DOI: 10.15439/2018F169
ISSN 2300-5963 ACSIS, Vol. 15

On the Autotuning Potential of Time-stepping methods from Scientific Computing

Natalia Kalinnik[1], Robert Kiesel[2], Thomas Rauber[1], Marcel Richter[2] and Gudula Rünger[2]

[1] University Bayreuth
Email: {natalia.kalinnik,rauber}@uni-bayreuth.de

[2] Chemnitz University of Technology
Email: {robert.kiesel,m.richter,ruenger}@cs.tu-chemnitz.de

***Abstract*—Due to the ever changing characteristics of the newly provided hardware, there is the permanent requirement of designing and re-designing software adequately to meet the basic hardware conditions. Especially for well-established software, easy portability of the functional as well as the non-functional properties, such as runtime performance or energy efficiency, would be beneficial, so that the software adapts automatically to the given hardware conditions. In this article, we explore the autotuning potential of several methods from scientific computing. In particular, we consider time-stepping methods and investigate the effect of relevant tuning parameters of the different methods. We also address the question, whether offline or online autotuning approaches are appropriate for the specific method. The methods from scientific computing considered are particle simulation methods, solution methods for differential equations, as well as sparse matrix computations.**

***Index Terms*—offline and online autotuning, performance analysis, portability of efficiency, time-stepping methods.**

I. Introduction

IT IS a well-known fact that the life-span of software is usually much longer than the life-span of the hardware on which it is executed. The common practice is to port and tune the existing software for the new hardware generation by adapting it to the hardware details of the new architecture. Especially, if the software is a complex software system developed over many years, the effort of porting and tuning is high, but the development of a completely new software system would also be too time-consuming. Because of the advent of heterogeneous hardware platforms, the effort for porting, re-designing, tuning or re-developing software is even increasing. The challenge and a major research question is how complex software systems can be developed such that the runtime behavior of the software system can adapt or can be adapted to the ever-changing hardware of a varying heterogeneous type. Specifically, we consider the question whether and how software can gain portable efficiency by self-adaptation, also called autotuning.

In this article, we investigate the tuning potential of important simulation methods from scientific computing and address the question, which techniques can be used to support the tuning towards the development of flexible complex software systems. The simulation methods considered include sparse matrix computations, particle simulation methods, and solution methods for ordinary differential equations. The article provides a systematic investigation of the potential for self-adaptation towards a better runtime performance using offline and online autotuning.

Offline autotuning is performed in a separate offline phase, which is executed at software installation time before the actual software execution and in which the runtime behavior of the software on the given architecture is explored by detailed performance tests with different input scenarios. The test results are used to generate one or several implementations of the software that run efficiently on the given hardware platform for different input sets. After the generation of the software, no further adaptation is performed during production runs. Typical examples for offline autotuning are ATLAS [1] and PHiPAC [2] for dense matrix multiplication. Offline autotuning can be applied if the runtime behavior depends only or mainly on the size of the input set and other characteristics of the input set play only a minor role.

Online autotuning is integrated into the execution of the software. The software observes the performance behavior for the given input set during the runtime and adapts its behavior such that the performance is increased as much as possible. Thus, online autotuning is able to adapt the software behavior to the characteristics of the input set. Examples for online autotuning approaches are Active Harmony [3] or Periscope [4]. The challenge of online autotuning is to integrate the self-adaptation at runtime in such a way that the runtime performance is affected as little as possible.

This article investigates the autotuning potential of simulation methods from scientific computing and discusses the usage of offline and online autotuning approaches. In particular, we provide a detailed performance analysis of the different simulation methods, investigate relevant parameters which have a large influence on the performance, and analyze whether the parameters identified are amenable to autotuning and which autotuning methodology is suitable. Depending on the method from scientific computing and the parameters identified, different autotuning approaches are best suited for different methods. The article derives a guideline, which method requires which degree for offline and online autotuning.

The rest of the article is structured as follows. Section II gives an overview of relevant aspects of autotuning approaches. Section III considers applications from different areas and discusses their autotuning potential. Section IV discusses related work. Section V summarizes the observations for the different applications and concludes the article.

II. Overview of Autotuning Approaches

In this section, we give an overview of key terms and techniques that are important for applying autotuning for time-stepping simulation methods.

A. Key aspects of self-adaptation

The main terms portability of efficiency and auto-tuning or self-adaptive software are often used as keywords today and we start by giving a precise definition for each of the terms.

An important goal in parallel scientific computing is *portability of efficiency* of simulation algorithms. A performance portable implementation of an application or algorithm can be defined as one that will achieve high performance across a variety of target systems [5]. Depending on the target system, high performance may be quite different, and the definition means high performance for each specific target system. For heterogeneous resources, a major challenge is the diversity of devices on different machines, which provide widely varying performance characteristics [6]. A program optimized for one processor processor may not run as well on the next generation of processors or on a device from a different vendor, and a program optimized for GPU execution is often very different from one optimized for CPU execution.

Adaptation and self-adaptation of software in general refers to the the ability of self-adjustment or self-modification of the software in accordance with changing conditions of environment or structure. This term has first been explored in [7]. Thus, self-adaptive software evaluates its own behavior during execution and changes its behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when a better functionality or performance seems to be possible [8].

There are different levels on which software can be tuned or influenced towards a better performance:

- hardware-level: this includes hardware techniques such as dynamic voltage and frequency scaling (DVFS) for energy optimization;
- system-level: this includes scheduling approaches by a compiler or operating system to improve task executions on a parallel target system;
- software-level: this includes modifications at the software-level towards better performance, e.g., source-code transformations such as loop transformations; this captures also the case that the software can adapt itself using knowledge from earlier computing steps.

There are a variety of interactions between these different levels and the final performance improvements achieved for a specific situation may come from performance improvements at different levels. The main focus of this article are performance improvements at the software-level and the questions which methods of self-adaptation are suitable for which application areas.

B. Tuning parameters and possibilities

The efficiency of an application software can depend on many influencing parameters and program transformations, system parameters and characteristics of the input set may have a large influence on the resulting runtime performance or energy efficiency. In this subsection, we provide a short overview.

A new implementation version for a simulation method from scientific computing can be generated by applying correctness-preserving program transformations such as loop interchange, loop distribution, loop unrolling or loop tiling. Moreover, SIMD instruction or memory prefetching techniques can be applied. The result is an implementation version with the same input-output behavior but potentially improved non-functional properties according to a given optimization goal.

Some of the program transformations may be based on the use of transformation parameters. Examples are loop unrolling or loop tiling where an unroll factor or a blocking factor needs to be specified. The selection of a suitable set of program transformations along with their parameter values and an order in which the program transformations should be applied may significantly increase the performance or the energy efficiency. The application of the program transformation could be controlled by a performance model such as the ECM model [9].

The execution of a program implementation may also be influenced by configuration and system parameters. These include the usage of compiler options, the number of threads or processes used for the execution of the implementation and the mapping of these threads or processes to the resources of the given HPC system. For the case of energy efficiency as target function, the selection of the operational frequency for DVFS may also play an important role. For some of the simulation methods, also characteristics of the input set may play an important role for the resulting performance or energy efficiency. Examples are signal processing where the size of the input set may determine which algorithm is the most efficient one [10], [11] or spare linear algorithms where the sparsity and composition of the data structures such as vectors or matrices may play an important role [12], [13]. This behavior can also be observed for particle simulation methods that are considered in this article, where the initial distribution of the particles may have a strong influence on which simulation method and which implementation leads to the fastest execution.

C. Offline and Online Autotuning approaches

The main emphasis of this article are methods of self-adaptation for the application class of time-stepping simulation methods from scientific computing, since these methods constitute an important class of HPC software. Depending on the characteristics of the specific simulation method and the underlying simulation algorithm, offline or online approaches may be suitable for self-adaptation. In the following, we give an overview.

1) Offline Autotuning: Analyzing the hardware and software for tuning and preselecting parameter sets are main aspects of the offline autotuning. The offline autotuning is performed before the first time step is executed and is performed once.

In this autotuning approach there should be an automated generation of test parameter sets for the identification of the influence of various platform and input parameters, e.g., processor frequency or number of threads. For the quantification of the influences, these parameters will be evaluated and analyzed. This evaluation and analysis can be done with some microbenchmarks. In the offline autotuning approach is also a creation of application-specific performance models, e.g., the Roofline model [14] or the ECM model[9], for the simulation application which can provide additional information. These performance models are evaluated by experiments and facilitate a selection of suitable program variants and configurations regarding all available program variants. The selected program variants are arranged in a decision tree which is provided to the online tuning step, whereby the leaves of the decision tree contain implementation variants considered for the execution and the inner nodes contain conditions to decide which leaves are appropriate. The conditions capture relevant information about the input data that is suitable to identify implementation variants than potentially lead to good runtime results under the conditions given. Depending on the application, the conditions may include the size of the input data and specific properties of the input data such as distribution characteristics for particle simulation methods, distribution patterns for sparse matrix computations, or access distances for solution methods for ordinary differential equations.

2) Online Autotuning: The monitoring and control of the self-adapting behaviour of time-stepping simulation methods are key components of the online autotuning approach. Many mechanisms should be provided to ensure a comprehensive online adaptation process for different simulation methods, which differ significantly in their computational structure, e.g. data arrangement or loop structures. Hence, the following online tuning mechanisms require to be as much application independent as possible in order to be applied to diverse time-stepping simulation methods.

The offline autotuning step pre-selected a diverse set of implementation variants that should be considered for the execution of the time-stepping simulation method. These implementation candidates are processed by a selection and preparation mechanism which ensures a later usage of these candidates. Architectural and algorithmic parameters are determined by an evaluation mechanism and include major properties of the actual input data. This evaluation step is performed before the first time step is executed. The set of determined parameters is used by a search mechanism, which works on parameter configurations and traverses the decision tree built in the offline tuning step. Thus, the search mechanism selects a final set of suitable implementation candidates based on the given parameter configuration. A generic iteration controller mechanism applies the final set of implementation candidates to the first time steps and compares their overall performance. This comparison process determines the best implementation candidate and the appropriate runtime parameters for the execution of the remaining time steps. The monitoring mechanism utilizes the initial comparison of the implementation candidates within the first time steps and observes the overall performance behaviour of the following time steps until the application finishes its execution. Hence, significant deviations of the performance measured can be detected and a new selection of a more suitable implementation candidate can be initiated to react on varying input data properties.

3) Cost estimations: The estimation of resources, e.g. time, energy or memory space, required to solve a given problem can be used to provide upper and lower limits for the appropriate resource. Hence, existing implementations can be rated based on these limits and are more or less suitable to solve a specific problem with actual input data on different HPC-systems or with different execution units, e.g. CPU, GPU or multiple HPC-systems. The cost estimation of a given algorithm or an implementation can be provided by a cost function. The granularity of cost functions can range from estimating whole programs of arbitrary size, which may result in quite complex or merely rough estimations, to specific parts of an implementation, e.g. time-step loops, single loop kernels or basic operations as vector, load or store operations. Since cost functions for loop kernels can be formulated quite accurately, offline autotuning approaches can benefit greatly of such estimation functions to predict the resource consumption of repeating calculations.

Time-stepping simulation methods should ensure an efficient calculation of each time step for varying input data. Thus, several implementations are provided for executing an actual time step, i.e. one loop iteration, and in general perform best with different input data and parameter configurations. Therefore, a selection of implementations has to be applied to limit the number of existing implementations to match a given requirement, e.g. time or energy consumption. Additionally, the input data may provide further selection criteria for the most suitable implementation variants based on the given cost functions. This leads to a significant reduction of the search space used within the autotuning process and, hence, ensures a faster convergence of the offline tuning step to find the optimal implementation variant for the current execution state.

III. Tuning examples and experimental results

To investigate the potential for self-adaptivity, we consider particle simulation methods, sparse matrix computations, and solution methods for differential equations and analyze their performance behavior with respect to varies tuning parameters, including the degree of parallelism, the operational frequency used, and application-specific parameters such as the grid size for particle simulation methods.

A. Particle simulation methods

We consider particle simulations with long-range interactions caused, e.g., by electrostatic or gravitational forces. The behavior of the particles is simulated by a series of time steps. In each time step, for each particle the simulation computes the forces caused by all other particles and determines the resulting new positions and velocities of the particles. This results in $O(N^2)$ complexity per time step if all interactions are taken into account. Many approaches have been proposed to reduce the complexity, including Fourier-based methods and hierarchical methods. Most of these advanced methods use a splitting approach and distinguish between long-range interactions from far-away particles and short-range interactions from nearby particles. Short-range interactions are typically computed exactly and long-range interactions are typically approximated.

The distinction between short-range interactions and long-range interactions is often performed by using a 3D grid structure with spatial cells. The short-range interactions cover all particles residing in the same or neighboring cells. The size of the spatial cells influences the computational behavior of the simulation and may also have an influence on the resulting accuracy of the simulation. The different advanced particle simulation methods mainly differ in the computation of the long-range interactions. In the following, we consider two different approaches, a Fourier-based approach and a hierarchical tree-based approach based on multipole expansions.

The performance behavior of the particle simulation methods considered depends on different hardware-specific and application-specific parameters. The hardware-specific parameters include the hardware platform used and the number of processes or threads employed. The application-specific parameters mainly include the separation between the short-range and long-range interactions. The number of particles and the initial distribution of the particles may also have a strong influence on the resulting performance of the different methods. In the following, we concentrate on the influence of the number of particles, the number of processors used, and the separation between the short-range and long-range interactions and the resulting grid sizes. The experiments are performed on an Intel Haswell system with two Xeon E5-2683 v3 processors, each equipped with 14 cores and a L3 cache of size 35,840 KB. The performance experiments are performed with two particle systems with non-uniform distribution: a small particle system with $300 \cdot 8^2 = 19,200$ particles and a large particle system with $300 \cdot 8^5 \approx 9.8$ million particles.

Fourier-based methods

Fourier-based methods compute the long-range interactions in Fourier space. Often, fast Fourier-transforms (FFT) are employed. The resulting computational complexity per time step is $O(N \cdot \log N)$, if the particles are sufficiently uniformly distributed [15]. In the following, we use an FFT-based particle simulation method for our experiments. For this method, the separation between the short-range and long-range interactions is determined by the grid sizes used. Figure 1 depicts the resulting execution times for both the small and the large particle system for different grid sizes. The execution times for 4 MPI processes are shown in the left diagram and the execution times for 56 MPI processes are shown in the right diagram. The diagrams show that different grid sizes lead to significantly different execution times due to the differences in the near-field and far-field computations. The optimum grid size that leads to the smallest execution time depends on the number of particles and the number of processes used. For both 4 and 56 MPI processes, the optimum grid size is 32 for the small particle system and 320 for the large particle system. Measurements have shown that these grid sizes are the optimum sizes also for other numbers of processes. For a sequential execution, the optimum grid size remains at 32 for the small particle system and changes to 448 for the large particle system (not shown in a figure).

The development of the execution time for different numbers of MPI processes is shown in Fig. 2 for the small particle system. It can be seen that the relative order between the resulting performance for the different grid sizes does not change with the number of processes.

Hierarchical multipole method

The fast multipole method (FMM) computes the particle interactions based on multipole expansions [16]. In each time step, the method calculates the (gravitational or electrostatic) potential at each particle position, which allows the computation of the new positions and velocities of the particles. The potential is split into a near-field and a far-field potential. To do so, an octree structure is defined, which results from a spatial decomposition

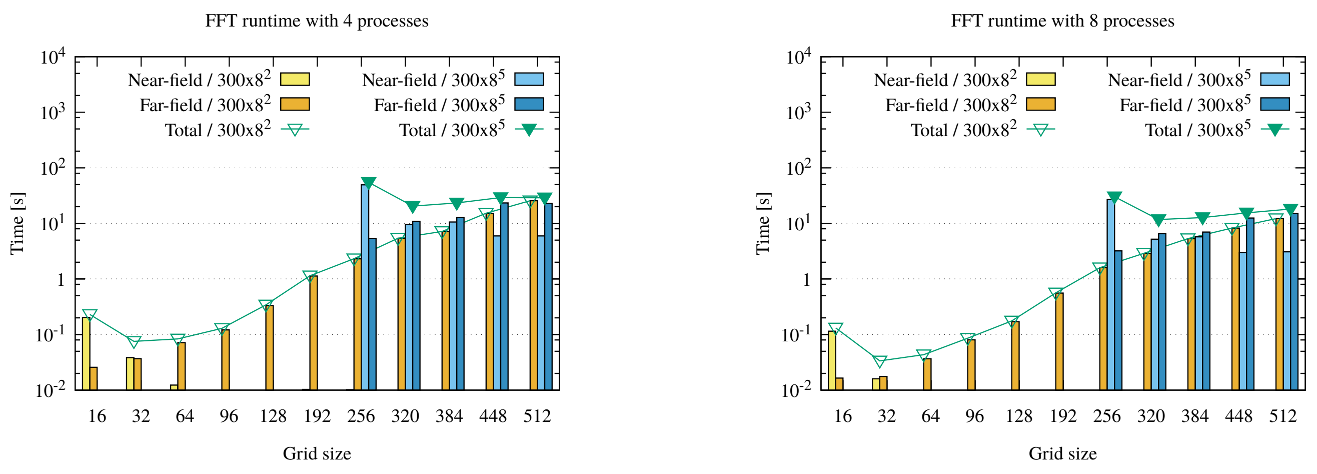

Figure 1. Execution times of the FFT-based particle simulation method for different grid sizes for the small and the large particle system for 4 (left) and 8 (right) MPI processes.

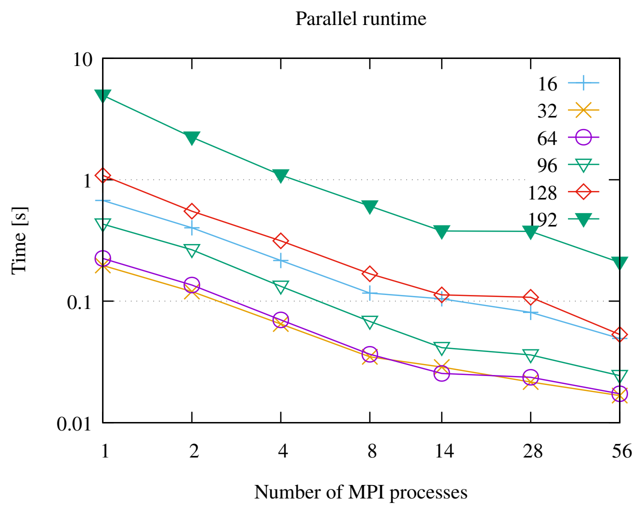

Figure 2. Development of the execution times of the FFT-based particle simulation method for different numbers of MPI processes and different grid sizes between 16 and 192.

into hierarchical boxes, and the particles are sorted into these boxes according to their current position. The spatial decomposition is controlled by a predefined maximum tree depth. For a particle p, the near-field potential is determined by computing the potential at the position of p caused by each of the particles in the same and in neighboring octree boxes. The far-field potential is computed by using approximations for the potential caused by all particles in a specific octree box. These approximations are computed for each octree level. The approximations at suitable octree levels are then used for approximating the far-field potential at a specific particle position. The maximum tree depth determines the separation in the near-field and the far-field potential. The resulting complexity is $O(N)$. In contrast to the FFT-based particle simulation, the FMM does not require that the particles are sufficiently uniformly distributed.

Figure 3 depicts the resulting execution times for both the small and the large particle system for different maximum tree levels. The execution times for 4 MPI processes are shown in the left diagram and the execution times for 56 MPI processes are shown in the right diagram. The figure shows that the maximum tree depth can have a significant influence on the resulting execution time. For the small particle system, the optimum maximum tree depth is 4 for 4 MPI processes, 8 for 8 MPI processes, and 3 for 56 MPI processes. For the large particle system, the optimum maximum tree depth is 7 for 4 MPI processes, 9 for 8 MPI processes, and 10 for 56 MPI processes.

Autotuning-Potential

The experiments of both long-range interaction approaches confirm that the runtime performance is influenced by the particle system size and the separation of the short-range and long-range interactions, i.e., by the grid size or the maximum tree depth. For the hierarchical method, the optimal separation setting to get the best runtime also depends on the number of MPI processes used. Some estimates, e.g., choose high number of MPI processes for better performance, can be done with offline autotuning before the first time step to start with acceptable parameters. To get the optimal parameters they have to be determined with online autotuning. Since the distribution of the particles in the particle system changes after each time step, it is also possible that the optimal parameter setting is changing over time. Thus the online autotuning has to be reapplied after several time steps to respond to imbalances and improve the overall runtime performance. Therefore the performance must be constantly checked for irregularities. For other optimization goals, e.g. energy efficiency, the autotuning-potential behaves the same.

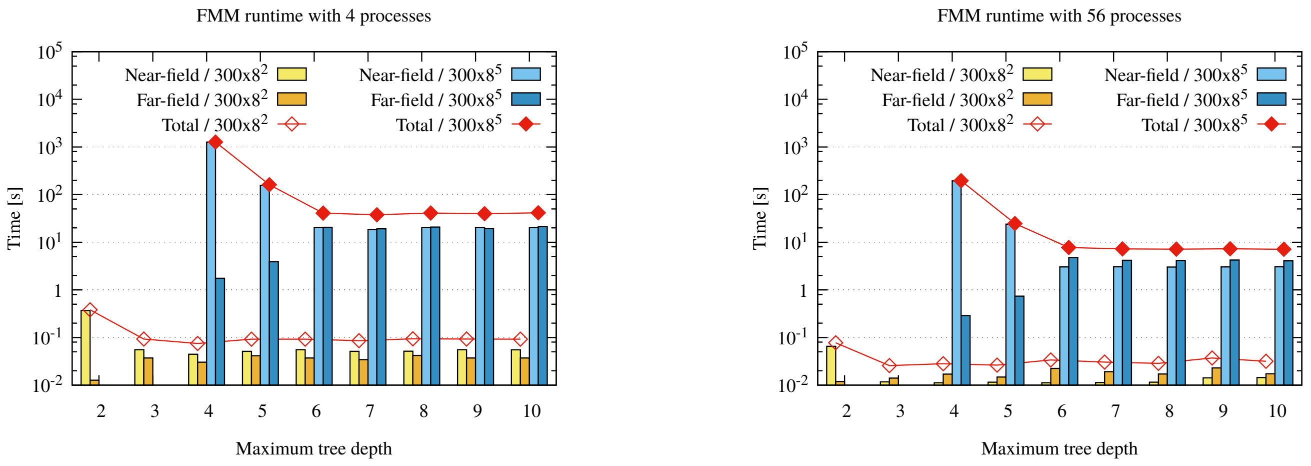

Figure 3. Execution times of the FMM particle simulation method for different maximum tree levels for the small and the large particle system using 4 (left) and 56 (right) MPI processes.

B. Sparse matrix methods

Methods performing operations on sparse matrices face additional challenges compared to methods processing dense matrices. Therefore typical calculations as sparse matrix-vector product (SpMV), sparse general matrix-matrix multiplication (GEMM) or solving sets of linear equations using Cholesky factorization have to consider carefully, which memory layout of the sparse input matrices provides the highest benefit in terms of the required performance and given limitations. The sparse property refers to matrices with a significant number of zero matrix elements and, thus, few actual non-zero matrix elements, which in general constitute a rather small fraction of the overall matrix elements.

In the case of the sparse GEMM operation, there is a upper runtime complexity of $O(n^3)$ for a dense matrix memory layout and a simple GEMM algorithm for dense matrices. However, the usage of more suitable sparse matrix memory layouts, called sparse matrix formats (SMF) in the following, is in general based on the exploitation of specific matrix properties of the input matrices and can lead to significant performance improvements. Since the reduction of the number of zero matrix elements stored leads to a better spatial locality when accessing the non-zero matrix elements, memory hierarchies, i.e. the cache hierarchy, can be utilized more efficiently. Hence, performance improvements in terms of time, power and memory consumption can be achieved.

Although the chosen SMF can have a great influence on the performance of the sparse GEMM, the set of parameters influencing performance rather consists of a diverse mix of software and hardware parameters. Solving sparse matrix methods efficiently on a given HPC system relies on several parameters, including the usage of different numbers of threads, the scaling of core and uncore frequency, the availability of SIMD processing units and a proper workload balance within the chosen implementation. Some of the listed parameter influences are considered subsequently with a comparison of three different SMF for a sparse GEMM operation. The SMF chosen are Compressed Sparse Row (CRS), Block Sparse Row (BSR) and Ellpack-Itpack (EIP). The thermal1 matrix with 574,458 non-zero matrix elements is used for the measurements and is taken from the SuiteSparse Matrix Collection [17]. Test systems are a Skylake system with 2× Xeon Gold 6130 processor each with 16 physical cores at 2.1 GHz and a Knights Landing system with a Xeon Phi 7250 processor with 68 physical cores at 1.4 GHz.

The effect of different numbers of threads are depicted in Fig. 4 and show different optimal thread utilizations of the SMF. Moreover, the effect of the nominal core frequency of two different HPC systems can be observed and shows implicitly an inefficient usage of power for certain proposed SMF, e.g. the EIP format can not utilize more than 32 threads for the given test matrix on the Knights Landing system. As an in-depth variable implementation parameter the block size for the BSR format has a significant influence on the achieved runtime as shown in Fig. 5. Comparing different modifiable frequency ranges, the results indicate that the runtime performance of different block sizes can differ such that a specific block size can perform better within a specific frequency range, i.e. the BSR-2 Version can achieve a better runtime within a low frequency range compared to the BSR-16 variant.

Autotuning-Potential

Since a great proportion of the significant parameters for sparse matrix methods are based on hardware properties or on the properties of the input matrices, which are invariant for most sparse matrix methods, a offline autotuning approach implicates the most benefits. Therefore, the application of the proper optimizations can improve the performance required, e.g. runtime, energy consumption or memory bandwidth utilization. Accordingly, performance models as the Roofline model [14], which have to be created only once for a given hardware configuration, can be used to a great extent for decision making processes of choosing a proper implementation for the desired sparse matrix operation. Furthermore, other performance models, i.e. the Execution-Cache-Memory (ECM) model [18], [9], can provide additional information about memory bound algorithms, which applies to a great proportion of sparse matrix methods. Thus, predictions for a given algorithm can be made for different numbers of threads, so that several optimization goals can be pursued, e.g. optimizing the runtime or finding an optimal number of resources used to satisfy specific energy or memory constraints.

Additionally, an online tuning phase can be used to observe and react to imbalances, which may occur during the execution of the actual problem. This execution behaviour can result due to the initial parameter setup, which was determined in the offline phase. Therefore the necessity for further optimizations can be caused by multiple reasons, e.g. inappropriate estimation of cost functions for kernels executed or data transfer times while using multiple HPC-systems at once. As a result, the runtime or energy measurements may not match the expectations and lead to workload imbalances between different execution units, e.g. CPUs or GPUs. Hence, an online tuning step is desired to adjust the workloads or distribution of data to use all execution resources to full capacity. For a sparse matrix-vector multiplication (SpMV), such an online tuning approach was investigated by [19] and is based on a redistribution of workload between MPI processes if the runtime measurements for the processing of a given number of matrix rows are not similar to measurements of other MPI processes.

Sparse matrix methods benefit the most of an offline tuning phase. However, online tuning approaches can still be applied and are a fine tuning process of the parameters determined in the offline phase. Moreover, the application of online tuning may depend on the actual sparse matrix operation, which should be performed, or the execution units used, e.g. online GPU workload adjustments may get quite complex if the initial data distribution for multiple GPUs has to be modified. Nevertheless, the potential of a mixed tuning approach still contributes to the overall goal of utilizing the given hardware resources to full capacity while optimizing for a given performance goal, e.g. runtime or energy consumption.

C. *Solving differential equations*

Numerical solution methods for ordinary differential equations (ODEs) compute an approximate solution for a given ordinary differential equation of the form

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)) \text{ with } \mathbf{y}(x_0) = \mathbf{y}_0. \tag{1}$$

by performing a series of time steps one after another until the end of the predefined integration interval is reached

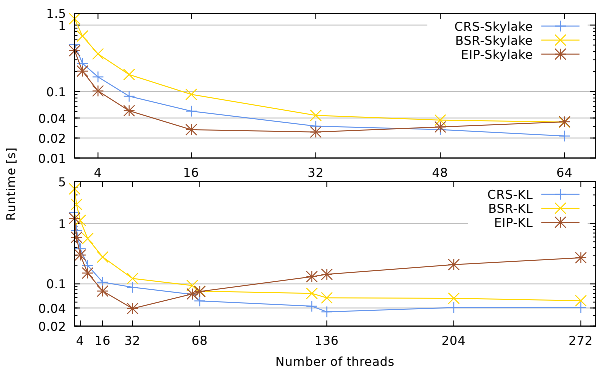

Figure 4. Runtime-Thread scaling of three sparse matrix formats for a Skylake system (top) and a Knights Landing system (bottom).

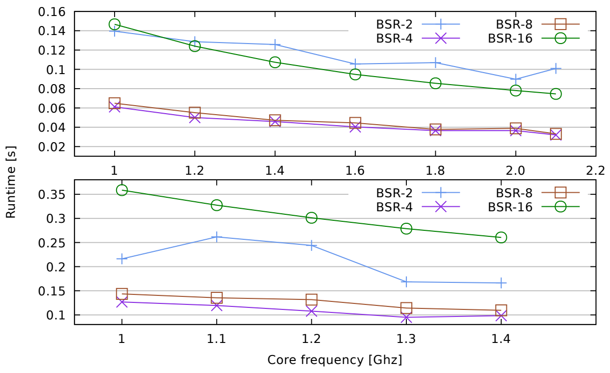

Figure 5. Runtime comparison of the Block Sparse Row matrix format with different block sizes as variable parameter for a Skylake system (top) and a Knights Landing system (bottom). Both diagrams refer to 64 threads used for the BSR variants.

[20]. As example method, we consider iterated Runge-Kutta (RK) methods which perform a fixed number m of computation steps in each time step using the approximation $\mathbf{y}_\kappa$ of the preceding time step. In each computation step, a fixed number s of stage vectors is computed using the stage vectors from the preceding time step and evaluating the function $\mathbf{f}$ defined by the differential equation to be solved. After the last computation step, the stage vectors are combined and an approximation $\mathbf{y}_{\kappa+1}$ for the next point in time is computed. An additional approximation of lower order can be computed additionally for error control and for the selection of the step size for the next time step. Overall, a four-dimensional loop structure results within each time step and many typical loop transformations such as loop interchange, loop unrolling, or loop tiling can be applied.

Taking the parameters of the transformations such as tile sizes and unrolling factors into account, a large number of code variants can be generated and it is not a priori clear, which of these variants will lead to the best performance on a given HPC system. This is an ideal situation for autotuning. A pure offline approach cannot be applied, since the function $\mathbf{f}$ of the differential equation to be solved may have a large influence on the resulting computational behavior of the different code variants. However, **f** is not known in advance and a numerical solution method should be efficient for different differential equations. On the other hand, a pure online autotuning is also not feasible, since too many code variants would need to be tested. Some of these code variants could be quite slow, leading a further increase of the resulting overhead. Thus, a combination of offline and online autotuning is most promising.

The diagrams in Fig. 6 show the performance of different shared memory implementation variants of the iterated RK method (IRK) for the BRUSS2D example and for different system sizes N. The time per step and component is plotted against the increasing number of threads. BRUSS2D is derived from a reaction diffusion PDE by a spatial discretization on a $N \times N$ grid using the method of lines. The resulting ODE system has dimension $n = 2N^2$. The experiments have been done on a system with two Intel Xeon E5-2697 v3 processors, each equipped with 14 cores and 35 MB L3 shared cache. As RK method we use the LobattoIIIC (8) [20] method with $s = 5$ stages and $m = 7$ computation steps. The code variants [21] utilize data parallelism, the n equations of the ODE system are distributed among the available number of threads p. The variants differ in the loop and the data structures used. Consequently, they have different memory access patterns, resulting in different utilization of the cache and the memory hierarchy. The variants denoted with suffix 'mt' use loop tiling as an optimization technique to further improve the locality of the memory references. Further, four variants (PipeDb1m, PipeDb1mt, ppDb1m and ppDb1mt) exploit a special structure of the function **f** of BRUSS2D by overlapping of vectors and by using pipeline-like computational structure of the computation steps m [21]. These variants only require lock-based local synchronization with neighbor threads, whereas all other variants need global barrier operations.

The diagrams in Fig. 6 indicate that the performance of IRK variants depends on the runtime parameters, such as the dimension of the ODE system and the number of threads executing the program. In particular, following observations can be made: (1) For the same system size, but for different numbers of threads, the order of the implementation variants varies. For example, for $N = 460$ and thread numbers $p < 20$, the variant EAmt offers the best performance, closely followed by the variant E and A. For $p = 20$ all variants require nearly the same execution time. For even larger numbers of threads, the variants EAmt, A and E are the slowest variants. The main reason for the smaller efficiency of these variants for large numbers of threads are the costs of the barrier operations used for synchronization of the threads. The variants A, E, EAmt require two barrier operations per stage in each computation step. All other variants need either only one barrier operation per computation step or use more efficient lock-based synchronization. (2) For the same number of threads, but for different system sizes, different variants obtain the best performance. For example, for $p = 16$ and $N = 460$, the variants EAmt, E and A perform best, whereas for $p = 16$ and $N = 2960$ other variants run faster.

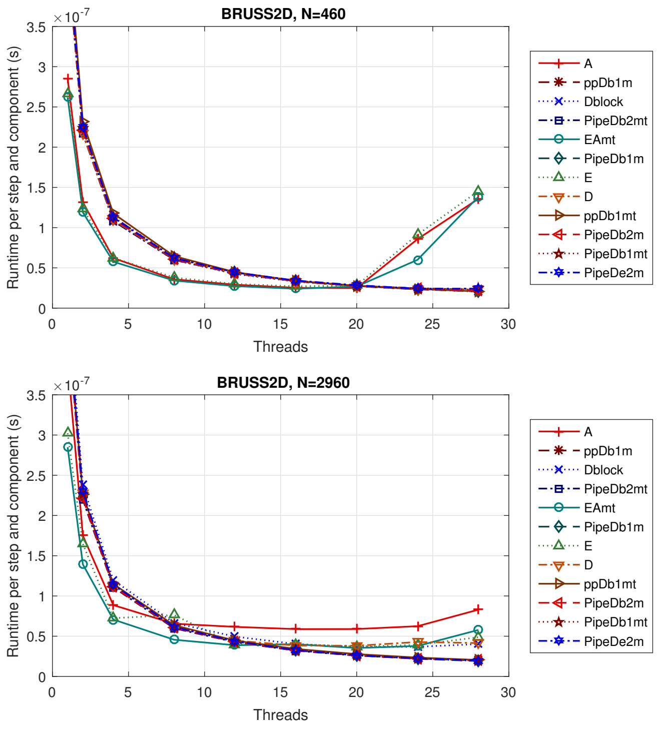

Figure 6. Execution time per step and component of iterated RK method for $N = 460$ (top) and $N = 2960$ (bottom) for different numbers of threads.

Autotuning-Potential

The experiments confirm that the performance of IRK variants is strongly influenced by the input data. Moreover, our experience shows that the performance of IRK variants also depends on the characteristics of the target architecture, such as the specific multi-core processor design, the cache architecture and the resulting memory latency and bandwidth. Thus, to obtain maximum performance, it is important that an IRK solver can adapt to the characteristics of the underlying architecture and of the ODE problem to be solved. Since usually these parameters are only known at runtime, the best variant cannot be determined at compile or installation time, and offline autotuning is not sufficient. For ODE solvers online autotuning has to be applied, but due to the large search space of candidate implementations and implementation parameters such as tile sizes for loop tiling or factors for loop unrolling, an online autotuning strategy should be supported by offline autotuning. For example, for multi-threaded implementations, offline measurements can be used to estimate the synchronization overhead of different implementation variants. At runtime this information can be used to avoid the evaluation of variants if their synchronization overhead is too high to outperform variants with lower synchronization.

IV. Related work

The first autotuning approaches were offline approaches for numerical methods for dense linear algebra problems for which the properties of the HPC systems used play the most important role for the execution time. These approaches include ATLAS [1] and PHiPAC [2]. For some application areas, properties of the input data may have a significant influence on the resulting execution time [22]. In these cases, online approaches need to be employed or integrated. An example for such an application area is signal processing, where the size of the problem to be solved determines which algorithm is the most efficient one. Examples for approaches in this direction are FFTW [10] and SPIRAL [11], [23]. For sparse linear algebra problems, the distribution on the non-zeros in the matrix to be processed may have a large influence, see OSKI [12] and SALSA [13]. Several autotuning approaches for specific application areas have been developed on the basis of domain-specific languages (DSL) for the description of the processing algorithms. From these DSL descriptions, a compiler can generate different code variants that can be tested in an online phase. An important application area for these approaches are stencil computations, see PATUS [24], Pochoir [25] and Halide [26] for approaches in this direction. All approaches mentioned above are application-specific, i.e., they have been developed for a specific application area and exploit specific properties of this application area.

Several general autotuning framework have been proposed that work independent from a specific application area. Active Harmony [3] is such a general autotuning framework that aims at iterative parallel applications that can come from many application areas. It includes a source-to-source compiler tool to generate new code variants at runtime according to loop transformations as specified by the user. Active Harmony uses a pure online approach, no offline component is included. Another general autotuning framework is Perpetuum [27], which aims at the selection of tuneable parameters such as block sizes and number of threads. Parcae [28] provides a user-level runtime system and a compiler to translate parallel constructs and patterns into a task-based execution model. PetaBricks [22] and Periscope [4] are other approaches in this direction. A detailed survey of compiler autotuning approaches with an emphasis on machine learning is given in [29].

All approaches mentioned above are either offline or online approaches, depending on the needs of the specific applications area. None of these approaches employs both an offline and an online phase as it is attempted in this paper.

The generation of different code variants is an important part of many autotuning approaches. The polytope model [30] and compiler-based approaches [26] are often used in this context. The resulting number of code variants can be large and efficient heuristic search strategies are important, including Simulated Annealing or Nelder-Mead [31]. OpenTuner [22] provides several search strategies, but other techniques based on machine learning are also investigated [32]. Energy and performance autotuning for two irregular applications, graph community detection using the Louvain method (Grappolo) and high-performance conjugate gradient (HPCCG) have been investigated in [33] for OpenMP multithreaded programs using OpenTuner.

V. Conclusion

In this article we have investigated the tuning potential of important simulation methods from scientific computing. The simulation methods considered include sparse matrix computations, particle simulation methods and solution methods for ordinary differential equations. In particular, a detailed performance analysis has been performed with considerations of relevant parameters. The tuning potential has been investigated for offline and online tuning.

The investigation shows that sparse matrix computations are mainly amenable to offline autotuning, particle simulation methods require the use of online autotuning, and solution methods for ordinary differential equations need a combination of both offline and online autotuning. This is related to the complexity of the input data and the the number of implementation variants available.

The offline autotuning analyzes the hardware and software and pre-selects parameters. With performance models, e.g., the roofline model or ECM model, some predictions can be done. The three applications make different usage of this offline approach. While good parameters can be chosen for sparse matrix calculations, the offline approach is used for the differential equations to reduce the search space used in the online approach.

The online autotuning is good for adjusting the parameters and reacting to imbalances. While the importance of this approach is different for the applications, each time-stepping method can be adjusted at runtime to achieve the best performance. Insufficient decisions from the offline approach can be adjusted. To detect these imbalances, a monitoring must be performed. This monitoring should be application independent and can be done with established tools.

Thus, it is usually advantageous to use both approaches to tune methods from scientific computing. The offline approach to set start parameters as good as possible and to select suitable code variants, and the online approach to fine-tune parameters, react on imbalances and select the appropriate code variant.

Acknowledgement

This work has been supported by the German Ministry of Science and Education (BMBF), Project title SeASiTe (Self-Adaptation of Time-step-based Simulation Techniques on Heterogeneous HPC Systems) with project number 01IH16012A/B.

References

[1] R. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001. doi: 10.1016/S0167-8191(00)00087-9

[2] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel, "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology," in *Proc. of the 11th Int. Conf. on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997. doi: 10.1145/263580.263662 pp. 340–347.

[3] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Proc. of the 2011 IEEE Int. Parallel & Distributed Processing Symp. (IPDPS 2011)*. IEEE, 2011. doi: 10.1109/IPDPS.2011.86 pp. 879–892.

[4] M. Gerndt, E. César, and S. Benkner, Eds., *Automatic Tuning of HPC Applications – The Periscope Tuning Framework*. Shaker Verlag, 2015, doi: 10.2370/9783844035179.

[5] M. Wolfe, "Compilers and More: What Makes Performance Portable?" April 19, 2016.

[6] P. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable Performance on Heterogeneous Architectures," in *Proc. of the Eighteenth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 2013. doi: 10.1145/2451116.2451162 pp. 431–444.

[7] J. Aseltine, A. Mancini, and C. Sarture, "A survey of adaptive control systems," *IRE Transactions on Automatic Control*, vol. 6, no. 1, pp. 102–108, Dec 1958. doi: 10.1109/TAC.1958.1105168

[8] D. BAA-98-12, "DARPA Broad Agency Announcement on Self Adaptive Software," 1997.

[9] 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. doi: 10.1002/cpe.3180

[10] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. doi: 10.1109/JPROC.2004.840301 Special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] F. de Mesmay, Y. Voronenko, and M. Püschel, "Offline library adaptation using automatically generated heuristics," in *Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2010. doi: 10.1109/IPDPS.2010.5470479

[12] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Institute of Physics Publishing*, vol. 16, 2005. doi: 10.1088/1742-6596/16/1/071

[13] V. Eijkhout and E. Fuentes, "Machine learning for multi-stage selection of numerical methods," in *New Advances in Machine Learning*. INTECH, feb 2010, pp. 117–136, doi: 10.5772/9376.

[14] 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. doi: 10.1145/1498765.1498785

[15] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. Bristol, PA, USA: Taylor & Francis, Inc., 1988, doi: 10.1137/1025102.

[16] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*. Boston: MIT Press, 1988.

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. doi: 10.1145/2049662.2049663

[18] J. Hofmann, J. Eitzinger, and D. Fey, "Execution-Cache-Memory Performance Model: Introduction and Validation," *ArXiv e-prints*, Sep. 2015.

[19] S. Lee and R. Eigenmann, "Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08. ACM, 2008. doi: 10.1145/1375527.1375558 pp. 195–204.

[20] E. Hairer, S. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*. Berlin: Springer–Verlag, 1993, doi: 10.1137/1032091.

[21] M. Korch and T. Rauber, "Locality optimized shared-memory implementations of iterated Runge-Kutta methods," in *Euro-Par 2007. Parallel Processing*, ser. Springer LNCS, vol. 4641. Springer, 2007. doi: 10.1007/978-3-540-74466-5_78 pp. 737–747.

[22] J. Ansel, "Autotuning programs with algorithmic choice," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, Feb. 2014. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2014/ansel-phd-thesis.pdf

[23] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005. doi: 10.1109/jproc.2004.840306

[24] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proc. of the 25th IEEE Int. Parallel and Distributed Processing Symp.*, May 2011. doi: 10.1109/IPDPS.2011.70

[25] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir stencil compiler," in *Proc. of the Twenty-third Annual ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '11)*. ACM, 2011. doi: 10.1145/1989493.1989508 pp. 117–128.

[26] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'13)*, 2013. doi: 10.1145/2499370.2462176 pp. 519–530.

[27] T. Karcher and V. Pankratius, "Run-time automatic performance tuning for multicore applications," in *Euro-Par 2011. Part I.*, ser. LNCS, E. Jeannot, R. Namyst, and J. Roman, Eds., no. 6852. Springer, 2011. doi: 10.1007/978-3-642-23400-2_2 pp. 3–14.

[28] A. Raman, A. Zaks, J. Lee, and D. August, "Parcae: A System for Flexible Parallel Execution," in *Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ser. PLDI '12, 2012. doi: 10.1145/2254064.2254082 pp. 133–144.

[29] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano, *Automatic Tuning of Compilers Using Machine Learning*, 1st ed. Springer, 2017. doi: 10.1007/978-3-319-71489-9.

[30] 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. doi: 10.13140/2.1.5066.3368 pp. 45–54.

[31] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965. doi: 10.1093/comjnl/7.4.308

[32] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *28th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS 2014)*, May 2014. doi: 10.1109/IPDPS.2014.59 pp. 501–512.

[33] A. Panyala, D. Chavarra-Miranda, J. B. Manzano, A. Tumeo, and M. Halappanavar, "Exploring performance and energy tradeoffs for irregular applications," *J. Parallel Distrib. Comput.*, vol. 104, no. C, pp. 234–251, Jun. 2017. doi: 10.1016/j.jpdc.2016.06.006