elements process their pipelines in opposite directions so that during the initialization and the finalization of the pipelines all data that have to be received from neighbor processing elements are available before they are needed for a function evaluation.

Data-parallel implementations of the pipeline-like loop structure can use the same distributed data structures and the same efficient communication patterns as the specialized implementations optimized for limited access distance described in Section 3.

A modification of the computation order to support ODE systems where the access distance is limited only in a cyclic sense as it occurs in discretized PDE systems with periodic boundary conditions is possible but not discussed here.

5 Storage and Working Spaces

Sequential implementations store $k - 1$ function results and the two vectors $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$, which amounts to a total storage space of

$$S_{\text{seq}}(k, n) = (k + 1)n. \tag{3}$$

All parallel implementations store n/p components of the $k - 1$ function results per processing element. Differences in the storage space between the parallel implementations result from how $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ are handled. The general and specialized shared-address-space implementations with shared storage of $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ require the same storage space as sequential implementations:

$$S_{\text{sas,ys}}(p, k, n) = p \left\lceil (k - 1)\frac{n}{p} \right\rceil + 2n = (k + 1)n. \tag{4}$$

General MPI implementations store n/p components of $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ in local arrays, the pointers to which are swapped at every time step. Additionally, one array of size n to collect $\mathbf{y}_\kappa$ is used, which amounts to a total storage space of

$$S_{\text{das,mbcast}}(p,k,n) = p\left[(k-1)\frac{n}{p} + 2\frac{n}{p} + n\right] = (p+k+1)n. \tag{5}$$

Specialized implementations for distributed address space which use neighbor-to-neighbor communication and specialized implementations for shared address space which store $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ in a distributed fashion, need additional storage space, compared with the sequential implementations, only for the ghost cells:

$$\begin{aligned} S_{\text{das,single}}(p,k,n,d(\mathbf{f})) = S_{\text{sas,yd}}(p,k,n) &= p\left[(k-1)\frac{n}{p} + 2\left(\frac{n}{p} + 2d(\mathbf{f})\right)\right] \\ &= (k+1)n + 4d(\mathbf{f})p. \end{aligned} \tag{6}$$

Since all implementations iterate over all their data structures during one time step, the storage space they require per processing element constitutes the most significant working space of their loop structures. If the ODE system is large so that not all data used by a processing element per time step fits in the cache, the tiled or the pipeline-like loop structure can be expected to be more efficient, because they create additional smaller working spaces that allow temporal reuse of cache data.

The working space created by loop tiling, i.e., the size of one *tile*, consists of $k-1$ blocks of size B of the function results, one block of size B of $\mathbf{y}_{\kappa+1}$ computed in the current time step, and one block of size $B + 2d(\mathbf{f})$ of $\mathbf{y}_\kappa$ for which the function is evaluated:

$$W_{\text{tile}}(k, d(\mathbf{f}), B) = (k+1)B + 2d(\mathbf{f}). \tag{7}$$

The most important working space created by the pipeline-like loop structure using pipeline length L is the working space of one pipelining step, i.e., the computation of one diagonal consisting of L blocks. This working space can be viewed as being built up of L loop tiling working spaces, but components within the access distance of the function evaluation partially overlap. The resulting size of the working space is

$$W_{\text{pipe}}(k, d(\mathbf{f}), B, L) = L(k+1)B + 4d(\mathbf{f}). \tag{8}$$

6 Experimental Results and Discussion

6.1 Experimental Setup

In this article, we present selected experimental results measured on three computer systems. Sequential jobs needed for empirical search of optimal block sizes and pipeline lengths were run on a small cluster system consisting of 32 2-way AMD Opteron DP 246 nodes with 64 KB L1 data cache and 1024 KB L2 cache.

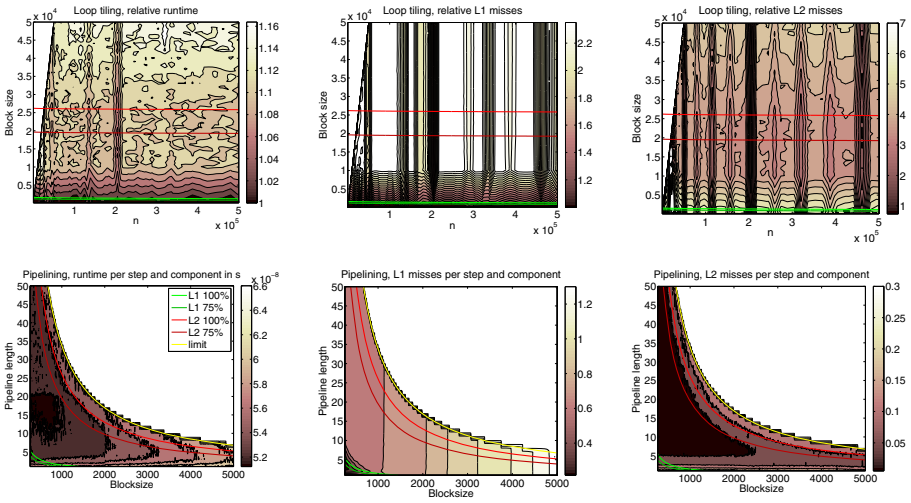

Fig. 4. Sequential runtime and locality behavior of the tiled and the pipeline-like loop structur on AMD Opteron DP 246. $k = 4$. Test problem: BRUSS2D-MIX. For the tiled loop structure, the problem size is varied. For the pipeline-like loop structure, results for problem size $N = 130$ ($n = 33\,800$) are shown. Dark color indicates better behavior.

To compare the sequential and parallel implementations, results measured on the two supercomputer systems JUROPA and HLRB II are shown. JUROPA (Jülich Supercomputing Centre (JSC)) consists of 2 208 compute nodes equipped with two quad-core Intel Xeon X5570 (Nehalem-EP) processors running at 2.93 GHz and interconnected by an Infiniband QDR network. The L1 data cache has a size of 32 KB; the L2 cache size is 256 KB. The L3 cache is shared between the four cores and has a size of 8 MB. HLRB 2 (Leibniz Supercomputing Centre (LRZ) Munich) is an SGI Altix 4700 system based on dual-core Itanium 2 9040 (Montecito) processors running at 1.6 GHz with a total number of 9 728 CPU cores. The system is interconnected by an SGI NUMAlink 4 network and is divided into 19 shared-memory partitions containing 512 cores. The L1 data cache has a size of 16 KB, but does not store floating point data. The L2 and the L3 cache have a size of 256 KB and 9 MB, respectively. All cache levels are non-shared.

The test problems considered are BRUSS2D-MIX (2D Brusselator reaction-diffusion equation [1,5]), which is derived from a first order 2D PDE system with two variables using an $N \times N$ grid and which has a system size of $n = 2N^2$ and an access distance of $d(\mathbf{f}) = 2N$, and STRING (mechanical vibration of a string [5]), which is derived from a second order 1D PDE system with one variable and which has a system size of $n = 2N$ and an access distance of $d(\mathbf{f}) = 3$.

6.2 Choosing Blocksize and Pipeline Length

For the tiled loop structure, the runtime depends on the block size B. The upper part of Fig. 4 illustrates the influence of the block size on the sequential runtime

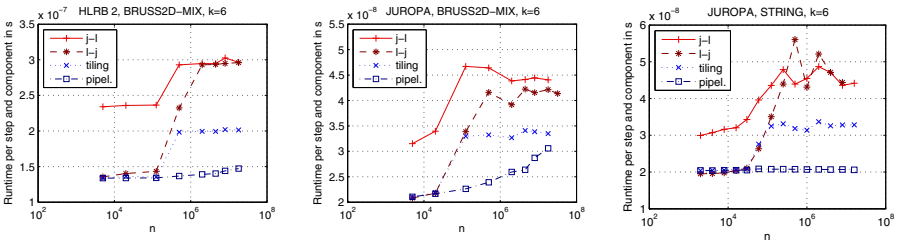

Fig. 5. Comparison of the normalized runtime of the sequential implementations

on an AMD Opteron DP 246 processor for the test problem BRUSS2D-MIX, $k = 4$ and varying system sizes between $n = 200$ and $n = 500\,000$. For this example, small block sizes up to ≈ 1000 deliver the smallest execution time. But the block size should not be too small, i.e., $B \gtrsim 100$, so that it spans several cache lines. This observation conforms with the working space model (7), which suggests maximum block sizes between 1630 ($n = 200$) and 1238 ($n = 500\,000$) for a tile to fit in the L1 cache. In fact, Fig. 4 shows that in the ranges of the block size with best execution times the smallest numbers of L1 misses occur.

The execution time of the pipeline-like loop structure is influenced by two parameters: the block size and the pipeline length. The lower part of Fig. 4 illustrates this influence for the test problem BRUSS2D-MIX with problem size $N = 130$ ($n = 33\,800$) and $k = 4$ on an AMD Opteron DP 246 processor. Though according to (8) it is possible to fit the working space of a pipelining step in the L1 cache using pipeline length 5 or smaller, best performance, in this example, is obtained for a pipeline length between about 10 and 20 and block sizes up to 1000. Generally, an area of good performance is framed by the working space model (8) applied to the size of the L2 cache, but within this area neither the block size nor the pipeline length should be chosen too large.

6.3 Influence of the Working Spaces on Sequential Performance

Figure 5 compares the sequential implementations on one processor core of HLRB II and JUROPA using normalized runtime, i.e., the execution time per step and component. For the loop tiling and the pipelining implementations, a set of block sizes and pipeline lengths were precomputed using their working space models, and the runtime of the best parameter choice is shown. Since the function evaluation costs per component for the two test problems are independent of the system size, an increase in the normalized runtime is usually caused by working spaces of loops growing larger than a cache level. For small system sizes, where all data structures used in a time step fit in the cache, general implementations can obtain a good performance. For larger system sizes, loop tiling or pipelining is required for best performance. Pipelining performs best in the range of system sizes where the pipelining working space (8) fits in the cache but the overall working space of a time step (3) is too large to fit in the cache.

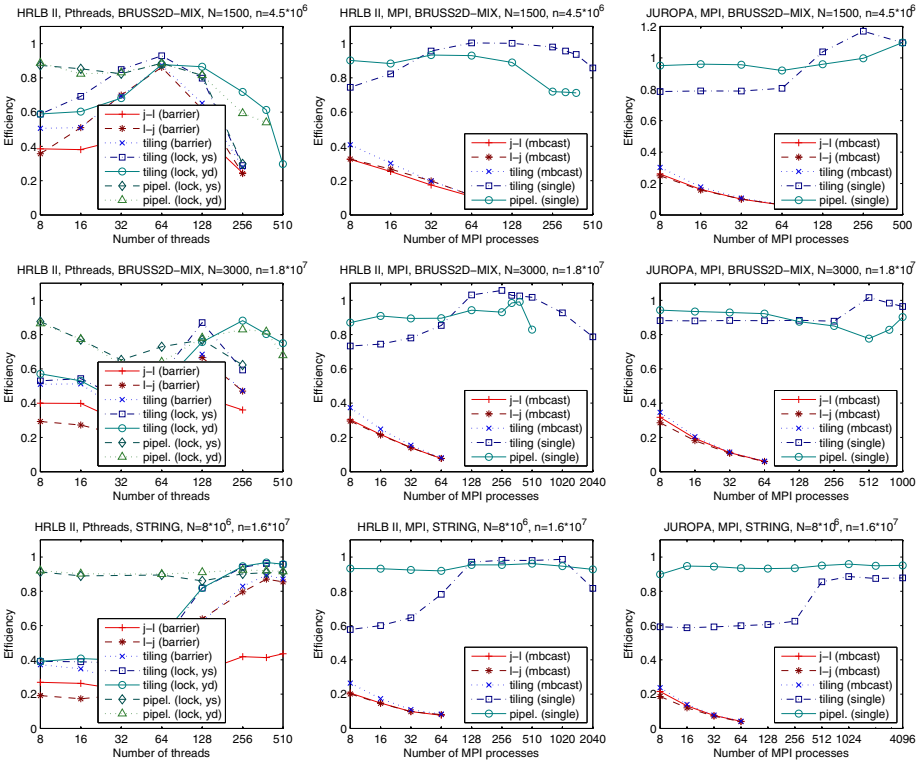

Fig. 6. Strong scalability of the Pthread and MPI implementations

6.4 Parallel Performance on Different Architectures

To investigate the strong scalability of the parallel implementations, Fig. 6 shows selected experimental results measured on HLRB II and JUROPA. The comparison is based on efficiency, i.e., execution time of the fastest sequential implementation divided by the execution time of the parallel implementation and the number of processing elements. Thus, an efficiency of 1 is optimal. A set of block sizes and pipeline lengths is precomputed and the best efficiency is shown, similarly to the comparison of the sequential implementations.

The general MPI implementations, which need to use `MPI_Allgatherv()`, do not scale. The MPI pipelining implementation obtains a very high, nearly constant efficiency (measured for STRING up to 4096 cores on JUROPA). A loop tiling implementation that exploits limited access distance by using single transfer operations is not as good as the pipelining implementation, in particular for small numbers of processor cores, but it catches up with or even outperforms the pipelining implementation as the number of processing elements is increased and the amount of data processed by each processor core per time step decreases.

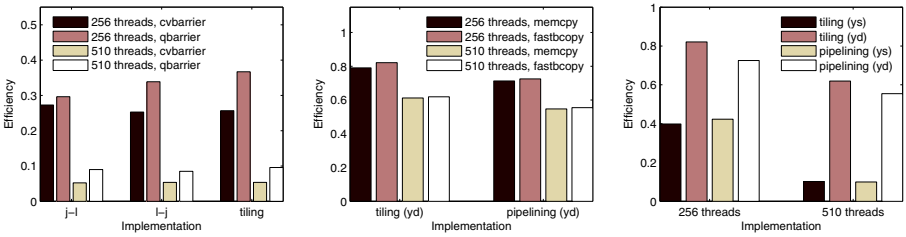

Fig. 7. Influences on the scalability of Pthread implementations on HLRB II. Test problem: BRUSS2D-MIX with $N = 2000$ ($n = 8 \cdot 10^6$). $k = 6$. Left: Barrier operations. Middle: Memory copy operations. Right: Distributed vs. shared storage.

The Pthread implementations could be investigated on HLRB II using up to 510 threads. To improve the performance of the general implementations, which need barrier operations, a barrier operation based on busy waiting [3] was used (cf. Fig. 7 (left)). In contrast to the general MPI implementations, the general Pthread implementations benefit from a parallel execution. Depending on the test problem and the problem size, even several hundred threads can be run efficiently. Using 256 threads and $k = 6$, speedups between 92 (j–l) and 121 (loop tiling) have been measured for BRUSS2D-MIX and $N = 3000$ and between 62 (j–l) and 72 (loop tiling) for BRUSS2D-MIX and $N = 1500$. Using 510 threads, for STRING and $N = 8 \cdot 10^6$, the general implementations even reached speedups between 222 (j–l) and 445 (loop tiling).

The specialized Pthread implementations are even more efficient than the general Pthread implementations but do not reach the performance of the specialized MPI implementations. In the example cases shown in Fig. 6 speedups between 346 and 383 have been measured for BRUSS2D-MIX and $N = 3000$ and between 464 and 489 for STRING using 510 threads. At least for BRUSS2D-MIX, the implementations with distributed storage of $\mathbf{y}_\kappa$ and $\mathbf{y}_{\kappa+1}$ are more efficient than those with shared storage of these vectors (cf. Fig. 7 (right)). To further improve these implementations, which have to copy data from neighbor threads, the `memcpy()` operation from the standard C library was replaced by the faster (internal) `fastbcopy()` operation of the SGI MPT library. This, however, increased efficiency only marginally (cf. Fig. 7 (middle)). As for the MPI implementations, pipelining is most efficient for smaller numbers of processing elements, where the amount of data processed by each processing element is larger than the cache.

7 Conclusions

Data-parallel implementations of Adams–Bashforth methods can be used efficiently on hundreds and thousands of processing elements if the ODE system is large enough. Since general MPI implementations require the use of multibroadcasts, only specialized MPI implementations which exploit the specific structure

of the ODE system can reach high speedups. Pthread implementations also can obtain significant speedups from a data-parallel execution, even for ODE systems with arbitrary coupling, but the performance of the specialized MPI implementations is higher. If the amount of data processed per processing element at each time step exceeds the cache size, locality optimizations such as loop tiling or pipelining are required for best performance. Pipelining of time steps, as proposed in this paper, outperforms standard loop tiling if the working space of a pipelining steps fits in the cache. Efficient block sizes and pipeline lengths can be chosen using a working space model. In future work, the most efficient implementation, block size and pipeline length could be chosen automatically.

Acknowledgments. We thank the JSC and the LRZ Munich for providing access to their supercomputer systems. This work was supported by the German Research Foundation (DFG) [grant numbers RA 524/17-1 and RA 524/17-2].

References

1. Burrage, K.: Parallel and Sequential Methods for Ordinary Differential Equations. Oxford University Press, New York (1995)
2. Butcher, J.C.: Numerical methods for ordinary differential equations, 2nd edn. John Wiley & Sons, Chichester (2008)
3. Chen, J., Watson III, W.: Software barrier performance on dual quad-core Opterons. In: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage, pp. 303–309. IEEE Computer Society (2008)
4. Cong, N.H., Xuan, L.N.: Twostep-by-twostep PIRK-type PC methods with continuous output formulas. J. Comput. Appl. Math. 221, 165–173 (2008)
5. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations I: Nonstiff Problems, 2nd rev. edn. Springer, Berlin (2000)
6. Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, 2nd rev. edn. Springer, Berlin (2002)
7. Korch, M., Rauber, T.W.: Locality Optimized Shared-Memory Implementations of Iterated Runge-Kutta Methods. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 737–747. Springer, Heidelberg (2007)
8. Korch, M., Rauber, T.: Parallel low-storage Runge-Kutta solvers for ODE systems with limited access distance. Int. J. High Perf. Comput. Appl. 25(2), 236–255 (2011)
9. Korch, M., Rauber, T., Scholtes, C.: Scalability and locality of extrapolation methods on large parallel systems. Concurrency Computat.: Pract. Exper. 23(15), 1789–1815 (2011)
10. Ley, K.: Parallele Implementierung und Analyse eines expliziten Adams-Verfahrens. Bachelor's thesis, University of Bayreuth (November 2010)
11. Rauber, T.W., Rünger, G.: Execution Schemes for Parallel Adams Methods. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 708–717. Springer, Heidelberg (2004)
12. Schmitt, B.A., Weiner, R., Jebens, S.: Parameter optimization for explicit parallel peer two-step methods. Appl. Numer. Math. 59, 769–782 (2008)
13. van der Houwen, P.J., Messina, E.: Parallel Adams methods. J. Comput. Appl. Math. 101, 153–165 (1999)