



A performance- and energy-oriented extended tuning process for time-step-based scientific applications

Natalia Kalinnik¹ · Robert Kiesel² · Thomas Rauber¹ · Marcel Richter² · Gudula Rünger²

Published online: 25 August 2020
© The Author(s) 2020

Abstract

Scientific application codes are often long-running time- and energy-consuming parallel codes, and the tuning of these methods towards the characteristics of a specific hardware is essential for a good performance. However, since scientific software is often developed over many years, the application software usually survives several hardware generations, which might make a re-tuning of the existing codes necessary. To simplify the tuning process, it would be beneficial to have software with inherent tuning possibilities. In this article, we explore the possibilities of tuning methods for time-step-based applications. Two different time-step-based application classes are considered, which are solution methods for ordinary differential equations and particle simulation methods. The investigation comprises a broad range of tuning possibilities, starting from the choice of algorithms, the parallel programming model, static implementation variants, input characteristics as well as hardware parameters for parallel execution. An experimental investigation shows the different characteristics of the application classes on different multicore systems. The results show that a combination of offline and online tuning leads to good tuning results. However, due to the different input characteristics of the two application classes, regular versus irregular, different tuning aspects are most essential.

Keywords Time-stepping methods · Ordinary differential equation · Particle simulation · Offline tuning · Online tuning · Multicore

1 Introduction

The design and implementation of efficient parallel simulation codes are still a very time-consuming and cumbersome task, which requires not only insight into the scientific problem to be simulated but also advanced programming capabilities and

✉ Thomas Rauber
rauber@uni-bayreuth.de

Extended author information available on the last page of the article

knowledge of the hardware behavior. Many decisions have to be taken, starting with the choice of the simulation method and including several further choices, such as the programming model, the data structures or the code structuring. For the goal to achieve a good performance as well as a low energy consumption, there are more decisions at runtime, such as the choice of the operational frequency or the number of cores for a multicore platform. The multitude of choices for implementation and execution influences the performance and energy consumption for the execution of the final program, such that it is almost impossible to pick the most efficient alternative without extensive planning. When considering time-step-based simulation applications, another influencing factor can be crucial, which is the dependence of the computation behavior on the input data and the varying data during computation. For this class of applications, we explore the design and implementation possibilities and propose an extended tuning process which includes early choices, such as the algorithmic and programming model choice, as well as the influence of the input data and varying data during runtime.

Time-step-based simulation applications are used to simulate scientific problems with an inherent time evolution. The coarse structure of such applications distinguishes between a loop over discrete time steps and the space-based data which are manipulated in each time step providing modified data for the next time step. The solution of such a problem might be the series of space data resulting from the time step computations or only the final result after a series of time steps. Several classes of algorithms exist for solving such problems with time evolution. In this article, we consider solvers for systems of ordinary differential equations and particle simulation methods for long-range interactions in N-body problems.

Ordinary differential equations (ODEs) are differential equations describing a function in dependence on its derivative in one independent variable, which is referred to as time variable. The numerical integration determines the function value at consecutive points in time by calculating the function value at this time point based on previously computed function values. For a concrete solver implementation, the choices are the specific algorithm, such as an Euler method or a Runge–Kutta method, or the programming model, such as MPI, Pthreads or OpenCL. The choice has to be suitable for the architecture and the algorithm. Most important is the influence of the start value and the computation structure of the right-hand side of the ODE, which depends on the problem modeled; sparse problems and dense problems can be distinguished.

Particle simulation methods are used to simulate the interaction of a set of given particles, where the kind of interaction is given by the underlying problem and might be the calculation of forces between particles. The particle problem might comprise interactions only between particles being close in space but also long-range interactions where interactions between all particles have to be computed and contribute to the result. The simulation is done in time steps such that each time step takes the particles and their properties, such as velocity and position, and recomputes the properties in the time kernel such that this new information is used in the subsequent simulation. The important property of a particle simulation is that the positions of the particles are not fixed but the particle distribution may vary with the simulation which in turn has influence on the computation structures and the performance.

Thus, qualities of the particle sets, such a homogeneous distribution in the space can change to an inhomogeneous distribution and vice versa, which can be a challenge for the tuning.

The choice for the different implementations and execution details can take place at different stages of programming and execution. In general, such a tuning process distinguishes between offline and online tuning, where offline tuning comprises all choices which are taken without knowledge about the actual input data and online tuning can include such knowledge at runtime. In this article, we propose an extended version of the tuning stages especially suited for the holistic tuning of time-step-based simulation algorithms. The offline tuning is extended by including the algorithm choice step in which an algorithm is chosen from a set of algorithms solving the same problem. Also, a design decision with incomplete knowledge about the input data is included by choosing a program structure suitable for the characteristics of the data, i.e., the access structure, without knowing the exact content. An important observation is that the program code in each time step is performing the same computation, however with varying data. Thus, the offline tuning can concentrate on this inner kernel. The online tuning can exploit the repeated computation of the kernel computation of a time step. We propose an online tuning consisting of several tuning stages with an early stage covering the early time steps and a later stage covering the following time steps, separately. The advantage is that data-dependent information gathered in the early time steps can be exploited in later time steps. Also, an adaptation of the choices is possible in a continuous way for the entire simulation time with updates after a fixed number of steps.

To make this approach work, the provision of different program variants for a simulation method is necessary as well as an assessment of this program variants by different metrics, e.g., the execution time or the energy consumption. We propose the key concepts and present the workflow of the extended tuning process together with the essential components. For parallel program variants, the exploitation of parallelism and frequency scaling is included in the tuning process. In summary, the contributions of this article include:

- A performance and energy analysis of particle simulation methods and solution methods for ODEs on different multicore architectures and for several input data sets;
- The identifications of parameters with a large influence on the performance and energy consumption and the analysis of their amenability for offline and online tuning for these time-step-based simulation applications;
- An investigation of the potential for interaction between offline and online tuning steps and of the use of information gathered in the offline phase to support the choices in the online phase;
- A proposal of an extended tuning process approach to balancing the offline and online tuning stages, which exploits the time-step-based nature of the simulation applications.

The article is structured as follows. Related work is discussed in Sect. 2. Section 3 introduces essential definitions and proposes an extended tuning process for

time-step-based simulation algorithms. Section 4 studies solver for ODEs for different input problems. Section 5 investigates the potential for tuning of particle simulations by considering several algorithm, programming models, data parameters and frequencies. Section 6 concludes.

2 Related work

Early tuning approaches such as ATLAS [37] and PHiPAC [5] aim at problems from dense linear algebra. They generate a lot of implementation variants of the specific kernels, execute them and determine the (near) optimal configuration *offline*, i.e., during compilation and installation of the library on the target platform. For some application areas, the performance strongly depends on the properties of the input data. In these cases the best variant has to be selected at application runtime *online* when the structure of the input data is known. An example for such an application area is signal processing, with auto-tuning libraries FFTW [9] and SPIRAL [28]. All these approaches have been developed for the optimization of a specific numerical application and can take the specific characteristics of the application into consideration. In this article, we propose an approach that has been designed for general time-step-based applications and therefore captures a wider class of algorithms from different areas.

Tuning techniques have been also used to optimize the performance of stencil computations on CPUs and GPUs, see, for example, PATUS [6], Pochoir [34], Halide [29]. In these approaches, a stencil is specified through a domain-specific language (DSL) and the DSL description is used to generate different implementation variants that are evaluated at runtime. YASK [38] also provides a DSL to describe stencil computations and uses genetic-algorithm-based auto-tuning to generate optimized codes for Intel Xeon and Intel Xeon Phi architectures. Stencil computations capture a large class of applications from different areas, including time-stepping applications. However, the approaches mentioned do not exploit the time-stepping nature of the applications, if it exists, as it is done in this article. Instead, they are focused on the optimization of the individual stencils.

There are application-independent frameworks to simplify self-adaptivity such as active harmony [35], Parcae [30], Periscope [11] and OpenTuner [1]. Active harmony generates new code versions at runtime using a source-to-source compiler and user-specified loop transformations and is a pure online approach. Parcae generates a task-based execution model based on an user-level runtime system and a compiler to translate specified parallel constructs. Similar frameworks in this direction include Perpetuum [21] and PetaBricks [1]. In contrast to our approach, these frameworks are application independent and they are also not targeted towards time-stepping algorithms.

The generation of different code variants is an important part of many tuning approaches. The polytope model [8] and compiler-based approaches [29] are often used in this context. The search space of code variants explored can be large, and therefore, efficient heuristic search strategies are important. OpenTuner [1] provides several search strategies and uses a multiarm bandit technique

to choose automatically the best strategy for the target application. The choice of the MPI library used (such as OpenMPI [10] or MPICH), the MPI parameter settings and the underlying MPI algorithms can affect the resulting performance and energy consumption of parallel variants. For example, the optimal MPI algorithm for a collective operation often depends on the message size, the target hardware architecture and number of processes used. In [33] Periscope is extended to automatically optimize the values of MPI configuration parameters. The approach supports performance analysis and tuning of MPI parameters at runtime.

All tuning approaches mentioned above are search-based. An alternative to search-based approaches is the use of machine learning techniques such as neural networks and support vector machines (SVMs). Nitro [25] uses SVMs in an offline phase to build a model through training and to consult it at runtime for the selection of the best configuration. An overview of compiler-based frameworks considering machine learning approaches is presented in [3]. The potential of using a reinforcement learning for tuning MPI communication libraries is explored in [7]. Such machine learning techniques are currently not included in our approach, but they could in principle help to select candidate configurations for both particle simulation methods and ODE solvers.

The state of the art in tuning and auto-tuning in high-performance applications is described in [4]. A framework to exploit domain knowledge in order to reduce the search space and to optimize the distribution of work and various platform-specific parameters (such as tile sizes, thread blocks, shared memory, unrolling) is presented in [27]. The auto-tuning approach is embedded in heterogeneous compiler Polly-ACC [12]. Auto-tuning approaches for minimizing the energy consumption have been investigated in [26, 36].

Tuning techniques are also important in many other application areas. An important example is weather forecasting systems that are often very time-consuming and can therefore benefit much from a good tuning. Many weather forecasting systems are based on the weather research and forecasting (WRF) model. This model is considered in [24], and it is investigated how to obtain the best WRF configuration, including compilers, MPI libraries and hybrid shared memory paradigms. In contrast to our work, one specific application is considered and no different code variants are taken into consideration.

The two application fields, particle simulation methods and solving differential equations, are well-known fields, which are also examined individually. In the approach of [18], an online tuning is presented to optimize the performance of parallel solution methods of solving differential equations. This approach does not consider the energy consumption and does not optimize the number of threads. Runtime and energy consumption of particle simulations are examined in [16, 22]. The tuning potential with respect to performance for particle simulation methods and solving differential equations is examined in [20]. A performance and energy analysis of these applications is presented in [19]. In this article, we extend this work by considering the tuning potential for both execution time and energy consumption and present an approach for combining offline and online tuning.

3 Extended tuning process for time-stepping methods

In this section, we outline our approach for a systematic exploration of different tuning possibilities that are available for time-step-based scientific applications.

3.1 Tuning possibilities and configurations

The execution of a scientific application depends on several parameters that have to be selected at different levels. This includes the selection of a specific simulation algorithm, the selection of a programming model, the generation of a specific program variant, the usage of specific compiler options and the setting of the parameters for the hardware environment. The selection of these parameters influences the resulting execution time and energy consumption. Depending on the specific application, different input data may require different parameter settings for the best performance. An illustration for the different choices is given in Fig. 1.

Depending on the application area, different algorithms may be available for performing a simulation. An example is particle simulation methods where different grid-based and tree-based simulation algorithms are available, see Sect. 5, and the initial distribution of the particles determines which simulation algorithm is suited best. Different programming models can be used for the implementation of

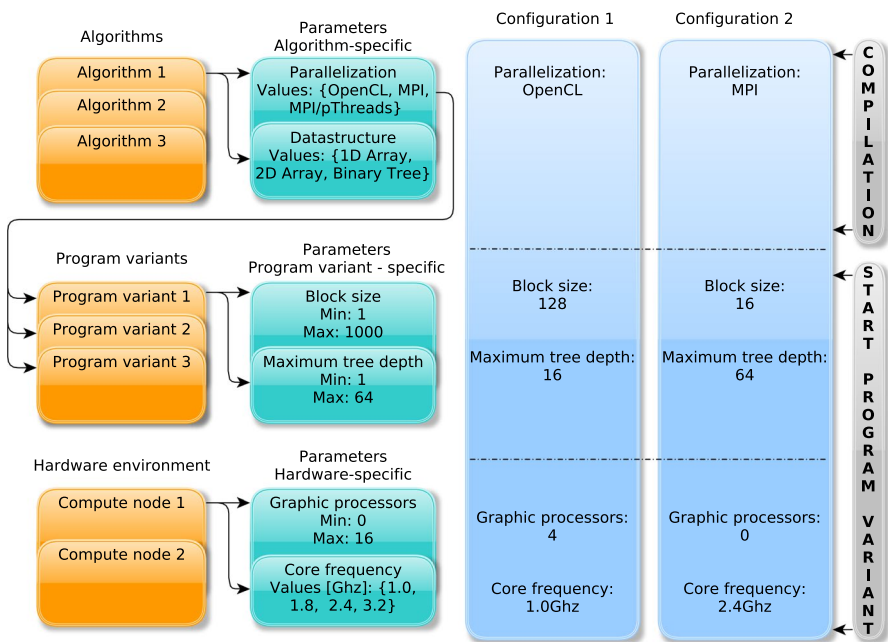


Fig. 1 Illustration of the design space for the configurations of a scientific simulation. The dashed lines indicate the design choices for: the algorithm, programming model, program variant, parameter space and the hardware parameter

the selected simulation algorithm, such as MPI, Cuda or OpenCL. For a specific simulation algorithm, different program variants may be available that represent different implementations obtained, e.g., by the use of program transformations. Such program transformations can be parameterized by parameters, such as block sizes for loop tiling or unrolling factors for loop unrolling. Since many transformations can be applied for different loops in different combinations, a large number of possible program variants may result. Such program variants can also be generated automatically by a compiler tool. For the generation of the executable program, different compiler options can be used, controlling, e.g., vectorization or optimizing transformations. A large number of combinations for compiler options are possible [2]. For the final execution of the application code, different hardware parameters can be selected, including the number of executing threads or processes as well as the operational frequency via DVFS (dynamic voltage frequency scaling).

In the following, especially in the experiments with the time-step-based methods, we use the following definitions:

Definition 1 A *program variant* is an implementation of a simulation algorithm available as source code. Different program variants of the same algorithm are identical in their numerical behavior but may differ in their memory access pattern or their order of computations. Program variants can be generated by using program transformations such as loop interchange or loop unrolling.

Definition 2 A *configuration* of a scientific application code is an executable program implementation resulting from specific choices for the algorithm, subcomponents, programming model, program variant, compiler options, hardware parameters and program parameters.

3.2 Tuning opportunities for time-step-based applications

Time-step-based simulation applications have several properties from which a tuning process can benefit. These properties include:

- the similarity of the computational structure of the computation kernel of each time step,
- the dependence of the computations performed in each time step on the input data,
- the similarity of the characteristics and the data access structure to the input data in each time step,
- the potential of parallelism across the space in each time step.

Computation structure of time kernel For time-step-based applications, a dynamic behavior of the computations in each time step and a dependence on the initial input data is possible. Thus, even when starting with a configuration that is optimal for the first time steps, the simulation data may evolve in such a way that another configuration would be more appropriate after some time steps. Such a

dynamic behavior cannot be forecast in advance and it cannot be determined a priori which configuration should be used for which time steps, especially if the computation behavior is highly input dependent.

Dependence of the computation structure on data To capture the existence of a dynamic behavior, we distinguish between *regular* and *irregular* time-stepping scientific applications: a regular time-step-based application does not change its computational behavior between different time steps, whereas an irregular time-step-based application is characterized by the dynamic computation behavior described above, which may lead to a change in the performance and energy behavior during the course of the time steps. For regular applications, the performance and energy behavior can be independent of the input data or it may depend on specific properties of the input data. For example, ODE solvers may lead to quite different performances depending on the right-hand side of the ODE system to be solved. Since the property of being regular or irregular is a property of the simulation algorithm, this is the same for each time step and does not change in a simulation run. For ODE solvers, the access pattern to the input data and the data structures storing the simulation data during the simulation can be sparse or dense or mixed, but it is fixed during runtime. For particle systems, the input data can be homogeneous or inhomogeneous or mixed and the characteristics can change during the simulation due to the evolvement of the particle system.

Tuning for parallelism The series of time steps in time-step-based algorithms is inherently sequential, since the result, such as a new data value or a new distribution of particles, is used as input for the next time step. However, there is usually a potential of parallelism within the calculation of each time step. For ODE solvers, the potential of parallelism is called parallelism across the space when systems of differential equations are solved and the function value computed in each time step is a potentially large vector. For particle simulation methods, the parallelism stems from the calculation of the interactions between the particles which can be distributed among processing units. Due to the varying particle distribution, the sorting methods resorting the particles for locality can help to reduce imbalances and to speed up the calculation by spatial locality. The tuning for parallelism includes:

- the number of cores and processes or threads used,
- frequencies scaling for energy reduction,
- simultaneous investigation of parallelism and application parameters, e.g., number of cores and gridsize,
- load balancing in particle simulation methods by sorting algorithms,
- load balancing in simulation algorithms for an execution on heterogeneous CPU+GPU platforms.

Thus, in general, the tuning method for parallelism within a time step are reflected by standard techniques. However, the interaction between time steps in the tuning stages of the online tuning can be beneficial for performance improvement.

3.3 Extended tuning process in different phases of program execution

Due to the multitude of different design decisions of a configuration, a systematic way of investigating and applying tuning steps in a specific time-step-based algorithm is advantageous. In this subsection, we propose such a systematic tuning process. For each choice of algorithm, hardware platform and programming model, a separate investigation of the tuning process is done. The tuning process is performed at different stages of program generation and execution. For time-step-based applications, it can be distinguished between offline tuning and online tuning:

- *Offline tuning* Offline tuning is performed at the generation time of the executable program and before the actual execution of the simulation. The hardware environment and the pool of available program variants is known. Offline tuning includes the selection of a pool of variants that is suitable for the hardware environment, the selection of transformation parameters for the different variants and the selection of compiler options. However, the input data are not available and the optimization goal may still be unspecified. The result of offline tuning is a pool of executable program variants that can be used for the time steps of the simulation.
- *Online tuning* Online tuning is performed during the execution of the simulation. All tuning information is available as well as the input data, the optimization goal and user-defined restrictions. Online tuning can be performed before the first time step, during the first time steps and also between the different time steps: before the first time step, suitable program variants can be selected that fit to the characteristics of the input data. During the first time steps, performance and energy measurements can be used to select the program variant best suited. Between the time steps, a dynamic behavior of the simulation progress can be captured and can be used for selecting variants for the following time steps.

The offline tuning phase is executed before the simulation starts and, thus, has no information about the actual input data. In this phase, performance and energy models can be used to predict the runtime and energy consumption of different algorithms and program variants [31]. These models can then be used to provide a coarse ranking of program variants and to select a candidate set of variants for the online phase. Regarding the overhead of the offline phase, the evaluation, selection and ranking of a potentially large number of program variants may be time-consuming, but for a given hardware environment, programming model and a fixed set of program variants it has to be performed only once for any input data that is processed within the online phase. In order to organize information within the tuning process, a decision tree and a history database can be used, as shown in Fig. 2. The decision tree can be considered as a link between the offline phase and the online phase which transfers relevant information from the offline phase to the online phase.

The online phase is based on the provision of user-defined information, such as the optimization goal. Moreover, the actual input data are now available. There are four major steps to be performed within the online phase: adaptation, tuning, simulation loop and monitoring; see Fig. 3 for an illustration. All these steps can be done

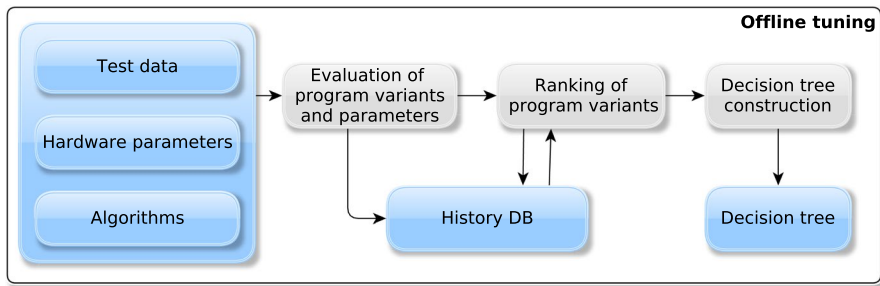


Fig. 2 Illustration of the offline tuning phase which is executed before the input data is available. Elements with a blue frame refer to input and result data, which may be used multiple times within the tuning process

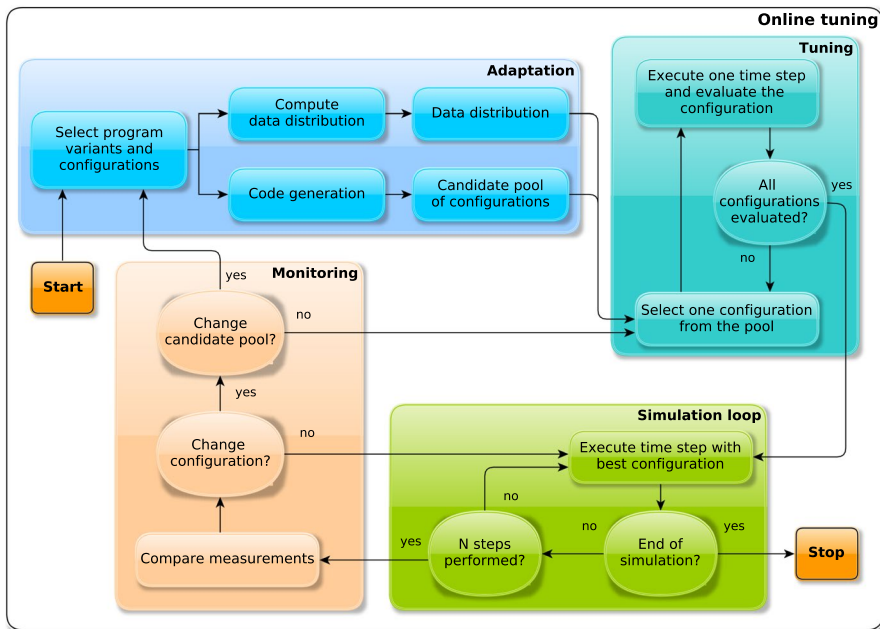


Fig. 3 Illustration of the online tuning phase which is to be performed when all tuning information is available. Elements with a blue frame refer to input and result data, which may be used multiple times within the tuning process

by a tool or can be incorporated in the development of the time-step-based simulation. The online phase starts with the adaptation component which selects a suitable subset of configurations using the information about the actual input data. The resulting pool of configurations, referred to as candidate pool, is used in the subsequent tuning steps. The tuning step evaluates configurations from the previously created candidate pool by executing single time steps and recording the measured performance and energy behavior with regard to the optimization goal. The performed

time steps contribute to the solution of the problem, which means that no dummy data but the real input data are processed. For the simulation, the best configuration is chosen from the previously evaluated configurations to process a number of time steps. The task of the monitoring is to verify the performance behavior of the currently running configuration and to initiate a repetition of the tuning cycles if required.

3.4 Workflow of the overall tuning process

Figure 4 illustrates the workflow for the extended tuning process combining offline and online tuning opportunities. The online part of the tuning process is shown with its internal structure using the following main components:

- *Adaptation* The adaptation step processes information provided by the offline phase to create the candidate pool of configurations for the subsequent components. Based on the pool of program variants determined by the offline phase, the adaptation step refines the program variants and selects those variants that fit to the characteristics of the input data and the optimization goal. Accordingly, the adaptation step includes the generation of program variants obtained by setting specific parameter values for program transformations, the management of the distribution of data and the selection of program variants based on the user input.
- *Tuning* The tuning step evaluates configurations of the candidate pool to find the best program variant and its parameter values. This is done by performance and energy measurements during the first time steps. The size of the candidate pool obtained by the adaptation step determines the number of time steps needed for the tuning step. In accordance with the specified optimization goal, the best configuration is passed to the simulation loop.

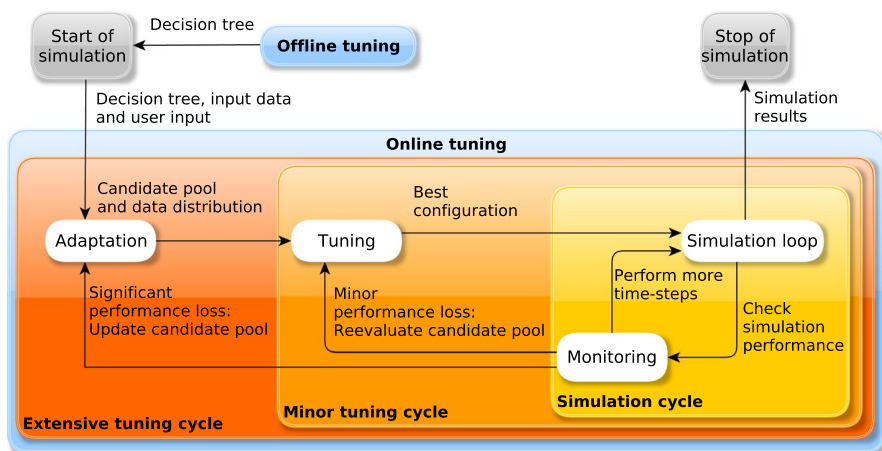


Fig. 4 Illustration of the general tuning structure including offline and online tuning

- *Simulation loop* The simulation loop executes the actual time-step-based simulation algorithm. It performs a number of time steps with the current configuration determined by the tuning step and ensures that the performance stays similar between the time steps. The tuning overhead is reduced to a minimum and progress towards the solution is focused.
- *Monitoring* The monitoring step observes the progress of the simulation loop and the performance of the time steps. If a larger performance degradation is observed, the tuning step can be re-entered to revise the decision on the configuration used for the simulation loop. This decision is based on performance or energy measurements.

The workflow of the extended tuning process can be understood as a sequentially processed list of main steps which comprise relevant tuning decisions. Regarding the depicted workflow, the tuning process will always start with the adaptation step, the tuning step and the simulation loop in this order. Accordingly, a set of configurations is created in the adaptation step, evaluated in the tuning step and the best configuration is then used to perform many time steps in the simulation loop.

The characteristics of the simulation data may change during the course of the simulation, especially for irregular simulation applications. Thus, the need for changing the running configuration may arise during the simulation. Therefore, the proposed monitoring step is a central component to incorporate tuning steps to find a better configuration for the current state of the simulation data. The required tuning steps may lead to different tuning overheads, e.g., depending on the comparison between the measured and the expected performance. In order to differentiate the degree of overhead caused by the monitoring step to find a new configuration, the resulting overhead can be categorized as minimal, minor or extensive. Since any set of applied tuning steps leads to a new configuration for the next instance of the simulation loop, this cyclic behavior and the degree of expected overhead are encapsulated in the following three tuning cycles; see Fig. 4 for an illustration:

- *Extensive tuning cycle* The extensive tuning cycle is required, if the set of program variants and the related configurations have become insufficient. Thus, the adaptation step has to be re-entered and the set of program variants must be improved by adding or removing configurations to match the current simulation data. A major tuning overhead may result.
- *Minor tuning cycle* The minor tuning cycle is required, if the currently used configuration needs to be changed. Therefore, program variants or parameter values may have to be adapted. The tuning steps are executed to find a better configuration from the available ones. A minor tuning overhead results.
- *Simulation cycle* Only the necessary performance measurements are executed in addition to the computations of the simulation. These measurements are required to monitor the performance of the running configuration. A minimum tuning overhead results.

The online tuning phase ends after the last simulation time-step. After the storage of the results, which is handled by the application, the tuning process accesses the

post tuning phase. In this phase all information relevant to the tuning process are gathered, sorted by significance and compressed to store only relevant data for later evaluation of the decisions made during the tuning process and the achieved performance by the selected program variants and configurations. These data can be used to enhance future tuning processes.

3.5 Case studies and experimental evaluation

To evaluate the potential for adaptivity of time-step-based methods, experimental studies with the two application fields, which are solution methods for ODEs and particle simulation methods, are conducted. The tuning goals are both execution time and energy consumption. The energy consumption for particle simulation methods is gathered with the Performance Application Programming Interface (PAPI) 5.6 and the Running Average Power Limit (RAPL) interface to read the appropriate model-specific registers (MSRs). For the measurements of the ODE solvers, the LIKWID toolset has been used. All application codes are compiled using the GNU Compiler Collection (GCC). Each measurement was repeated multiple times to avoid artifacts in the results, while the exact number of runs differs between the application fields. The average of the measurements is shown. Since the performance and energy behavior is influenced by different factors for different applications, we provide an analysis of a variety of tuning parameters, including the operational frequency, the degree of parallelism and application-specific parameters.

4 Solution methods for ordinary differential equations

4.1 Numerical method

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

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

by performing a series of time steps $\kappa = 0, 1, \dots$ until the end of the predefined integration interval $[t_0, t_e]$ is reached [14]. Starting at time t_0 with the initial approximation $\mathbf{y}_0 = \mathbf{y}_0$, at each time step κ a new approximation $\mathbf{y}_{\kappa+1}$ is computed using the approximation \mathbf{y}_κ and, depending on the specific method, possibly further previously computed approximations. Sophisticated methods estimate the local error ϵ committed at each time step. Based on this error estimate, the step size for the next time step h_{new} can be adapted such that a larger step size can be chosen where a small step size is not needed to obtain the required accuracy, and, hence, the overall number of time steps and the computation time is reduced. If ϵ is below a user-defined tolerance TOL , the approximation is accepted and the algorithm proceeds with the next time step, generally using a larger step size. If, however, the new approximation does not satisfy the required accuracy, the step size control algorithm rejects the current time step and repeats it with a smaller step size.

As example method, iterated Runge–Kutta (IRK) methods which perform a fixed number $p = m - 1$ of corrector steps in each time step κ using the approximation \mathbf{y}_κ of the preceding time step are considered. In each corrector step k , a fixed number s of stage vectors $\mathbf{Y}_l^{(k)}$ is computed using the stage vectors $\mathbf{Y}_i^{(k-1)}$ from the preceding time step $k - 1$ and evaluating the function \mathbf{f} defined by the ODE to be solved:

$$\begin{aligned} k &= 1, \dots, m, \quad l = 1, \dots, s : \\ \mathbf{Y}_l^{(k)} &= \mathbf{y}_\kappa + h_\kappa \sum_{i=1}^s a_{li} \mathbf{F}_i^{(k-1)}, \\ \text{with } \mathbf{F}_i^{(k-1)} &= \mathbf{f}(t_\kappa + c_i h_\kappa, \mathbf{Y}_i^{(k-1)}), \end{aligned} \quad (2)$$

After the last corrector step, the stage vectors are combined and an approximation $\mathbf{y}_{\kappa+1}$ for the next time step $\kappa + 1$ is computed. An additional approximation of lower order can be computed additionally for error control and for the selection of the step size h_{new} for the next time step.

The coefficient matrix $A = (a_{li}) \in \mathbb{R}^{s,s}$, the vector $\mathbf{c} = (c_i) \in \mathbb{R}^s$, the order p , and the number of stages s are determined by the implicit RK method used as corrector method.

4.2 Pool of configurations

The implementation of the corrector steps given by (2) leads to a nested loop structure with four dimensions iterating over:

1. the corrector steps ($k = 1, \dots, m$),
2. the argument vectors $\mathbf{Y}_l^{(k)}$ ($l = 1, \dots, s$),
3. the summands of $\sum_{i=1}^s a_{li} \mathbf{F}_i^{(k-1)}$ ($i = 1, \dots, s$),
4. the system dimension ($j = 1, \dots, n$).

The four-dimensional loop structure allows an application of typical loop transformations such as loop interchange, loop unrolling, or loop tiling. 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 or to the minimum energy consumption on a given hardware system.

Furthermore, many ODE systems, in particular those of large dimension which arise from partial differential equations (PDEs) discretized by the method of lines, are sparse, i.e., they are characterized by a right-hand-side (rhs) function \mathbf{f} which uses only a small number of components of the argument vector \mathbf{y} to compute one component of the function result. For many sparse ODE systems, an ordering of the components can be chosen such that the components of the argument vector \mathbf{y} accessed by the function evaluation are located within a limited index range near the component index evaluated. This property of the function \mathbf{f} , called limited access distance $d(\mathbf{f})$ [23], allows the implementation of specialized variants by overlapping

of vectors and using pipeline-like computational structure of the corrector steps [23]. Additionally, it also allows an efficient lock-based synchronization.

Table 1 summarizes the implementation variants. The variants [18, 23] exploit parallelism across the system, i.e., 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. This results in different utilizations of the cache and the memory hierarchy and consequently leads to different locality behaviors of the variants. General implementation variants in the pool can be applied to arbitrary ODE problems, where the rhs function $\mathbf{f}(t, \mathbf{y})$ may access all components of the vector \mathbf{y} . All general implementation variants require global communication and barrier operations. The variants E and EAblock are derived from the variant A and use the same data structures. They require two barrier operations per stage s . All other general variants are derived from the variant D and require only one barrier operation per corrector step m .

Specialized variants exploit a limited access distance of the function \mathbf{f} . These variants partition the ODE system into $n_B = n/B$ blocks. The block size B must be greater than or equal to the access distance, i.e., $B \geq d(\mathbf{f})$, such that the function evaluation of each block J depends only on the argument vector blocks $J - 1$, J and $J + 1$. Variants ppDb1m and ppDb1mt exploit the limited access distance by a loop interchange of the j and the k -loop, leading to a pipeline-like computational structure of the corrector steps. The j -loop running over the system dimension from $1, \dots, n$ becomes the outermost loop. Figure 5 illustrates the pipelined computation of the corrector steps for the case $m = 4$. The boxes in this figure correspond to blocks of the matrices $Y^{(k)}$ and to blocks of the approximation vectors $\hat{\mathbf{y}}_{\kappa+1}$ and $\mathbf{y}_{\kappa+1}$. The numbers inside the boxes indicate the order in which the blocks are computed. Because the access distance is limited, at most three blocks from the previous corrector step $k - 1$ are needed to compute one block in step k . For example, in Fig. 5, the blocks with numbers 13, 18, and 23 are required to compute block number 24. Thus, at each pipelining step (iteration of the j -loop with stride B) all blocks on one diagonal across the corrector steps can be computed. Since the length of the pipeline is m , we assume that the ODE system is partitioned into at least m blocks.

Specialized variants exchange data only between neighbors, global barriers are no longer required. Instead, specialized variants need only locks for the synchronization. Depending on the size of the ODE system and the time needed for the evaluation of function \mathbf{f} , the specialized implementation variants may achieve a higher scalability than the general variants.

Fig. 5 Computation order of the pipelining of the corrector steps used in ppDb1m and ppDb1mt for $m = 4$ corrector steps

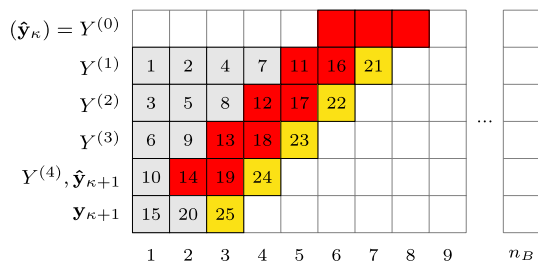


Table 1 Candidate pool of parallel implementation variants [18, 23]

Variant	Loop structure	Remarks
<i>General implementation variants</i>		
A	$k-l-i-j$	Vector-oriented: inner loops over system dimension; high spatial locality
E	$k-l-j-i$	Exploits temporal locality of the i -loop, i.e., writes to argument vector components
EAblock	$k-l-j-i-jj$	Similar to E, but loop tiling of the j -loop with the i -loop
D	$k-i-j-l$	Exploits temporal locality of the l -loop, i.e., reads from results of function evaluations
Dblock	$k-i-j-l-jj$	Similar to D, but loop tiling of the j -loop with the l -loop
PipeDe2m	$k-j-i-l$	Based on D; j -loop surrounds l - and i -loop; exploits temporal locality of the i - and the l -loop
PipeDb2m	$k-j-i-jj-l$	Similar to PipeDe2m, loop tiling of j -loop with i -loop
PipeDb2mt	$k-j-i-(jj)-l-jj$	Similar to PipeDb2m, but loop tiling expanded to the l -loop
<i>Specialized implementation variants</i>		
PipeDb1m	$k-j-i-jj-l$	Similar to PipeDb2m, but the vectors $\mathbf{Y}_l^{(k)}$ are overlapped to reduce space requirements
PipeDb1mt	$k-j-i-(jj)-l-jj$	Similar to PipeDb1m, but loop tiling expanded to the l -loop
ppDb1m	$j-k-i-jj-l$	Based on PipeDb1m; j - and k -loop are interchanged using a pipelining approach
ppDb1mt	$j-k-i-(jj)-l-jj$	Similar to ppDb1m, but loop tiling expanded to the l -loop

4.3 Experimental setup and evaluation

Figures 6 and 7 show the performance of different multithreaded implementation variants of the IRK method for two example problems: the 2D Brusselator equation (BRUSS2D) [14] and the vibrating string problem (STRING) [14]. The test problems were derived from PDE systems by a spatial discretization using the method of lines. BRUSS2D was discretized on an $N \times N$ grid and has a dimension $n = 2N^2$. The interleaving storage of grid points results in function \mathbf{f} accessing only $2N$ components of \mathbf{y} to compute one component of the function result, i.e., the function \mathbf{f} of BRUSS2D has a limited access distance $d(\mathbf{f}) = 2N$. The second test problem STRING results from the discretization of a 1D PDE using N grid points. STRING has dimension $n = 2N$ and a limited access distance $d(\mathbf{f}) = 3$. In the figure the time per step is plotted against the increasing number of threads.

The experiments have been performed on a system with two Intel Xeon E5-2697 v3 processors, each equipped with 14 cores, 35 MB shared cache and maximum frequency 2.6 GHz. In the experiments Intel Turbo Boost has been enabled. As RK method we use the LobattoIIIC (8) [14] method with $s = 5$ stages and $m = 7$ computation steps. Parallel variants of the ODE methods are developed using Pthreads and C, compiled with GCC 4.8.5 and -O3 optimization level.

Considering the diagrams in Figs. 6 and 7, it can be noticed that the fastest implementation variant depends on the system size, the number of threads executing the program and the ODE test problem. For the BRUSS2D problem and system size $N = 460$ or $n = 4232 \cdot 10^2$ the variant EAblock delivers the best performance for the thread numbers $p \leq 20$, closely followed by the variants E and A, whereas for the STRING problem and $n = 4232 \cdot 10^2$ the variant E is the fastest variant until $p \leq 8$, followed by the variants EAblock and A. For BRUSS2D with $n = 4232 \cdot 10^2$ and $p > 20$, the variants A, E, EAblock are the slowest implementations, while all other variants obtain similar performance. For the same system size and the STRING problem, the variants Dblock and PipeDb2mt offer the best performance for $p \leq 24$, whereas for $p = 28$ the specialized variants ppDb1mt and PipeDb1mt are the fastest.

For the BRUSS2D and $N = 2960$ or $n = 175232 \cdot 10^2$, the variant EAblock is the fastest variant for $p \leq 12$. For even larger number of threads, the specialized variants PipeDb1mt and PipeDb1m exhibit the best performance, closely followed by

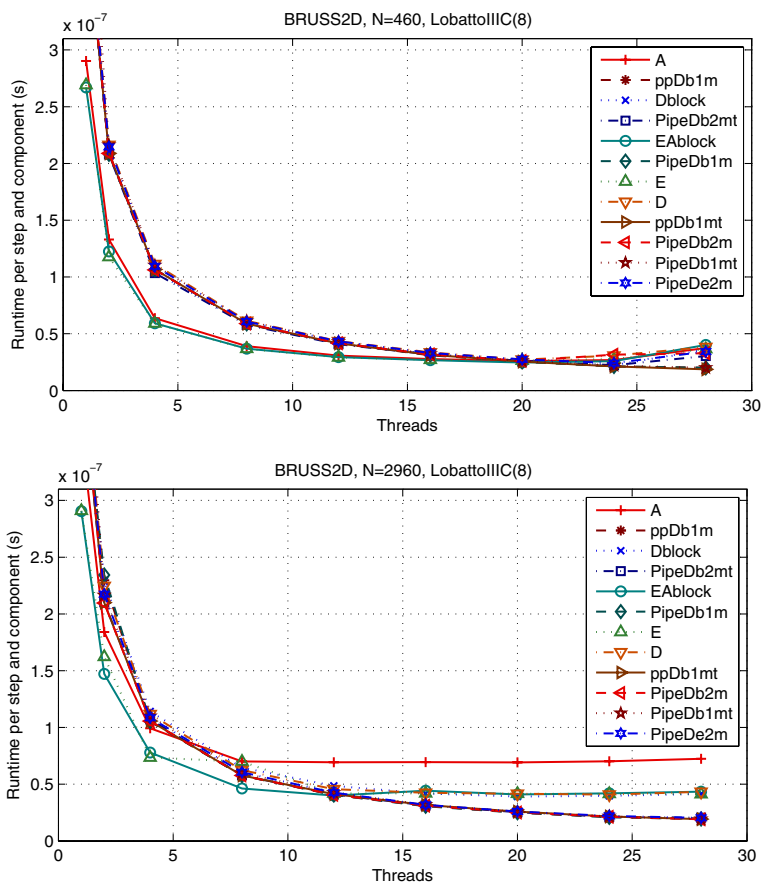


Fig. 6 Execution time per step and component of IRK method for BRUSS2D example for $n = 4232 \cdot 10^2$ (top) and $n = 175232 \cdot 10^2$ (bottom) for different numbers of threads

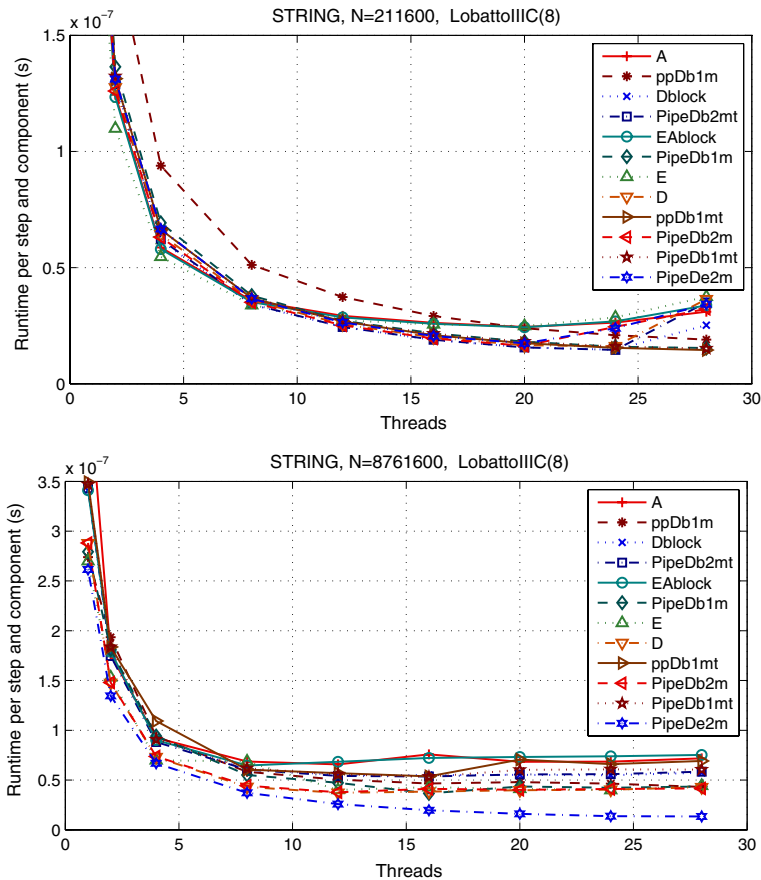


Fig. 7 Execution time per step and component of IRK method for STRING example for $n = 4232 \cdot 10^2$ (top) and $n = 175232 \cdot 10^2$ (bottom) for different numbers of threads

the variant PipeDb2mt. For the STRING problem with $n = 175232 \cdot 10^2$, the variant PipeDe2m is the fastest variant over the entire range of threads considered, the variant E is relatively closed to PipeDe2m up to thread numbers $p \leq 4$. For the STRING problem, the variants A and EAblock are the slowest variants for $p \leq 8$, the variant E ranks in the midfield. For the BRUSS2D problem the variant A is the slowest one, the variants E and EAblock ranks in the midfield.

The main reason for the low performance of the variants A, E and EAblock for an increasing number of threads are high synchronization costs. These variants require two barrier operations per stage in each computation step [for LobattoIIIC (8) 10 barriers in each corrector step m], whereas all other variants need only one barrier per corrector step or use locks for the synchronization of the threads.

The experiments in Figs. 6 and 7 indicate that ODE methods require online tuning due to the strong dependency on the input data. Further, the experiments show that for ODE methods an offline tuning can be beneficial for the online tuning. In

the offline phase micro-benchmarks can be used to measure the time for a barrier operation on the target platform for different numbers of threads to estimate the synchronization overhead of the general implementation variants. This information can be used to limit the number of variants evaluated in the online phase. In the online phase we can measure the runtime of the fastest specialized implementation variant and avoid the execution of the general implementation variants if their synchronization overhead is expected to be higher than the runtime of the best specialized implementation. Another idea is to use performance models in the offline phase, such as the ECM model, in order to predict the execution time of variants or at least to forecast their performance ranking [32].

Figure 8 demonstrates that for variants with loop tiling an appropriate selection of the tile size is important to achieve a good performance. The diagrams visualize the variation of the performance of the tiled implementation variants EAblock and PipeDb2mt for different numbers of threads for the BRUSS2D problem with $n = 175232 \cdot 10^2$. More precisely, the diagrams show by which percentage the execution time for a specific tile size is slower than the best execution time for the particular number of threads considered and over the entire range of tile sizes. The tile size is varied from 16 (representing two cache lines of 64 Byte each) to $n/28$ (maximum number of components per thread, if we have 28 threads) by doubling the tile size to generate the next sample point. For the variant EAblock, the maximum variation in the performance with respect to the best tile size sampled lies between 28% and 61%. For the variant PipeDb2mt, the maximum variation is between 26 and 170%. It can be observed that the best range of tile sizes depends on the implementation variant. For implementation variant EAblock and all thread numbers considered a good performance can be observed for small tile sizes in the range [16, 64], where the performance is less than 3% away from the optimum. Additionally, the runtime of a tiled implementation is also influenced by the number of threads executing the program. For tile sizes larger than 64 and up to tile size 65536, the maximum variation in the performance for 8 threads is 5–16%, for 12 threads 1–8% and for 20 and 28 threads $\leq 3\%$.

In the case of the variant PipeDb2mt, the best range of tile sizes depends on the number of threads considered. For 8 and 12 threads, the tile sizes in the range [128, 32768] perform best, for 20 threads in the range [128, 16384] and for 28 threads in the ranges [128, 512] and [2048, 16384] and are less than 3% away from the optimum.

Since the search space of possible tile sizes is very large, an exhaustive search over all tile sizes and for all tiled variants in the candidate pool is too expensive to be performed in the online phase. An offline phase can help to reduce the search space of potential tile sizes. In the offline phase, potentially good tile sizes can be preselected using an analytical model which is based on working spaces of the loop structures and describes the behavior of the implementation variants [18]. The preselected set of tile sizes can then be evaluated in the online phase.

Figure 9 shows the performance, the energy and the power consumption with respect to the clock frequency on the Intel Xeon E5-2630 system, equipped with 16 cores and 20 MB shared L3 cache. The IRK variants are executed with 16 threads as test example BRUSS2D with $n = 175232 \cdot 10^2$ is used. The energy

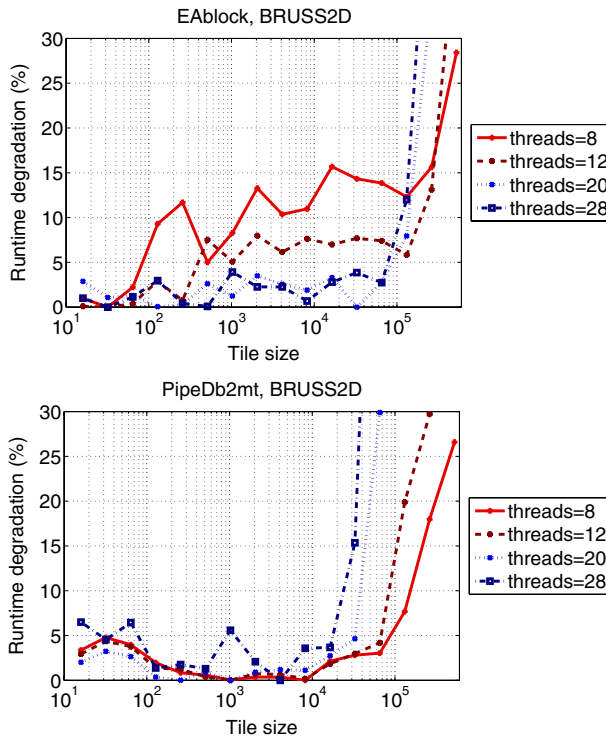


Fig. 8 Impact of the tile size on the performance of the EAblock (top) and PipeDb2mt (bottom) variants for different numbers of threads for BRUSS2D problem with $n = 175232 \cdot 10^2$

measurements do not include DRAM consumption and Turbo Boost is disabled. All implementations perform 60 time steps.

The results show that the best frequency depends on the optimization goal. The variant yielding the best performance for a given frequency is not necessarily the one with minimum energy consumption. When tuning towards the minimum execution time, the best time is achieved at a frequency of 2.4 GHz with the variant PipeDb1mt, while the smallest amount of energy is consumed by one of the variants PipeDb1mt or PipeDb2mt at a frequency of 2.2 GHz. Even though the variants PipeDb1mt and PipeDb2mt are not the fastest variants at this frequency, they show the lowest power usage. The variant leading to the best performance at 2.2 GHz is EAmt, but compared to the PipeDb1mt variant it consumes 15% more power and is only 3% faster. The experiments also show that the energy consumption of the variants is affected not only by the execution time but also by the power consumption.

We conclude that the variant and the frequency found to be the best for obtaining the minimum execution time are not necessarily the best for obtaining the lowest energy consumption and vice versa. In a multiobjective scenario, we would like to find the best configuration(s) (variant and parameter) for both objectives: minimum energy consumption and minimum execution time. One way to accomplish this is to

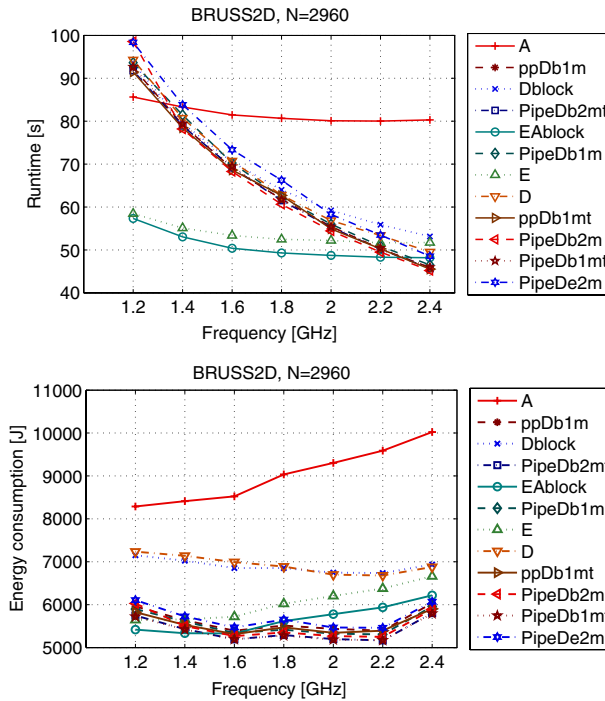


Fig. 9 Runtime, energy and power consumption of the different variants of the IRK method executed with 16 threads for varying clock frequencies. Test system BRUSS2D with $n = 175232 \cdot 10^2$

use fixed weight metrics such as the energy delay product (EDP) reducing the multiobjective problem to a single-objective problem [36]. Another approach is to find a set of non-dominated solutions (configurations), also called Pareto front. The Pareto front consists of solutions that cannot be improved according to an optimization goal without worsening another goal.

4.4 Summary of the observations for ODE solvers

The experiments in Sect. 4.3 show that the performance and the energy consumption of ODE methods are significantly influenced by the characteristics of the input data. Important factors are the access pattern of the rhs function \mathbf{f} and the number of equations constituting the problem, as well as the number of threads or parallel processes the ODE solver is start with. For different input data, different configurations (program variants and parameter values) may lead to the minimum energy consumption or the minimum execution time.

Since the complete input information is only available at runtime, ODE methods require online tuning. ODE solvers compute an approximate solution by performing a time-stepping procedure with a large number of time steps. This time-stepping nature of ODE methods can be exploited naturally for online tuning such that

the evaluation of the configurations already contributes to the solution process. A simple online tuning approach could be to use the first time steps of the integration to successively evaluate different configurations to find the best one and then to compute all remaining time steps with this configuration. The main obstacle for the application of such a pure online tuning approach is a large search space of code variants and parameters that would need to be tested at runtime. A lot of code variants can be generated by applying correctness-preserving program transformations. Furthermore, variants with loop tiling require an appropriate selection of the tile size to achieve good performance. Parallel implementations of ODE solvers offer further options for tuning, e.g., in the choice of data structures and their distribution among the threads or processes as well as communication and synchronization routines. The runtime and the energy behavior of parallel variants depend also on the number of threads or processes used and their mapping to the resources of the given hardware system. For the case of energy efficiency as objective function, the selection of the operational frequency for DVFS also plays an important role. The resulting search space is large and some of the configurations could be quite slow or energy intensive, leading to a high tuning overhead. Thus, in order to minimize the time spent at runtime in a searching process, online tuning should be combined with offline tuning.

For ODE methods a separate offline phase, performed before the first time step, can be used to shrink the search space and to identify the most promising configurations based on the information about the details of the hardware or with help of analytical models, heuristics and micro-benchmarks. Furthermore, the offline phase can be used to estimate synchronization and communication overheads for different configurations and input scenarios with the help of micro-benchmarks. This information can be used together with a performance model to predict a performance ranking of different implementation variants [32] and different parameter values. All evaluated configurations can be arranged in a decision tree according to their ranking, which is then forwarded to the online tuning phase.

In addition, the offline tuning phase can be used to preselect potentially good tile sizes for variants with loop tiling. For this purpose an analytical model can be build, based on working spaces of the loop structures of the implementation variants [18]. The preselected set of tile sizes can then be evaluated in the online phase. In general, it is sufficient to select a tile size that lies in a range of tile sizes with acceptable near-optimal performance. This range differs for different implementation variants and depends on the number of cores used.

5 Particle simulation methods for long-range interactions

In this section, we examine particle simulation methods for long-range interactions and explore their tuning potential. The section is structured in an introduction part of particle simulations, an explanation of the configurations, an evaluation part with experiments and a summary of this case study.

5.1 Particle simulation

Particle simulation methods are widely used in various areas of computational science, such as chemistry or physics. To simulate the interactions of the particles over time, the time is simulated in discrete time steps, in which new information for each particles, e.g., positions and velocities, is calculated.

Listing 1 shows the pseudocode of a time-step-based particle simulation. First the solver method has to be initialized. After reading the particle system, some solver-specific parameters can be set. Then, the time-step-based simulation is started. Each time step, first computes the particle interactions based on the current positions of the particles and then computes new positions and velocities of the particles based on the previously computed interactions. These positions are used as input for the following time step.

Listing 1 Pseudocode of the time-stepping structure of a particle simulation

```
Initialization of a solver method;
Set particle system properties;
Set solver specific parameters;

for (number of time steps) {
    Compute particle interactions;
    Compute new positions and velocities of the particles;
}

Termination of the solver method;
```

The calculation of the interactions of N particles with a direct method, i.e., the calculation of the interaction of each particle with every other particle, requires $O(N^2)$ operations, where N denotes the number of particles in the system. To reduce the complexity of the interaction calculation, there exist more efficient methods with splitting techniques which comprise near-field interactions with a direct method and separate far-field interactions, which are often approximated. This is especially crucial for particle simulations with long-range particle interactions, such as Coulomb or gravitational interactions, which have a strong impact in the resulting computation time.

The far-field interactions can be computed with Fourier-based methods, which often use the fast Fourier transforms (FFTs), e.g., the particle–particle non-equispaced FFT (P²NFFT). Fourier-based methods are efficient for homogeneous particle systems [15]. Another approach is the computation based on multipole expansions [13], e.g., with the fast multipole method (FMM).

Particle systems can be given in different particle distributions and can change their distribution over the time steps. Figure 10 illustrates the four particle systems, which are used in this article. The Cube and the Grid Face (first and second images in Fig. 10) are homogeneous particle systems in which the particles are uniformly distributed in the particle system. The Ball and Two Balls (third and fourth images in Fig. 10) are inhomogeneous particle systems. The particles distribution is irregular in these particle system, e.g., clustered in the examples.

5.2 Configuration of particle simulations

A configuration for a particle simulation consists of multiple pairs of values, with one pair describing a setting of the particle simulation. The configuration is split in three parts as described in Sect. 3 to set the algorithm parameter space, the program variant parameter space and the hardware parameter space. Each parameter space has a list of parameters, with a number of parameters being a possible configuration.

The algorithm parameter space includes the simulation algorithms like FMM and P²NFFT and their parallelizations, potentially using different programming models. For both algorithms, there exists an MPI parallelization and for the P²NFFT near-field part there also exists an OpenCL parallelization. The program variant parameter space includes solver-specific settings, such as the gridsize for P²NFFT or the maximum tree depth for FMM. These solver-specific settings can be set before the program starts without a new compilation. The hardware parameter space defines the compute node to use, e.g., a Xeon or an i7 system, and its specific settings, such as the operational frequency of the CPU or the number of MPI processes used. Figure 11 illustrates a set of possible configurations for particle simulations which will be used for the experimental evaluation. The illustrates the parameter space of the three parts, e.g., the gridsize of the P²NFFT algorithm with the MPI implementation can be set from 2 to 512. The hardware list consists of two compute nodes with different possible settings, depending on the available hardware on the compute node. The configuration 1 shows a possible configuration of how the particle simulation can be performed.

5.3 Experimental evaluation

Experimental setup The experiments with varying configurations have been performed on two HPC systems. The first system consists of two Intel Xeon E5-2683 v3 processors with 14 cores each, which have a maximum frequency of 2.0 GHz. Some experiments are also executed with an Intel Core i7-4770K desktop processor with 4 cores at 3.5 GHz. The energy measurements do only include the CPU, i.e., it does not include the energy consumption of the DRAM or any other component in the system. All measurements are repeated five times to obtain the average values shown in the figures. Intel Turbo Boost has been disabled for the experiments.

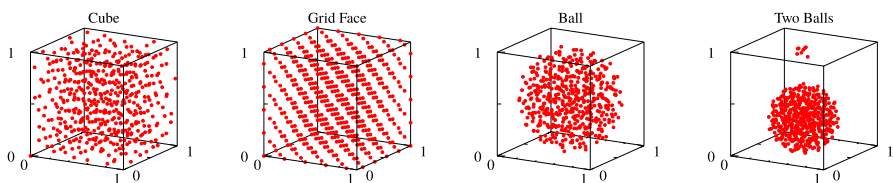


Fig. 10 Illustration of four particle systems. Two homogeneous (Cube and Grid Face) and two inhomogeneous systems (Ball and Two Balls)

The Scalable Fast Coulomb Solvers (ScaFaCoS) library is used for different particle simulation methods which are parallelized using MPI.

Experimental results and evaluation Figure 12 shows the execution time (left) and the energy consumption (right) of the FMM and P²NFFT solver applied to the four different particle systems illustrated in Fig. 10 and five particle system sizes on the Intel Xeon system with 56 MPI processes. The diagrams show that for small homogeneous particle systems, the P²NFFT solver is better, but for larger systems, i.e., more than 50,000 particles, the FMM solver outperforms the P²NFFT solver. If the particle system is inhomogeneous, the FMM solver outperforms the P²NFFT with a smaller system size than for homogeneous systems, e.g., 5000 particles for the Two Balls system. The results for the energy consumption confirm the observations, as they show the same behavior as the runtime. If the distribution and the number of particles are known before the first time step starts, an appropriate particle simulation solver can be chosen to obtain the lowest runtime or energy consumption.

Particle simulation solvers have specific parameters to tune their behavior by changing the split between near-field and far-field computations, e.g., by choosing the maximal tree depth for FMM or the gridsize for P²NFFT. Figure 13 depicts the optimal gridsize for the particle system Two Balls and Cube, i.e., an inhomogeneous and a homogeneous system. The optimal gridsize leads to a minimum runtime, energy consumption or even both. The Intel Xeon and the Intel Core i7 system are used with MPI processes that correspond to cores plus hyper-threading, i.e., 56 processes on the Intel Xeon and 8 processes on the Intel Core i7. The figure shows the optimal gridsize for both the runtime (left) and energy consumption (right). With more particles in the particle system the optimal gridsize generally increases. The optimal gridsize differs with the HPC system used and also the distribution of the particles. For a system with less cores, a smaller gridsize is generally the best for runtime and energy consumption. Also, some differences in the behavior of runtime and energy consumption can be seen. Since the optimal gridsize changes with the distribution of the system, after some time steps the parameters may have to be adjusted to reflect to possible changes.

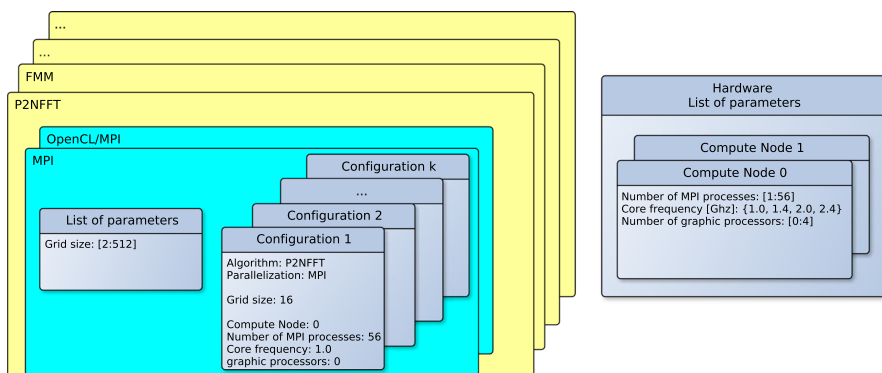


Fig. 11 Illustration of possible configurations for particle simulations

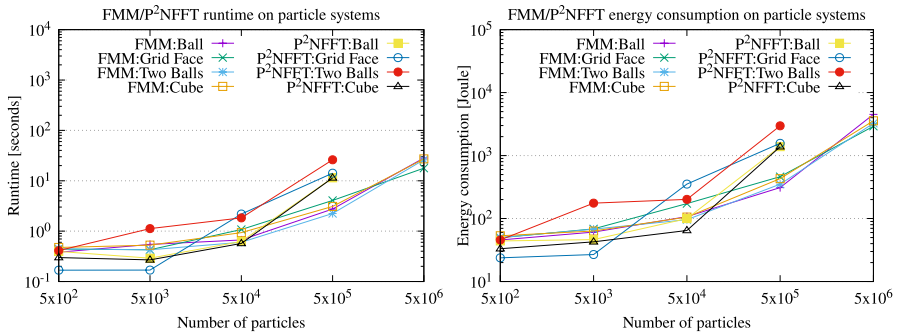


Fig. 12 Runtime (left) and energy consumption (right) of the FMM and P²NFFT solver on the Intel Xeon system for different particle systems with different numbers of particles

Figure 14 shows the runtime and energy consumption for varying processor frequencies. The two methods FMM and P²NFFT with the particle system Two Balls are considered on the Intel Core i7 system with 8 MPI processes. The experiments for frequencies are executed on the Intel Core i7, since it has a wider frequency range, i.e., 0.8 GHz to 3.5 GHz compared to 1.2 GHz to 2.0 GHz on the Intel Xeon system. The runtime is strongly decreasing with an increasing processor frequency. The energy consumption shows a slight U-shape with the lowest energy consumption at around 1.2 GHz. The best energy consumption for 5000 particles is achieved at the same frequency as for 50,000 particles; only with smaller particle systems a larger fluctuation of the results can be observed. Thus, to find the optimal processor frequency to obtain the lowest energy consumption for big particle systems a microbenchmark could be used that determines the optimal processor frequency for a small particle system with around 5000 particles.

An OpenCL implementation of the near-field part of the P²NFFT solver from the ScaFaCoS library was presented in [17]. Thus, the ScaFaCoS library is not bound to CPUs, but the use of GPUs is possible for the near-field part.

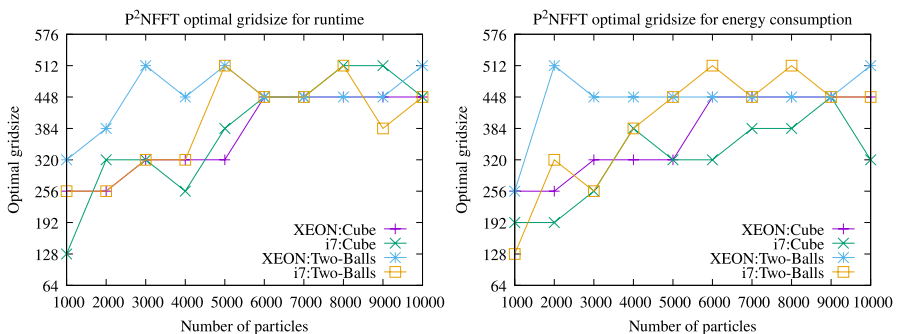


Fig. 13 Optimal gridsize of the P²NFFT solver with a homogeneous and an inhomogeneous system with different sizes on a Xeon system compared to the Intel Core i7 system

Figure 15 left shows the runtime of the near-field part tested on the Intel Xeon E5-2683 v3 processor and compared with the OpenCL implementation on a Nvidia Geforce GTX Titan Black for the same particle system. The y-axis shows a logarithmic scale. The runtime increases with increasing number of particles. This effect is bigger on the CPU. For small particle systems, the OpenCL overhead is too big for a good performance. The runtime on the GPU increases very slowly across all particle system sizes tested, since most of the time is needed for the transfer to the GPU and not for the computation. The CPU has a better runtime on the cube particle system than on the two balls particle system, but the reverse is true for the GPU. For small particle systems, the CPU is faster, but for bigger particle systems the GPU is faster.

Figure 15 right shows the runtime of the near-field part for different GPUs: (i) the Titan system with one and two GPUs; (ii) the Tesla system with a Nvidia Tesla C2050/C2075 Rev. a1 card. The figure shows that the runtime increases with increasing number of particles. The Tesla system is always faster than the Titan systems. The usage of two GPUs has more overhead than the usage of one GPU and thus is useful only for bigger particle systems. For small particle systems, the usage of just one GPU is faster.

The experiment leads to the following estimation for the offline and the online phase. The frequency can be set in the offline phase. For the lowest runtime, the highest frequency has to be selected, but for the lowest energy consumption a mid-range frequency is needed. The exact frequency can be determined by testing.

For a good estimation of the gridsize, the number of particles and the number of processes have to be considered, which are only available in the online phase. What also has to be considered is the fact that the particle system can change its distribution over time steps and, thus, some parameters may have to be adjusted during the simulation. Since the number of particles is constant, the choice of the particle simulation solver does not have to be changed, but its parameters might need an adjustment. For example, in the online tuning phase, the gridsize can be optimized by testing different solver parameters. After some time steps, the distribution of the particle system might have changed, thus, a different particle simulation solver parameter

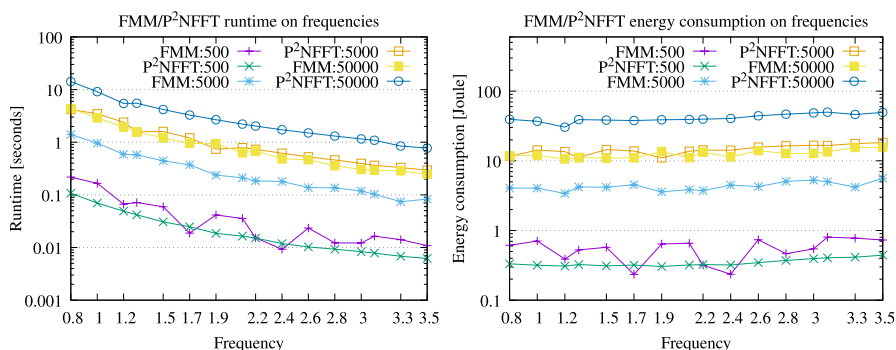


Fig. 14 Runtime (left) and energy consumption (right) depending on the processor frequency for different particle solver methods with the inhomogeneous Two Balls particle system on the Intel Core i7 system

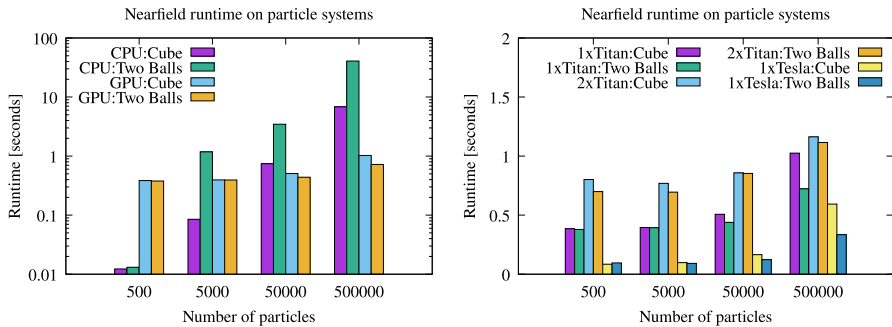


Fig. 15 Runtime of the near-field part of the P^2NFFT solver on an CPU compared with the OpenCL implementation on a GPU (left) and on different GPUs (right)

may have to be used to achieve a better performance or energy consumption. Therefore, after a certain number of time steps (e.g., after 1000 time steps) the parameter selection can be tested, e.g., by varying the gridsize during some time steps. If a performance gain can be observed when increasing or reducing the gridsize, the corresponding direction can be further investigated until the best performance is reached. The specific number of time steps after which the parameter selection is reviewed is significantly influenced by factors such as the chosen step size and the relative movement of particles between two successive time steps.

5.4 Summary of the observations for particle simulation methods

The experiments in the preceding subsection have shown that the performance and energy consumption of particle simulation solvers are influenced by the size and distribution of the input data, i.e., the particle system. For different particle system distributions or sizes, different configurations, i.e., choice of particle simulation solver and parameters, are required. Since the particles influence each other and thus change their position in the particle system, the particle system can change its distribution over time steps, which leads to a different configuration required to get the lowest runtime or energy consumption. Also, the choice of the hardware, which is part of the configuration, plays an important role. With the OpenCL implementation, the usage of GPUs is possible for particle simulations and useful for bigger particle systems, but has a too big overhead for small particle systems.

As shown by the benchmarks, the input data have to be known for a selection of an advantageous particle simulation solver. Hence, this selection has to be done with the start of the online tuning phase. Because the particle distribution for the particle simulation solver may change, a monitoring in the online tuning phase is required to adjust the parameters of the configuration. Observations for the processor frequencies have shown that the optimal settings for runtime and energy consumption can differ, i.e., highest frequency for lowest runtime, but a lower frequency for the lowest energy consumption. Optimal solver parameters, e.g., gridsize for P^2NFFT , vary too much, such that they have to be adjusted over time, i.e., by online tuning.

The offline phase for particle simulation solvers is required to set start parameters for the first time steps. A first configuration may be made based on available hardware. The selection for the processor frequency for DVFS can be set to maximum to get the best runtime. If the tuning goal is the lowest energy consumption, a lower frequency has to be set which varies with the given hardware. But as the particle simulation solver decision is highly dependent on the input data a decision of an optimal particle simulation solver cannot be done offline. Also, solver-specific parameters are too variable to be adjusted by offline tuning.

As both offline and online optimization are important for tuning particle simulation solvers, a combined approach that uses both phases is required. An offline tuning is required to get a good startup configuration for the particle simulation, and an online tuning to optimize some parameters, e.g., solver-specific parameters, and monitor the particle simulation runtime and energy consumption to achieve the best results in the time steps even after particle movements.

6 Conclusions

This article has proposed an extended online tuning process for time-step-based simulation methods which is based on an interaction between offline information gathering and online exploitation of this information. More precisely, the process builds up a pool of simulation codes which represent code configurations where a configuration is a program variant with additional decision. The online tuning can benefit from the pool by choosing an appropriate implementation variant. Two case studies have been investigated, and it has been shown how the relevant aspects for tuning can be extracted, relevant program variants can be identified and assessed by execution time and/or energy. For the particle simulation, different algorithm FMM and P2NFFT, different programming models MPI and OpenCL, different input data Ball and Cube and different hardware CPU and GPU have been chosen and assessed. For ODE solvers, two application problems leading to different right-hand sides of the equation to be solved are considered. In this case, the pool of implementation variants results from the application of code transformation.

Acknowledgements This work was supported by the German Ministry of Science and Education (BMBF) project SeASiTe, Grant No. 01IH16012A and No. 01IH16012B.

Funding Open Access funding provided by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.


References

1. Ansel J (2014) Autotuning programs with algorithmic choice. PhD thesis. Massachusetts Institute of Technology. <http://groups.csail.mit.edu/commit/papers/2014/ansel-phd-thesis.pdf>
2. Ansel J et al (2014) OpenTuner: an extensible framework for program autotuning. In: International Conferences on Parallel Architectures and Compilation Techniques. <http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf>
3. Ashouri AH et al (2017) Automatic tuning of compilers using machine learning. Springer, Berlin
4. Balaprakash P et al (2018) Autotuning in high-performance computing applications. *Proc IEEE* 106(11):2068–2083. <https://doi.org/10.1109/JPROC.2018.2841200>
5. Bilmes J et al (1997) Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: Proceedings of the 11th International Conferences on Supercomputing. ICS'97. Vienna, Austria, pp 340–347. ISBN: 0-89791-902-5
6. Christen M, Schenk O, Burkhardt H (2011) PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium
7. Fanfarillo A, Del Vento D (2019) AITuning: machine learning-based tuning tool for run-time communication libraries. *arXiv: 1909.06301* [cs.LG]
8. Feld D et al (2013) Facilitate SIMD-code-generation in the polyhedralmodel by hardware-aware automatic code-transformation. In: Proceedings of 3rd International Workshop on Polyhedral Compilation Techniques, pp 45–54
9. Frigo M, Johnson S (2005) The design and implementation of FFTW3. *Proc IEEE* 93(2):216–231
10. Gabriel E et al (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings of the 11th European PVM/MPI Users' Group Meeting, pp 97–104
11. Gerndt M, César E, Benkner S (eds) (2015) Automatic tuning of HPC applications—the periscope tuning framework. Shaker Verlag, Herzogenrath
12. Grosser T, Hoefler T (2016) Polly-ACC: transparent compilation to heterogeneous hardware. In: Proceedings of the 30th International Conference on Supercomputing (ICS'16)
13. Greengard L (1988) The rapid evaluation of potential fields in particle systems. MIT Press, Boston
14. Hairer E, Nørsett SP, Wanner G (1993) Solving ordinary differential equations I: nonstiff problems. Springer, Berlin
15. Hockney RW, Eastwood JW (1988) Computer simulation using particles. Taylor & Francis Inc, Bristol
16. Hofmann M, Kiesel R, Rünger G (2018) Energy and performance analysis of parallel particle solvers from the ScaFaCoS library. In: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE 2018). ACM, pp 88–95. ISBN: 978-1-4503-5095-2. <https://doi.org/10.1145/3184407.3184409>
17. Hofmann M et al (2018) A hybrid CPU/GPU implementation of computationally intensive particle simulations using OpenCL. In: 2018 17th International Symposium on Parallel and Distributed Computing (ISPDC), pp 9–16. <https://doi.org/10.1109/ISPDC.2018.00011>
18. Kalinnik N, Korch M, Rauber T (2014) Online auto-tuning for the timestep-based parallel solution of ODEs on shared-memory systems. *J Parallel Distrib Comput* 74(8):2722–2744. <https://doi.org/10.1016/j.jpdc.2014.03.006>
19. Kalinnik N et al (2018) Exploring self-adaptivity towards performance and energy for time-stepping methods. In: Proceedings of the 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2018). IEEE
20. Kalinnik N et al (2018) On the autotuning potential of time-stepping methods from scientific computing. In: Proceedings of the 11th Workshop on Computer Aspects of Numerical Algorithms (CANAL'18), vol 15. ACSIS, pp 329–338. <https://doi.org/10.15439/2018F169>
21. Karcher T, Pankratius V (2011) Run-time automatic performance tuning for multicore applications. In: Euro-Par 2011. Part I. Ed. by E. Jeannot, R. Namyst, and J. Roman. LNCS 6852, pp 3–14
22. Kiesel R, Rünger G (2019) Performance and energy evaluation of parallel particle simulation algorithms for different input particle data. In: Position Papers of the 2019 Federated Conference on Computer Science and Information Systems (FedCSIS 2019), 12th Workshop on Computer Aspects of Numerical Algorithms (CANAL'19), vol 19, pp 31–37. <https://doi.org/10.15439/2019F344>

23. Korch M, Rauber T (2007) Locality optimized shared-memory implementations of iterated Runge–Kutta methods. In: Euro-Par 2007. Parallel Processing, vol 4641. Springer LNCS. Springer, Berlin, pp 737–747
24. Moreno R et al (2020) Seeking the best weather research and forecasting model performance: an empirical score approach. *J Supercomput* 1:1. <https://doi.org/10.1007/s11227-020-03219-9>
25. Muralidharan S et al (2014) Nitro: a framework for adaptive code variant tuning. In: 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014), pp 501–512
26. Panyala A et al (2017) Exploring performance and energy tradeoffs for irregular applications. *J Parallel Distrib Comput* 104.C:234–251. <https://doi.org/10.1016/j.jpdc.2016.06.006>
27. Pfaffe P, Grosser T, Tillmann M (2019) Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping. In: Proceedings of the ACM International Conference on Supercomputing. ICS'19. ACM, pp 354–366. <https://doi.org/10.1145/3330345.3330377>
28. Püschel M et al (2005) SPIRAL: code generation for DSP transforms. *Proc IEEE* 93(2):232–275
29. Ragan-Kelley J et al (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13), pp 519–530
30. Raman A et al (2012) Parcae: a system for flexible parallel execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'12, pp 133–144
31. Rauber T, Rüniger G, Stachowski M (2019) Model-based optimization of the energy efficiency of multi-threaded applications. In: Sustainable Computing: Informatics and Systems 22, pp 44–61. ISSN: 2210-5379. <https://doi.org/10.1016/j.suscom.2019.01.022>
32. Seifert J et al (2018) Applicability of the ECM performance model to explicit ODE methods on current multi-core processors. In: High Performance Computing. Springer, pp 163–183
33. Sikora A et al (2016) Autotuning of MPI applications using PTF. In: Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications. SEM4HPC'16. ACM, pp 31–38. <https://doi.org/10.1145/2916026.2916028>
34. Tang Y et al (2011) The Pochoir stencil compiler. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11), pp 117–128
35. Tiwari A, Hollingsworth JK (2011) Online adaptive code generation and tuning. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011), pp 879–892
36. Tiwari A et al (2012) Auto-tuning for energy usage in scientific applications. In: Proceedings of the 2011 International Conference on Parallel Processing. Euro-Par'11. Springer, Bordeaux, France, pp 178–187
37. Whaley RC, Petitet A, Dongarra J (2001) Automated empirical optimizations of software and the ATLAS project. *Parallel Comput* 27(1–2):3–35
38. Yount C et al (2016) YASK-yet another stencil kernel: a framework for HPC stencil code-generation and tuning. In: Proceedings of the 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC. WOLFHPC'16. IEEE, pp 30–39. <https://doi.org/10.1109/WOLFHPC.2016.8>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Natalia Kalinnik¹ · Robert Kiesel² · Thomas Rauber¹  · Marcel Richter² · Gudula Rüniger²

Natalia Kalinnik
natalia.kalinnik@uni-bayreuth.de

Robert Kiesel
robert.kiesel@informatik.tu-chemnitz.de

Marcel Richter
marcel.richter@s2011.tu-chemnitz.de

Gudula Rünger
ruenger@cs.tu-chemnitz.de

¹ University of Bayreuth, Bayreuth, Germany

² Technical University Chemnitz, Chemnitz, Germany