Scalability and Locality of Extrapolation Methods for Distributed-Memory Architectures

Matthias Korch, Thomas Rauber, and Carsten Scholtes

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

Abstract. The numerical simulation of systems of ordinary differential equations (ODEs), which arise from the mathematical modeling of time-dependent processes, can be highly computationally intensive. Thus, efficient parallel solution methods are desirable. This paper considers the parallel solution of systems of ODEs by explicit extrapolation methods. We analyze and compare the scalability of several implementation variants for distributed-memory architectures which make use of different load balancing strategies and different loop structures. By exploiting the special structure of a large class of ODE systems, the communication costs can be reduced considerably. Further, by processing the microsteps using a pipeline-like loop structure, the locality of memory references can be increased and a better utilization of the cache hierarchy can be achieved. Runtime experiments on modern parallel computer systems show that the optimized implementations can deliver a high scalability.

1 Introduction

Systems of ordinary differential equations (ODEs) arise from the mathematical modeling of time-dependent processes. We consider the numerical solution of initial value problems (IVPs) of systems of ODEs defined by

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

with $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$. The solution of such problems can be highly computationally intensive, and the utilization of parallelism is desirable, in particular if the dimension n of the ODE system is large. Parallel solution methods have been proposed by many authors, for example, extrapolation methods [4,5,8,14,11], waveform relaxation methods [2], iterated Runge–Kutta methods [9,7], and peer two-step methods [12]. An overview is given in [2].

Numerical solution methods for ODE IVPs start with the initial value $\mathbf{y}_0$, and perform a (potentially large) number of time steps to walk through the integration interval $[t_0, t_e]$. At each time step κ, a new approximation $\boldsymbol{\eta}_{\kappa+1}$ to the solution function $\mathbf{y}$ at time $t_{\kappa+1}$ is computed, i.e., $\boldsymbol{\eta}_0 = \mathbf{y}_0$, $\boldsymbol{\eta}_{\kappa+1} \approx \mathbf{y}(t_{\kappa+1})$. Sophisticated methods can adapt the stepsize $h_\kappa = t_{\kappa+1} - t_\kappa$ such that the number of time steps is reduced while the local error is kept below a user-defined error tolerance. The different solution methods are distinguished mainly by the computations performed at each time step.

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 65–76, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Extrapolation methods compute a sequence $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ of approximations to $\mathbf{y}(t_{\kappa+1})$ of increasing accuracy, which are then used to extrapolate $\boldsymbol{\eta}_{\kappa+1}$. In order to compute these r approximations, the IVP $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$, $\mathbf{y}(t_\kappa) = \boldsymbol{\eta}_\kappa$ on the interval $[t_\kappa, t_{\kappa+1}]$ is solved r times by a *base method* using an individual constant stepsize $h_\kappa^{(j)}$, $j = 1, \ldots, r$, where $h_\kappa^{(1)} > h_\kappa^{(2)} > \cdots > h_\kappa^{(r)}$. The time steps with stepsize $h_\kappa^{(j)}$ computed by the base method are called *local time steps* or *micro-steps*, whereas the time steps of the extrapolation method are referred to as *global time steps* or *macro-steps*. Often, one-step methods of order 1 or 2 are used as base method. The stepsizes $h_\kappa^{(j)}$ are determined by an increasing sequence of positive integer numbers $n_1 < n_2 < \cdots < n_r$, such that $h_\kappa^{(j)} = h_\kappa / n_j$ for $j = 1, \ldots, r$. In the experiments presented in this paper, we use the harmonic sequence $n_j = j$, which is presumed to deliver a better practical performance than other commonly used sequences [3] and which facilitates load balancing.

Explicit extrapolation methods use an explicit base method, e.g., the explicit Euler method (Richardson-Euler method) or the explicit midpoint rule (GBS method [1,6]). In this paper, we use the Richardson-Euler method, which computes the r approximations $\boldsymbol{v}^{(j)} = \boldsymbol{\mu}_{n_j}^{(j)}$, $j = 1, \ldots, r$, according to

$$\boldsymbol{\mu}_0^{(j)} = \boldsymbol{\eta}_\kappa \ , \quad \boldsymbol{\mu}_\lambda^{(j)} = \boldsymbol{\mu}_{\lambda-1}^{(j)} + h_\kappa^{(j)} \cdot \mathbf{f}\left(t_\kappa + (\lambda - 1) h_\kappa^{(j)}, \boldsymbol{\mu}_{\lambda-1}^{(j)}\right) \ , \quad \lambda = 1, \ldots, n_j \ . \tag{2}$$

Interpreting the sequence $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ as values of a function $\mathbf{T}(h)$ such that $\mathbf{T}\left(h_\kappa^{(j)}\right) = \boldsymbol{v}^{(j)}$, we use the Aitken–Neville algorithm

$$\mathbf{T}_{j,k+1} = \frac{h_{j-k} \cdot \mathbf{T}_{j,k} - h_j \cdot \mathbf{T}_{j-1,k}}{h_{j-k} - h_j} \ , \quad k = 1, \ldots, r-1, \quad j = k+1, \ldots, r \ , \tag{3}$$

starting with $\mathbf{T}_{j,1} = \mathbf{T}\left(h_\kappa^{(j)}\right)$ to extrapolate $\mathbf{T}(0) = \mathbf{T}_{r,r}$. This value is then used as the new approximation $\boldsymbol{\eta}_{\kappa+1}$ to start the next macro-step.

In the following, we investigate the scalability of different implementations of extrapolation methods for distributed-memory architectures. At first, in Section 2, we describe how sequential and parallel implementations can be derived from the mathematical formulations (2) and (3). Then, in Section 3, we show how the communication costs can be reduced and how the locality of memory references can be improved by exploiting the special structure of a large class of ODE IVPs. In Section 4, we evaluate the results of runtime experiments performed on two modern supercomputer systems to investigate the scalability of the previously described implementations. Section 5 concludes the paper.

2 Implementation of Extrapolation Methods for ODE Systems with Arbitrary Access Structure

In this section, we derive sequential and parallel distributed-memory implementations of extrapolation methods, using (2) and (3) as a starting point. The implementations derived in this section are suitable for general ODE systems consisting of n arbitrarily coupled equations.

// *computation of* $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ *in the registers* $\mathbf{R}_1, \ldots, \mathbf{R}_r$

1: **for** $(j = 1;\ j \le r;\ {+}{+}j)$ // *for each stepsize* $h_\kappa^{(j)} = h_\kappa / n_j$

2: **for** $(i = 1;\ i \le n;\ {+}{+}i)$ $\mathbf{R}_j[i] = \boldsymbol{\eta}_\kappa[i]$; // *use* $\boldsymbol{\eta}_\kappa$ *as initial value*

3: **for** $(\lambda = 1;\ \lambda \le n_j;\ {+}{+}\lambda)$ // n_j *micro-steps*

4: **for** $(i = 1;\ i \le n;\ {+}{+}i)$ $\mathbf{F}[i] = f_i(t_\kappa + (\lambda - 1)h_\kappa^{(j)}, \mathbf{R}_j)$; // *evaluate* $\mathbf{f}$

5: **for** $(i = 1;\ i \le n;\ {+}{+}i)$ $\mathbf{R}_j[i] \mathrel{+}= h_\kappa^{(j)} \mathbf{F}[i]$; // *update* $\mathbf{R}_j$

// *extrapolation according to Aitken–Neville*

6: **for** $(k = 2;\ k \le r;\ {+}{+}k)$ // $r - 1$ *rounds*

7: **for** $(j = r;\ j \ge k;\ {-}{-}j)$ // *in-place computation of* $\mathbf{T}_{j,k}$ *in register* $\mathbf{R}_j$

8: **for** $(i = 1;\ i \le n;\ {+}{+}i)$

9: $\mathbf{R}_j[i] = (h_\kappa^{(j-k)} \cdot \mathbf{R}_j[i] - h_\kappa^{(j)} \cdot \mathbf{R}_{j-1}[i]) / (h_\kappa^{(j-k)} - h_\kappa^{(j)})$;

10: accept or reject $\mathbf{R}_r$ as new approximation $\boldsymbol{\eta}_{\kappa+1}$ and select new stepsize $h_{\kappa+1}$;

Fig. 1. Sequential loop structure

2.1 Algorithmic Structure and Sequential Implementation

The computations defined by (2) and (3) suggest the subdivision of the algorithmic structure of a macro-step into two phases: the computation of the approximation values $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ and the subsequent extrapolation of $T(0)$ using the Aitken–Neville algorithm. Figure 1 shows a possible sequential loop structure implementing these two phases as separate loop nests. This loop structure is used as basis for deriving parallel implementations in Section 2.2.

In the first phase of a macro-step, the approximation values $\boldsymbol{v}^{(j)} \approx \mathbf{y}(t_{\kappa+1})$, $j = 1, \ldots, r$, are computed by solving the IVP $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$, $\mathbf{y}(t_\kappa) = \boldsymbol{\eta}_\kappa$ on the interval $[t_\kappa, t_{\kappa+1}]$ r times by the explicit Euler method (2) using an individual stepsize $h_\kappa^{(j)}$. To exploit the locality of successive micro-steps $\lambda \rightarrow \lambda + 1$, the outermost loop is iterated over the sequence index $j = 1, \ldots, r$ such that each iteration computes one sequence j of micro-steps.

The computation of one sequence of micro-steps makes use of two vectors of size n (i.e., arrays which can store n floating-point values, referred to as *registers*). After initializing the register $\mathbf{R}_j$ with the n components of the initial value $\boldsymbol{\eta}_\kappa$, a loop is executed to compute the micro-steps $\lambda = 1, \ldots, n_j$. At each micro-step, an evaluation of the right-hand-side function $\mathbf{f}$ and an update of $\mathbf{R}_j$ is required. Since, here, we make no assumptions about the structure of the ODE system, the evaluation of $f_i(t_\kappa + (\lambda - 1)h_\kappa^{(j)}, \mathbf{R}_j)$ may depend on all components of $\mathbf{R}_j$, and, therefore, the results of the function evaluations $f_1, \ldots, f_n$ have to be stored in a temporary register ($\mathbf{F}$) before the components of $\mathbf{R}_j$ can be updated in a subsequent loop over the system dimension. After the update, the temporary register $\mathbf{F}$ can be reused to compute another micro-step.

At the beginning of the second phase (the Aitken–Neville algorithm (3)) the registers $\mathbf{R}_j$, $j = 1, \ldots, r$, contain the r approximation values $\boldsymbol{v}^{(j)} = \mathbf{T}_{j,1}$. According to (3), the computation of $\mathbf{T}_{j,k}$ only depends on $\mathbf{T}_{j-1,k-1}$ and $\mathbf{T}_{j,k-1}$. Hence, $\mathbf{T}_{r,r}$ can be computed in-place by carefully choosing the update order of the registers in use. To do so, one iteration of the outer loop k (one *round*) overwrites $\mathbf{T}_{j,k-1}$ in $\mathbf{R}_j$ by $\mathbf{T}_{j,k}$ for all $j \geq k$. Starting the j-loop with $j = r$ and counting j backwards avoids overwriting values needed for the computation of the remaining $\mathbf{T}_{j',k}$ with $j' < j$. Respecting these dependencies, all components of $\mathbf{R}_j$ can be updated in-place according to (3). After $r - 1$ rounds, the final result of the Aitken–Neville extrapolation, $\mathbf{T}_{r,r} = \boldsymbol{\eta}_{\kappa+1}$, resides in $\mathbf{R}_r$ and can be accepted or rejected by the stepsize controller.

2.2 Exploiting Data and Task Parallelism

Compared to classical Runge–Kutta methods, extrapolation methods provide a larger potential for parallelism. In addition to the potential for *parallelism across the system* (data parallelism), which increases with the dimension of the ODE system and which is available in all ODE solution methods, extrapolation methods can exploit *parallelism across the method* (task parallelism).

The following description of parallelization approaches assumes the use of a message-passing programming model, in particular MPI [13].

Consecutive Implementation. In a purely data-parallel implementation, the i-loops are parallelized such that the n components of the ODE system are distributed evenly across all P processors and each processor is responsible for the computation of one contiguous block of components of size n/P. As in the sequential implementation, the sequences of micro-steps $j = 1, \ldots, r$ as well as the micro-steps of these sequences $\lambda = 1, \ldots, n_j$ are computed consecutively to exploit the locality of successive micro-steps. Therefore, we refer to this implementation as *consecutive implementation.* Since the equations of the ODE system may be coupled arbitrarily, a processor has to provide the part of the current approximation vector $\boldsymbol{\mu}_\lambda^{(j)}$ it computes to all other participating processors before the next micro-step can be started. This requires the use of a multi-broadcast operation (`MPI_Allgather()`). Similarly to the computation of the micro-steps, the Aitken–Neville algorithm can be executed in data-parallel style using the same blockwise data distribution of n/P contiguous blocks of components to the P processors. During the execution of the Aitken–Neville algorithm, no communication is necessary. Only one further communication operation, a multi-accumulation operation (`MPI_Allreduce()`), needs to be executed near the end of the macro-step to enable stepsize control.

Group Implementations. As the r sequences of micro-steps within one macro-step are independent of each other, extrapolation methods provide potential for task parallelism. It can be exploited by parallelizing across the j-loop (line 1 of Fig. 1), i.e., by partitioning the processors into up to r disjoint groups where each group is responsible for the computation of one or several sequences of

micro-steps. Since these sequences consist of different numbers of micro-steps, load balancing is required for the group implementations. We consider two load balancing strategies:

- *Linear group implementation*: The processors are partitioned into r groups, and the number of processors per group is chosen to be linearly proportional to the number of micro-steps computed.
- *Extended group implementation*: The processors are partitioned into $r/2$ groups of equal size. Each group is responsible for the computation of (up to) two sequences of micro-steps, such that the total number of micro-steps per group is (nearly) evenly balanced. (For r odd, only one sequence of micro-steps can be assigned to one of the groups.) In case of the harmonic sequence, which we use in our experiments, and r even, group g computes $\boldsymbol{v}^{(g)}$ and $\boldsymbol{v}^{(r+1-g)}$, which leads to a total number of $r+1$ micro-steps per group.

As in the sequential and in the consecutive implementation, the groups compute the micro-steps of a sequence j one after the other to maintain locality. If a group is responsible for two sequences j and j', the two sequences are computed consecutively. Within a group consisting of G processors, work is distributed to the processors in a similar blockwise fashion as in the consecutive implementation such that each processor in the group is assigned n/G components of the ODE system. As in the consecutive implementation, multi-broadcast operations have to be executed between successive micro-steps, but data has to be exchanged only between processors belonging to the same group. Thus, less processors participate in a multi-broadcast operation, and multiple multi-broadcast operations are executed in parallel.

Compared to the consecutive implementation, the group implementations have the difficulty that the data distribution resulting from the task-parallel computation of $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ is not optimal for the Aitken–Neville algorithm, since each processor only has a subset of the components of one or two of the vectors $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ in its memory. Moreover, in the linear group implementation the groups differ in size, and the processors store different numbers of components of the vectors.

In order to avoid a complex and, thus, time consuming reorganization of the data distribution, we chose to gather all components of $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ at one processor (the leader) of the group responsible for the respective vector. The group leaders then execute the Aitken–Neville algorithm. This still involves a significant amount of communication: Since the vectors $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(r)}$ are distributed among the groups, each group leader has to send all components of one of its vectors, say $\mathbf{T}_{j-1,k}$, to the leader of the group which stores $\mathbf{T}_{j,k}$ and is responsible for the computation of $\mathbf{T}_{j,k+1}$. This has to be done once each round and can be implemented by single-transfer operations (`MPI_Send()`, `MPI_Recv()`). After all $r-1$ rounds, the processor which has computed the result $\mathbf{T}_{r,r}$ broadcasts it to all other processors using `MPI_Bcast()`.

3 Implementations Specialized in ODE Systems with Limited Access Distance

In the following, we show how the performance can be improved by exploiting the specific access structure of a large class of ODE systems.

3.1 Reducing Communication Costs

In case of general ODE systems, where the equations may be coupled arbitrarily, we have to assume that the evaluation of a component function $f_i(t, \mathbf{y})$ requires all components of the argument vector $\mathbf{y}$. General implementations of extrapolation methods therefore have to exchange the current approximation vector $\boldsymbol{\mu}_\lambda^{(j)}$ by a multi-broadcast operation once per micro-step. Consequently, general implementations cannot be expected to reach a high scalability, since the execution time of multi-broadcast operations increases with the number of processors.

There is, however, a large class of ODE systems where this multi-broadcast operation can be avoided. This includes ODE systems where the components of the argument vector accessed by each component function f_i lie within a bounded index range near i. Many sparse ODE systems, in particular many ODE systems resulting from the spatial discretization of partial differential equations (PDEs) by the method of lines, can be formulated such that this condition is satisfied. To measure this property of a function $\mathbf{f}$, we use the *access distance* $d(\mathbf{f})$ which is the smallest value b, such that all component functions $f_i(t, \mathbf{y})$ access only the subset $\{y_{i-b}, \dots, y_{i+b}\}$ of the components of their argument vector $\mathbf{y}$. We say the access distance of $\mathbf{f}$ is *limited* if $d(\mathbf{f}) \ll n$.

If the right-hand-side function $\mathbf{f}$ has a limited access distance, a processor only needs to exchange data with the two processors responsible for those $2 \cdot d(\mathbf{f})$ components immediately adjacent to his own set of components, i.e., with its two neighbor processors, to meet the dependencies of the function evaluation. (In our implementations, we assume that each processor is at least responsible for $2 \cdot d(\mathbf{f})$ components.) A similar approach has been followed in [5] to devise efficient linearly-implicit extrapolation algorithms.

A limited access distance not only allows the replacement of the expensive multi-broadcast operation by scalable neighbor-to-neighbor communication, it is furthermore possible to partially overlap the data transfer times with computations: While a processor is waiting for the incoming data from the neighboring processors, it can already evaluate those component functions which do not depend on the incoming data. These optimizations can be applied to all three general implementations introduced in Section 2.2.

3.2 Optimization of the Locality Behavior

If the right-hand-side function $\mathbf{f}$ has a limited access distance, the working space of the micro-steps can be reduced by a loop interchange of the i-loop and the λ-loop. Instead of processing all components $i = 1, \dots, n$ in the innermost loop, we can switch over to a block-based computation order using blocks of size $B \geq d(\mathbf{f})$

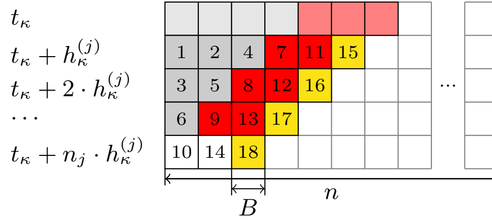

Fig. 2. Pipelined computation of the micro-steps

such that the ODE system is subdivided into $n_B = n/B$ contiguous blocks. Given this subdivision, the function evaluation of a block $I \in \{2, \ldots, n_B - 1\}$ only depends on the blocks $I - 1$, I and $I + 1$ of the previous micro-step. This enables a pipeline-like computation of the micro-steps as illustrated in Fig. 2.

The figure shows the pipelined computation of $\boldsymbol{v}^{(j)}$ for the case $n_j = 4$. Each box represents a block of B components at a certain micro-step. The time indices of the micro-steps proceed from top to bottom and are displayed at the left of each row, while the system dimension runs from left to right. The top row represents $\boldsymbol{\eta}_\kappa$ and is used as input to the first micro-step. The blocks are numbered in the order they are computed. The blocks on the diagonal (15, 16, 17, 18) are to be computed in the current pipelining step. The three blocks above each one of the blocks on this diagonal are needed as input to evaluate the right-hand-side function on the respective block. For example, the blocks with numbers 8, 12, and 16 are needed as input to compute the block with number 17. Numbered white blocks represent final values of the components of $\boldsymbol{v}^{(j)}$ at micro-step $t_\kappa + n_j \cdot h_\kappa^{(j)} = t_\kappa + h_\kappa$. All other numbered blocks have been used in previous pipelining steps, but are not needed to compute the current or subsequent pipelining steps. After the blocks on the current diagonal have been computed, all dependencies of the blocks on the next diagonal are satisfied. Thus, we can continue with the computation of one diagonal after the other until all blocks of $\boldsymbol{v}^{(j)}$ have been computed.

Using this computation order, the working space of one pipelining step contains only $3 \cdot n_j + 1$ blocks of size B, which is usually small enough to fit in the cache. For very large sequences of Euler steps, such as they are common in many PDE solvers where they are used for global time stepping, spanning a pipeline across all time steps would be inefficient. For such codes, diamond-like tile shapes as proposed in [10] lead to a higher performance.

We have implemented pipelining variants of the sequential implementation, the consecutive implementation, the linear group implementation, and the extended group implementation. Since the pipelining computation order requires a limited access distance as does the optimized communication pattern proposed in Section 3.1, the optimized communication pattern is used in all parallel pipelining implementations. Thus, the amount of data exchanged in the pipelining implementations is equal to that in the implementations from Section 3.1. During the initialization and during the finalization of the pipeline, communication with one

of the neighboring processors is required. In order to match these communication needs, pipelines running in opposite directions on processors with neighboring sets of components are used.

4 Experimental Evaluation

The scalability of the parallel implementation variants has been investigated on two supercomputer systems, JUROPA and HLRB 2. HLRB 2 is an SGI Altix 4700 based on Intel Itanium 2 (Montecito) dual-core processors running at 1.6 GHz, which are interconnected by an SGI NUMAlink network, totaling 9 728 CPU cores. The system is divided into 19 partitions with 512 cores each. JUROPA consists of 2208 compute nodes equipped with two quad-core Intel Xeon X5570 (Nehalem-EP) processors running at 2.93 GHz resulting in a total number of 17 664 cores. An Infiniband network interconnects the nodes. The sequential implementations have been investigated, additionally, on a third system equipped with AMD Opteron 8350 (Barcelona) quad-core processors running at 2.0 GHz. The implementations have been developed using C with MPI. As compilers we used GCC 4.1.2 on HLRB 2, GCC 4.3.2 on JUROPA, and GCC 4.4.1 on the Opteron system. The MPI libraries used on the two supercomputer systems were SGI MPT 1.22 on HLRB 2 and MPICH2 1.0.8 on JUROPA.

As an example problem we use BRUSS2D [2], a typical example of a two-dimensional PDE discretized by the method of lines. The spatial discretization of this problem on a $N \times N$ grid leads to an ODE system of size $n = 2N^2$. The components of the ODE system can be ordered such that a limited access distance of $d(\mathbf{f}) = 2N$ results. We use $B = d(\mathbf{f}) = 2N$ as the block size for the specialized implementations. This is the smallest possible choice and keeps the size of the working space at a minimum. On the other hand, this value is also large enough to exploit spatial locality of memory references. In all experiments presented in the following, we compute $r = 4$ sequences of micro-steps, and we use the harmonic sequence $n_j = j$ to determine the number of micro-steps in each sequence $j \in \{1, \ldots, 4\}$. As weight of the diffusion term, we use $\alpha = 2 \cdot 10^{-3}$.

4.1 Sequential Performance and Locality of Memory References

First, we investigate the two sequential implementations, i.e., the general implementation and the specialized pipelining implementation. Since an increase of the system size leads to a larger working space and thus to a different utilization of the cache hierarchy, we normalize the execution time by the number of macro-steps and by the size of the ODE system and plot the resulting normalized execution time against the size of the ODE system (Fig. 3). Consequently, every increase or decrease of the normalized execution time corresponds to an increase or decrease of the average execution time of memory operations and thus to an increase or decrease of the number of cache misses per work unit. The overhead incurred by instructions independent of the system size (such as the stepsize control mechanism) usually is negligible if the ODE system is sufficiently large.

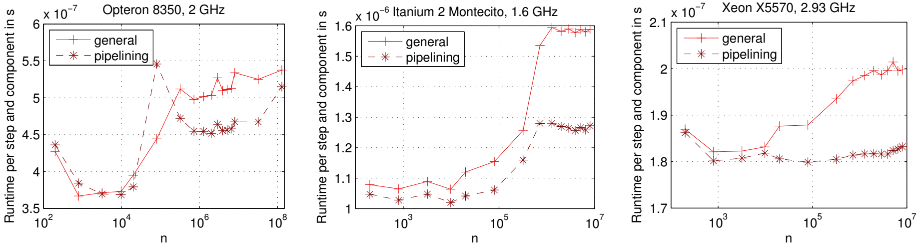

Fig. 3. Comparison of the sequential implementations

Except for very small system sizes, the normalized execution time is lower for smaller system sizes, and it increases when the system size is increased and thus the working space of a significant loop outgrows the size of one of the cache levels. The largest increase of the normalized execution time can be observed when the size of two registers starts exceeding the size of the highest level cache.

The Opteron processor has the smallest total cache size of the three target platforms considered. The L3 cache has a size of 2 MB, the L2 cache has a size of 512 KB, and the L1 data cache has a size of 64 KB. Up to $n \approx 3.3 \cdot 10^4$, two registers fit in the L2 cache. In this situation, the normalized execution times of the general and the pipelining implementation are nearly equal and reach their minimal value of $\approx 3.7 \cdot 10^{-7}$ s. If n increases above $3.3 \cdot 10^4$, the normalized execution times of both implementations increase rapidly. For $n \gtrsim 1.6 \cdot 10^5$ two vectors exceed the sum of the sizes of the L2 and the exclusive L3 cache. If n is increased beyond this value, the growth of the normalized execution times slows down to nearly 0. In this situation, the pipelining implementation runs about 10 % faster than the general implementation, because the working space of the general implementation no longer fits in the L3 + L2 cache while the working space of the pipelining implementation is still small enough. The working space of the pipelining implementation outgrows the cache size for $n \gtrsim 3 \cdot 10^8$. Figure 3 includes normalized execution times for $n \leq 1.28 \cdot 10^8$. While at this point the working space of the pipelining implementation still fits in the cache, for $n = 1.28 \cdot 10^8$ the normalized execution time of the pipelining implementation already starts growing and nearly approaches that of the general implementation.

On the Itanium 2 and the Xeon processor we observe a similar behavior as on the Opteron processor. Though on these two processors the pipelining implementation is faster than the general implementation for all system sizes, the difference is not very large as long as two registers fit in the 256 KB L2 cache, which is the case for $n \approx 1.6 \cdot 10^4$. For larger values of n, the normalized execution times of both implementations grow rapidly on the Itanium 2 processor until $n \approx 5.9 \cdot 10^5$ when two registers no longer fit in the 9 MB L3 cache. Then, the normalized execution times remain almost constant up to the largest system size considered in this experiment. In this situation where the working space of the pipelining implementation still fits in the cache, the pipelining implementation is nearly 20 % faster than the general implementation. On the Xeon processor

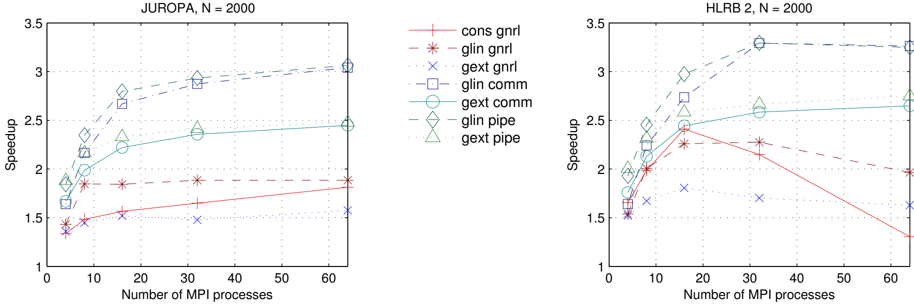

Fig. 4. Comparison of the general implementations and the specialized group implementations

the normalized execution time of the pipelining implementation increases only marginally when two registers outgrow the L2 cache. But the curve of the general implementation shows a step at this event, and it increases further for $n \gtrsim 10^5$ until $n \approx 2 \cdot 10^6$, because for $n \gtrsim 5.2 \cdot 10^5$ the 8 MB L3 cache is too small to store two registers completely, and the registers can only be partially reused by subsequent micro-steps. For values of n in this range, the pipelining implementation is about 8.5 % faster than the general implementation.

4.2 Parallel Speedups and Scalability

To investigate the parallel performance of the implementations, we consider the parallel speedups and the parallel efficiency on HLRB 2 and JUROPA with respect to the fastest sequential implementation on the respective machine. For the problem sizes $N = 1000$ and $N = 2000$ considered in this paper, the fastest sequential implementation on both systems is the pipelining implementation.

As shown in Fig. 4, the consecutive general implementation (cons gnrl) and the group implementations (glin, gext) cannot obtain a high scalability. The scalability of all general implementations (gnrl) is limited by the multi-broadcast operation. The specialized group implementations reach slightly higher speedups, but the speedups are also determined by the number of group leaders which participate in the Aitken–Neville algorithm. Consequently, the speedups of the linear group implementations (glin) are higher than the speedups of the extended group implementations (gext), since a larger number of groups is used.

Figure 5 shows speedups and efficiencies of the specialized consecutive implementation with unmodified loop structure and the consecutive pipelining implementation. Both can take advantage of neighbor-to-neighbor communication and obtain a high scalability. For large numbers of cores (Fig. 5, left), no significant difference in the parallel performance of the two implementations can be observed, because the working space of both implementations decreases reciprocally proportional to the number of cores. Thus, even the working space of the unmodified loop structure fits in the cache if the number of cores used is

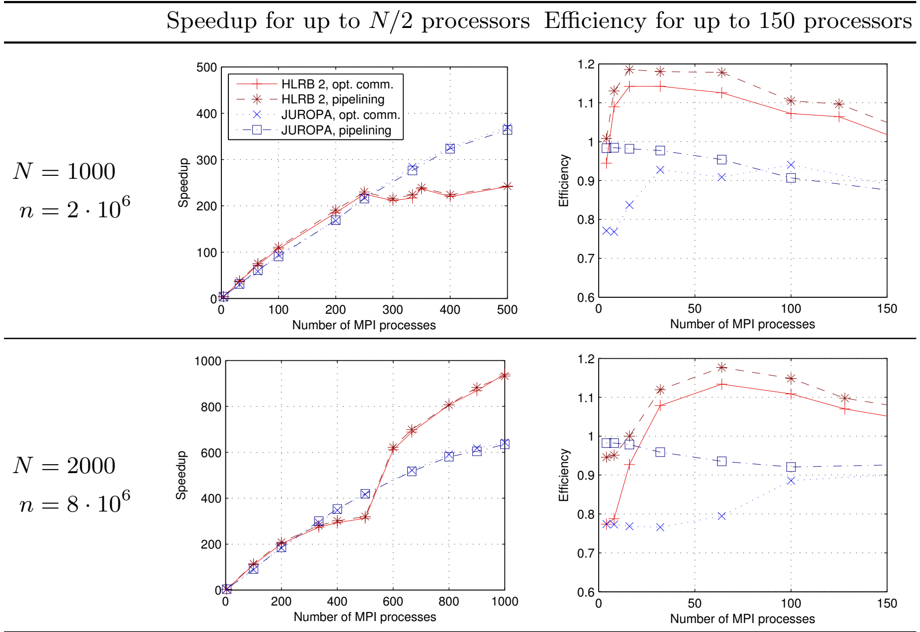

Fig. 5. Comparison of the specialized consecutive implementations

sufficiently large. On HLRB 2 a single partition was used for less than 512 MPI processes. For larger runs, processes were distributed evenly to two partitions. Using this strategy, we observe a decrease in the performance when the number of MPI processes per partition comes close to the partition size.

For small numbers of cores (Fig. 5, right), the decrease of the working space due to parallelization can produce superlinear speedups, since distributing the data to more processors has a similar effect on the computation time per component as decreasing the system size in a sequential run. On HLRB 2 the computation time per component for the system sizes $n = 2 \cdot 10^6$ and $8 \cdot 10^6$ are significantly larger than those for $n \lesssim 5.9 \cdot 10^5$ (cf. Fig. 3). Thus, superlinear speedups occur for $n/p \lesssim 5.9 \cdot 10^5$. On JUROPA no superlinear speedups are observed, because the normalized sequential runtime of the fastest implementation depends only marginally on the number of components to be processed. On both systems, the efficiency of the pipelining implementation is significantly higher than that of the unmodified loop structure for small numbers of cores.

5 Conclusions

We have investigated the scalability and the locality of several implementations of explicit extrapolation methods for ODE IVPs. The implementations exploit pure data parallelism, or mixed task and data parallelism combined with two

different load balancing strategies. Further, we have compared a general implementation suitable for ODE systems consisting of arbitrarily coupled equations with a pipeline-like implementation applicable to right-hand-side functions with limited access distance. We have exploited this limited access distance to replace expensive global communication by neighbor-to-neighbor communication.

Runtime experiments on two modern supercomputer systems show that a high scalability is possible if expensive global communication can be avoided by exploiting the special structure of the right-hand-side function. A pipeline-like loop structure can improve the performance of sequential and moderately-sized parallel runs significantly. For large numbers of processors, the amount of data handled per processor is only small and fits in the cache, and the influence of the loop structure on the performance is less significant.

Acknowledgments. We thank the Jülich Supercomputing Centre and the Leibniz Supercomputing Centre Munich for providing access to their systems.

References

1. Bulirsch, R., Stoer, J.: Numerical treatment of ordinary differential equations by extrapolation methods. Numer. Math. 8, 1–13 (1966)
2. Burrage, K.: Parallel and Sequential Methods for Ordinary Differential Equations. Oxford University Press, New York (1995)
3. Deuflhard, P.: Order and stepsize control in extrapolation methods. Numer. Math. 41, 399–422 (1983)
4. Deuflhard, P.: Recent progress in extrapolation methods for ordinary differential equations. SIAM Rev. 27, 505–535 (1985)
5. Ehrig, R., Nowak, U., Deuflhard, P.: Massively parallel linearly-implicit extrapolation algorithms as a powerful tool in process simulation. In: Parallel Computing: Fundamentals, Applications and New Directions, pp. 517–524. Elsevier, Amsterdam (1998)
6. Gragg, W.B.: On extrapolation algorithms for ordinary initial value problems. SIAM J. Numer. Anal. 2, 384–404 (1965)
7. van der Houwen, P.J., Sommeijer, B.P.: Parallel iteration of high-order Runge–Kutta methods with stepsize control. J. Comput. Appl. Math. 29, 111–127 (1990)
8. Lustman, L., Neta, B., Gragg, W.: Solution of ordinary differential initial value problems on an Intel Hypercube. Computer Math. Appl. 23(10), 65–72 (1992)
9. Nørsett, S.P., Simonsen, H.H.: Aspects of parallel Runge–Kutta methods. In: Numerical Methods for Ordinary Differential Equations. LNM, vol. 1386, pp. 103–117 (1989)
10. Orozco, D., Gao, G.: Mapping the FDTD application to many-core chip architectures. In: Int. Conf. on Parallel Processing (ICPP-2009). IEEE, Los Alamitos (2009)
11. Rauber, T., Rünger, G.: Load balancing schemes for extrapolation methods. Concurrency: Pract. Ex. 9(3), 181–202 (1997)
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. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI the complete reference, 2nd edn. MIT Press, Cambridge (1998)
14. van der Houwen, P.J., Sommeijer, B.P.: Parallel ODE solvers. In: ACM Int. Conf. on Supercomputing, pp. 71–81 (1990)