Performance Prediction of Explicit ODE Methods on Multi-Core Cluster Systems

Markus Scherg
University of Bayreuth
Department of Computer Science
95440 Bayreuth, Germany
markus.scherg@uni-bayreuth.de

Johannes Seiferth
University of Bayreuth
Department of Computer Science
95440 Bayreuth, Germany
johannes.seiferth@uni-bayreuth.de

Matthias Korch
University of Bayreuth
Department of Computer Science
95440 Bayreuth, Germany
korch@uni-bayreuth.de

Thomas Rauber
University of Bayreuth
Department of Computer Science
95440 Bayreuth, Germany
rauber@uni-bayreuth.de

ABSTRACT

When migrating a scientific application to a new HPC system, the program code usually has to be re-tuned to achieve the best possible performance. Auto-tuning techniques are a promising approach to support the portability of performance. Often, a large pool of possible implementation variants exists from which the most efficient variant needs to be selected. Ideally, auto-tuning approaches should be capable of undertaking this task in an efficient manner for a new HPC system and new characteristics of the input data by applying suitable analytic models and program transformations.

In this article, we discuss a performance prediction methodology for multi-core cluster applications, which can assist this selection process by significantly reducing the selection effort compared to in-depth runtime tests. The methodology proposed is an extension of an analytical performance prediction model for shared-memory applications introduced in our previous work. Our methodology is based on the execution-cache-memory (ECM) performance model and estimations of intra-node and inter-node communication costs, which we apply to numerical solution methods for ordinary differential equations (ODEs). In particular, we investigate whether it is possible to obtain accurate performance predictions for hybrid MPI/OpenMP implementation variants in order to support the variant selection. We demonstrate that our approach is able to reliably select a set of efficient variants for a given configuration (ODE system, solver and hardware platform) and, thus, to narrow down the search space for possible later empirical tuning.

KEYWORDS

performance prediction; variant selection; auto-tuning; ODE; ECM model; MPI; distributed-memory

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3310306

ACM Reference Format:
Markus Scherg, Johannes Seiferth, Matthias Korch, and Thomas Rauber. 2019. Performance Prediction of Explicit ODE Methods on Multi-Core Cluster Systems. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3297663.3310306

1 INTRODUCTION

In order to achieve a high efficiency, scientific applications typically need to be adapted to the characteristics of a specific *high-performance computing* (HPC) platform. Considering the growing diversity and complexity of modern computer architectures, this often means a large programming effort for software developers, since the performance of parallel programs strongly depends on the characteristics of the target platform, such as processor design, cache architectures, memory latency, memory and network bandwidth. The best implementation variant selected for one HPC platform is not necessarily the best, or even a good implementation variant, for another platform. Hence, to obtain optimal performance, applications might need to be tuned for each specific target platform anew. This can be a time-consuming and costly process.

A promising concept to avoid this manual tuning effort is *auto-tuning* (AT). Many different approaches have been proposed to automatically tune software. AT is based on two core concepts (i) to generate implementation variants of an application based on program transformation and optimization techniques such as loop unrolling or loop tiling for the compute-intensive inner kernels, and (ii) to select the implementation variant with the highest efficiency on the target platform from the set of generated implementation variants.

A big challenge in AT is to time-efficiently select a suitable implementation variant from a potentially large number of possible variants. A straightforward but time-consuming way to find this particular implementation variant is the comparison of implementation variants by runtime experiments, which might be steered by an exhaustive search or by more sophisticated mathematical optimization methods, such as the Nelder-Mead method [15] or differential evolution [6]. Alternatively, analytic performance models can be used to select the most efficient implementation variant directly or to filter out poorly performing implementation variants

in order to reduce the number of runtime experiments necessary. This requires the performance prediction to be accurate enough to clearly distinguish and rank the performance of the different implementation variants.

The contribution of this paper is

(1) the development of a performance prediction methodology for hybrid MPI/OpenMP implementation variants on multi-core cluster systems,
(2) the application to a complex numerical method with complicated runtime behavior: the parallel solution of *initial value problems* (IVPs) of *ordinary differential equations* (ODEs),
(3) the validation of the accuracy of the prediction for different settings (ODE system, inter-node communication and hardware architecture),
(4) a discussion of its applicability in the context of AT.

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 outlines the computational structure of the ODE solution method used as example and describes the implementation variants considered. In Section 4, the performance prediction methodology is discussed in detail. Section 5 describes the experimental setup, the results of which are presented in Section 6. Section 7 concludes the paper and gives an overview of future work.

2 RELATED WORK

In general, a distinction can be made between *offline* and *online* AT techniques. Offline AT is performed at installation or compile time and selects the supposedly most efficient implementation without actual knowledge of the input data. Thus, these approaches are appropriate for applications where characteristics of the input data only have a minor impact on the selection of the best implementation, as it is the case, e.g., for dense linear algebra problems. Examples for offline approaches from this field of application are, i.a., ATLAS [29] and PhiPAC [2]. In other fields such as sparse linear algebra or particle codes, however, the execution behavior is heavily influenced by properties of the input data. Online AT takes these influences into account by choosing the best variant at runtime when all input is known. Active Harmony [26] and Periscope [8] are examples for online approaches.

Several AT 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 or offline phase. An important application area for these approaches are stencil computations, see PATUS [5], Pochoir [24] and Halide [18] for approaches in this direction.

Performance modelling approaches can roughly be divided into two categories: *black box* models that apply statistical methods and machine learning techniques to observed performance data like hardware metrics in order to describe and predict performance behavior [23, 25] and *white box* models that describe the interaction of hardware and code using simplified machine models [22, 27, 30]. Further, model generation tools are covered for numerous application scenarios in prior work. The Empirical Roofline Toolkit (ERT) [13] characterizes the throughput potential of a kernel executed on a node using the Roofline model. *Kerncraft* [12] can be applied to loop kernels to obtain node-level ECM model predictions.

In previous work [21], we demonstrated the applicability of the ECM performance model for the selection of the most suitable implementation variant on *shared-memory* systems for a complex algorithm from numerical analysis, explicit ODE methods. These solution methods compute a numerical approximation for the solution by performing a series of time steps [10]. Each time step consists of the computation of a fixed number of stage vectors which are then combined to the next approximation vector for the unknown solution. Overall, these methods exhibit a complex loop structure modifiable by loop transformations, yielding a large set of implementation variants, whose performance behavior depends on the organization of the computations and the memory accesses within each time step, but also on the characteristics of the ODE system to be solved. To solve IVPs of ODEs systems, various numerical methods can be used. In addition to the classical explicit and implicit *Runge-Kutta* (RK) and *multi-step* methods [10], many specific methods aiming at exploiting parallelism, e.g., *parallel iterated RK* (PIRK) methods [16, 28], waveform relaxation methods [3], and peer two-step methods [20] have been proposed.

In this paper, as an enhancement of our previous work, we propose a performance prediction methodology for hybrid MPI/OpenMP implementation variants of PIRK methods suitable for *multi-core cluster systems*, which combines a white and black box model in order to predict the runtime of specific implementation variants. The ECM model (white box model) is applied to predict the node-level performance of the PIRK implementation variants considered. To estimate the intra-node and inter-node communication costs of the particular variants, a black box model is used to assess the costs of the particular communication and synchronization operations executed.

3 PARALLEL ITERATED RUNGE-KUTTA METHODS

We study PIRK methods as a representative example of the general class of explicit ODE methods to evaluate the performance prediction methodology proposed. PIRK methods perform a series of time steps one after another until the end of the given integration interval is reached. Within each time step, PIRK methods exhibit a four-dimensional loop structure. By applying different loop transformations to this loop structure, a large pool of possible implementation variants can be generated. As the performance characteristics of these variants can potentially vary highly depending on

(a) the target hardware,
(b) the compiler used and its flags,
(c) the number of processes or threads employed,
(d) the ODE system to be solved, and
(e) the number of stages of the selected base ODE method,

an accurate performance prediction would be valuable for either directly identifying the most efficient implementation variant or for pre-selecting a set of candidate implementation variants for further AT steps.

```
for (k = 0; k < m; k++)
  communication()
  RHS
  synchronization()
  LC
APPROX
UPDATE
```

Listing 1: Basic abstract PIRK implementation considered.

3.1 Computational structure of PIRK methods

PIRK methods are part of the class of one-step predictor-corrector methods, i.e, in each time step t_κ a new approximation of the solution is computed by an iterative process. The iteration process is started with an initial approximation of the solution function which we select as:

$$\mathrm{Y}_l^{(0)} = \mathrm{y}_\kappa, \quad l = 1, \ldots, s \quad . \tag{1}$$

Next, the corrector method, an s-stage implicit RK method [11] of order o, is applied a fixed number of $m = o - 1$ times:

$$k = 1, \ldots, m:$$

$$\mathrm{Y}_l^{(k)} = \mathrm{y}_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mathrm{F}_i^{(k-1)}, \quad l = 1, \ldots, s \tag{2a}$$

$$\text{with } \mathrm{F}_i^{(k-1)} = \mathbf{f}(t_\kappa + c_i h_\kappa, \mathrm{Y}_i^{(k-1)}) \ , \tag{2b}$$

where $\mathbf{f}$ is the *right-hand-side function* (RHS) of the ODE system solved. The coefficient matrix $A = (a_{ij}) \in \mathbb{R}^{s,s}$, the weight vector $\mathbf{b} = (b_i) \in \mathbb{R}^s$, the node vector $\mathbf{c} = (c_i) \in \mathbb{R}^s$, and the order o are given by the implicit RK method used as corrector method. After the iteration process, the new approximation $\mathrm{y}_{\kappa+1}$ is computed as follows:

$$\mathrm{y}_{\kappa+1} = \mathrm{y}_\kappa + h_\kappa \sum_{i=1}^{s} b_i \mathrm{F}_i^{(m)}, \tag{3}$$

For efficient step size control, an error vector can be computed by:

$$\mathbf{e} = h_\kappa \sum_{i=1}^{s} b_i \left(\mathrm{F}_i^{(m)} - \mathrm{F}_i^{(m-1)} \right) \quad . \tag{4}$$

Based on the difference between the norm of $\mathbf{e}$ and a user-defined tolerance, time steps may be accepted or rejected, and the step size can be increased or decreased.

3.2 Pool of PIRK implementation variants

In this section, we present the PIRK implementation variants used in our experimental evaluation. All implementation variants considered focus on parallelism across the ODE system, i.e., the n equations of the ODE system are partitioned among all p nodes and their c logical cores using a blockwise distribution. Thus, each node is assigned a block of $\approx n/p$ consecutive components and each of its cores computes a block of $\approx n/(p \cdot c)$ components. The independence of the stages is exploited, however, by reducing the number of synchronization and communication operations. The inter-node communication is realized with MPI, the parallelization within the nodes is done by OpenMP.

To explore the applicability of the proposed performance prediction methodology, we apply it to a specific base implementation (Listing 1). Apart from the first trivial predictor step (which is not shown in the listing), the implementation consists of a loop over the m corrector steps, which must be processed sequentially. The implementation separates a corrector step into two non-overlapping kernels, *LC* and *RHS*. Kernel *RHS* calculates the function evaluations of the right-hand-side functions (2b), which are needed for the linear combinations in kernel *LC* (2a). At the beginning of each corrector step, there is a communication phase which ensures that the required components of the argument vectors from the last iteration are available for kernel *RHS* at all processes. After the computation of the corrector steps two more kernels (*APPROX* and *UPDATE*) are executed that calculate the new approximation $\mathrm{y}_{\kappa+1}$ (3). The base implementation uses two $s \times n$ matrices to store $F^{(k)}$ (`F`) and $Y^{(k)}$ (`Y`) and two n-vectors for the input/output approximation y (`y`) and the difference between the input and the output approximation $\mathrm{y}_{\kappa+1} - \mathrm{y}_\kappa$ (`dy`). In addition, one $s \times s$ matrix and two s-vectors are required for the coefficients A, $\mathbf{b}$, $\mathbf{c}$ of the corrector method. To simplify the analysis, step control is not yet considered.

```
for (i = 0; i < s; i++)
  for (j = first; j <= last; j++) {
    for (l = 0; l < s; l++)
      Y[l][j] += a[l][i] * F[i][j]; }
  for (j = first; j <= last; j++)
    for (l = 0; l < s; l++)
      Y[l][j] = Y[l][j] * h + y[j];
```

Listing 2: Implementation *ijl* of kernel *LC*.

```
for (i = 0; i < s; i++) {
  for (l = 0; l < s; l++)
    for (j = first; j <= last; j++)
      Y[l][j] += a[l][i] * F[i][j]; }
for (l = 0; l < s; l++)
  for (j = first; j <= last; j++)
    Y[l][j] = Y[l][j] * h + y[j];
```

Listing 3: Implementation *ilj* of kernel *LC*.

```
for (j = first; j <= last; j++) {
  for (i = 0; i < s; i++) {
    for (l = 0; l < s; l++)
      Y[l][j] += a[l][i] * F[i][j]; }
  for (l = 0; l < s; l++)
    Y[l][j] = Y[l][j] * h + y[j]; }
```

Listing 4: Implementation *jil* of kernel *LC*.

```
for (j = first; j <= last; j++)
  for (l = 0; l < s; l++) {
    for (i = 1; i < s; i++)
      Y[l][j] += a[l][i] * F[i][j];
    Y[l][j] = Y[l][j] * h + y[j]; }
```

Listing 5: Implementation *jli* of kernel *LC*.

```
for (l = 0; l < s; l++)
  for (j = 0; j <= last; j++) {
    for (i = 0; i < s; i++)
      Y[l][j] += a[l][i] * F[i][j];
    Y[l][j] = Y[l][j] * h + y[j]; }
```

Listing 6: Implementation *lij* of kernel *LC*.

```
for (l = 0; l < s; l++) {
  for (i = 0; i < s; i++)
    for (j = first; j <= last; j++)
      Y[l][j] += a[l][i] * F[i][j];
  for (j = first; j <= last; j++)
    Y[l][j] = Y[l][j] * h + y[j]; }
```

Listing 7: Implementation *lji* of kernel *LC*.

The implementation of kernel *LC* in (2a) leads to a nested three-dimensional loop structure which iterates over:

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

These loops (l-, i-, and j-loop), are independent of each other and fully permutable, which results in six possible implementations of kernel *LC* (Listing 2 to 7) and, hence, six possible implementation variants. Different loop permutations lead to implementation variants with a high spatial locality and a high potential for SIMD vectorization by the compiler or to implementation variants that enable temporal reuse of argument vector components in write operations corresponding to updates of the sum $\sum_{i=1}^{s} a_{li}\mathbf{F}_i^{(k-1)}$ in (2), but which also reduce the spatial locality.

If we do not make assumptions about the access pattern of kernel *RHS*, we must exchange all components of the argument vector between the processes. Thus, the communication phase must perform a multi-broadcast operation (MPI_Allgather). Many problems, however, require the component functions $f_j(t, \mathbf{y})$ to access only a few components of the argument vector $\mathbf{y}$. If this is the case, we call the problem *sparse*, otherwise *dense*. A special case of sparse problems are problems with *limited access distance*, where f_j needs to access only the components $y_{j-d(\mathrm{f})}$ to $y_{j+d(\mathrm{f})}$, where $d(\mathbf{f})$ denotes the access distance of the problem. Problems with limited access distance allow neighborhood communication, where process p_i has to communicate with p_{i-1} and p_{i+1} exclusively and only small messages are exchanged. In this paper, we consider three implementations of the communication phase: the first implementation uses multi-broadcast operations, as required by dense problems, while the second uses neighborhood communication, as enabled by a limited access distance. For sparse problems without limited access distance we offer a third communication option in which the required components can be exchanged between all processes. We refer to this communication as *sparse communication*.

4 PERFORMANCE PREDICTION OF ODE SOLVERS ON MULTI-CORE CLUSTER SYSTEMS

In this work, we introduce a performance prediction methodology for ODE solvers executed on multi-core cluster systems. The methodology proposed is an enhancement of an approach for shared-memory ODE solvers we discussed in previous work [21], where we only studied single-node performance. Our prediction formula is capable of predicting the time required to execute a single time step of a particular implementation variant given its base ODE method, the size of the ODE system solved and characteristics of the target hardware architecture(s).

Our methodology consists of two core components, (i) a node-level runtime prediction of an implementation variant (Section 4.1), and (ii) an estimate of the intra-node and inter-node communication costs of the implementation variants (Section 4.2). To obtain a node-level runtime estimate of an implementation variant, first its basic computation kernels need to be identified. For the basic PIRK implementation studied in this work (Listing 1), these are the kernels *RHS*, *LC*, *APPROX*, and *UPDATE*. Next, ECM model predictions are determined for each kernel using the *kerncraft* tool (version 0.7.0, https://github.com/RRZE-HPC/kerncraft), and these ECM predictions are again combined to a total node-level runtime prediction for a time step of the implementation variant. The intra-node (OpenMP barrier operations) and inter-node (MPI communication operations) communication costs of an implementation variants, on the other hand, are determined using a black box model which estimates the costs depending on the message size for MPI operations and the number of threads for barrier operations using linear regression.

A decisive feature of our prediction-based methodology is that it is capable of re-using previously obtained individual components (ECM predictions of a particular kernel, costs of particular communication operations, ...) of its total prediction when giving predictions for additional implementation variants or IVPs. In the context of an AT procedure, this is a big advantage compared to techniques like variant sampling. While variant sampling would require to run all newly added implementations variants or even require to re-run all implementation variants available when adding an additional IVP, only the new ECM predictions must be made available to our methodology in order to give the new predictions. For example, when switching from IVP *Heat2D* to IVP *InverterChain*, only a single *kerncraft* run is necessary to obtain the ECM prediction for *InverterChain*'s implementation of the *RHS* kernel, while sampling would result in a re-run of all implementation variants available. Besides, prediction reuse can also be exploited when adding further implementation variants. For example, new implementation variants could be derived by fusing kernels *APPROX* and *UPDATE* (Listing 1) into a single kernel. With our methodology this again only requires one additional *kerncraft* run.

Certain assumptions are included in the methodology presented:

(1) The multi-core cluster systems consist of p homogeneous nodes p_i, each with c logical cores.
(2) The inter-node communication costs are roughly the same.
(3) Homogeneous nodes imply that all nodes have identical caches and that the number δ of data elements fitting into one cache line (CL) of a particular cache hierarchy level is the same on all nodes. As a simplification, we further assume that δ is equal on all cache levels.
(4) The cores of all nodes run at the same CPU frequency f.
(5) The n components of the ODE system are distributed to the cluster nodes and their logical cores in blockwise order. Hence, each node p_i covers a block of $n_p \approx n/p$ consecutive components and each of its cores computes $n_c \approx n/(p \cdot c)$ of these components.

4.1 Node-level runtime prediction

The time required to execute a given task on a multi-core cluster system is determined by the node that takes the longest time to finish its assigned subtask(s). A task can not be completed before all its subtasks have finished. Hence, understanding and being able to accurately predict the performance on a node-level is vital in optimizing a multi-core cluster application. Further, the gained knowledge could be used in an AT approach to minimize the runtime of an application by redistributing the workloads to the individual cluster nodes.

In previous work, we demonstrated the applicability of a performance prediction formula to shared-memory implementation variants of explicit ODE solvers [21]. Here, we use a similar methodology to predict the node-level performance of the multi-core cluster implementation variants considered. Compared to our previous work, however, we do not calculate node-level predictions for the entire ODE system, but only for the specific block of n_p components computed by a particular node p_i. The following is just a brief summary of the node-level performance prediction approach. For a detailed description, we refer to our previous work.

4.1.1 ECM model. Basis of our node-level prediction is the ECM performance model [22, 27], which is an analytic performance model capable of estimating the number of required CPU cycles per CL to execute a given number of loop iterations on a multi- or many-core chip. This estimation includes contributions from the data transfer time T_{data} and the in-core execution time T_{core}:

$$T_{\text{data}}^{\text{L3}} = T_{\text{L1L2}} + T_{\text{L2L3}} \ , \tag{5}$$

$$T_{\text{core}} = \max(T_{\text{OL}}, \ T_{\text{nOL}}) \ , \tag{6}$$

and is defined as:

$$T_{\text{ECM}}^{\text{level}} = \max(T_{\text{OL}}, \ T_{\text{nOL}} + T_{\text{data}}^{\text{level}}) \ . \tag{7}$$

$T_{\text{data}}^{\text{level}}$ describes the time it takes to transfer all data required from its location in the memory hierarchy to the L1 cache and back. The contributions of the single hierarchy levels (T_{L1L2}, T_{L2L3}, T_{L3Mem}) are determined depending on the amount of transferred CLs. $T_{\text{data}}^{\text{level}}$ is exemplary shown in (5) for data coming from the L3 cache. T_{core} is defined as the time spent executing instructions in the core under the assumptions that all data are in the L1 cache and that all instructions are scheduled independently to the ports of the units. Therefore, the unit that takes the most cycles to execute its instructions determines T_{core}. Further, the model assumes that single-core performance scales linearly with the cores until a shared bottleneck is saturated and names the core count necessary to saturate that bottleneck. This core count is called the loop's *performance saturation point*.

To obtain a prediction of the single core execution time $T_{\text{ECM}}^{\text{level}}$ (7), the in-core execution and data transfer times are combined. Therefore, the ECM model determines which of the runtime contributions can overlap with each other. T_{OL} is the part of the core execution that overlaps with the data transfer time and T_{nOL} the part that does not (6).

4.1.2 Node-level runtime prediction θ. Using ECM model predictions, a node-level runtime prediction θ_ϵ of an implementation variant ϵ can be derived. In a first step, this requires to identify the set λ, which consists of all basic computation kernels of ϵ. For each of these kernels, *kernel prediction* ζ_λ is determined by:

$$\zeta_\lambda = \frac{\alpha_\lambda \beta_\lambda}{\delta} \ . \tag{8}$$

ζ_λ yields the number of cycles necessary to execute λ, where α_λ is defined as the number of cycles required to execute one CL of data (e.g., the ECM model prediction given by (7)). To obtain the total number of cycles required to execute λ, α_λ is multiplied by the number of iterations executed β_λ and divided by the number of data elements δ fitting into one CL (e.g. eight doubles, each eight bytes, per CL on our target nodes). From this kernel prediction ζ_λ given in cycles, the *kernel runtime prediction* ϕ_λ in seconds can than be derived:

$$\phi_\lambda = \frac{\zeta_\lambda}{f \cdot \min(\tau, \sigma_\lambda)} \ , \tag{9}$$

where f is the CPU frequency of the target node p, τ the number of threads running on this node and σ the saturation point of λ. Given the kernel runtime predictions ϕ_λ of its basic kernels λ and an estimate of its intra-node communication costs, the *node-level runtime prediction* θ_{ϵ_p} of an implementation variant ϵ running on node p is defined as:

$$\theta_{\epsilon\_p} = \sum_\lambda \phi_\lambda + t_{\text{com\_node\_p}} \ , \tag{10}$$

where $t_{\text{com\_node\_p}}$ factors in the intra-node communication costs on node p when executing τ threads.

Remark: When a non-perfectly nested loop is split into several kernels, a prediction error may be introduced, because the split loop structure may have a different reuse pattern. In our previous work, we showed that this error is acceptable for this kind of application [21].

4.2 Estimating the MPI communication costs

Since we assume that the communication costs between all nodes are similar, the time spent on communication depends only on the MPI operations used. The implementation variants for dense problems use MPI_Allgather, while the implementation variants with neighborhood communication use MPI_Isend and MPI_Irecv. There are numerous benchmarks that identify the cost of MPI operations, such as the *Intel MPI Benchmark* [7], *SKaMPI* [19] and the *OSU Micro-Benchmarks* [17]. We use a modified version of the Intel MPI Benchmark and some self-created benchmarks to study the runtimes of the above mentioned MPI operations by benchmarking different process numbers and message sizes and use linear interpolation to estimate the communication costs for unknown message sizes.

4.3 System-level runtime prediction

The time required to execute a given task on a multi-core cluster system is determined by the node that takes the longest time to finish its assigned subtask(s). To determine the *system-level runtime prediction* ψ, thus, we need to identify on which node p_i the node-level runtime prediction is maximal. Given a multi-core cluster system containing homogeneous nodes p_i, the time required to execute a single time step of an ODE system of size n using an implementation variant ϵ is defined as:

$$\psi_\epsilon = \max \ \theta_\epsilon + t_{\text{com\_dm}} \ , \tag{11}$$

where $\theta = (\theta_1, \theta_2, \dots, \theta_i, \dots)$ are the node-level runtime predictions of the system's nodes p_i obtained for the execution of the node's share of $n_p = n/p$ components. Term $t_{\text{com\_dm}}$ takes into account the inter-node communication costs introduced when solving an ODE system of size n.

```
// Constants: N12
N12 * (y[j-N] + y[j-1] - 4 * y[j] + y[j+1] + y[j+N];
```

Listing 8: Kernel code fragment of IVP *Heat2D*.

```
// Constants: Uop, R, eta, Uthr, C
((Uop - y[j]) / R - (eta * ((y[j-1] - Uthr) * (y[j-1] - Uthr)
   - (y[j-1] - y[j] - Uthr) * (y[j-1] - y[j] - Uthr)))) / C;
```

Listing 9: Kernel code fragment of IVP *InverterChain*.

```
// Constants: dz, dz2, c
// z = (j + 1) * dz; t = (dz - 1.0) * (z - 1.0) / c;
// a = 2.0 * (z - 1.0) * t / c; b = t * t;
(a * y[j+1] - y[j-1]) / (2.0 * dz)
   + (y[j-1] - 2.0 * y[j] + y[j+1] * b / dz2)
   - k * y[j] * y[j+N]; // 1. PDE
- k * y[j+N] * y[j-1]; // 2. PDE
```

Listing 10: Kernel code fragments of IVP *Medakzo*.

```
// Constants: delta_x
-1.0 / delta_x * (-4.0 / 5.0 * (y[i - 1] - y[i + 1])
   + 1.0 / 5.0 * (y[i - 2] - y[i + 2])
   - 4.0 / 105.0 * (y[i - 3] - y[i + 3])
   + 1.0 / 280.0 * (y[i - 4] - y[i + 4]));
```

Listing 11: Kernel code fragment of IVP *Wave1D*.

```
// Constants: K2, dx2
y[i + N]; // displacement
K2 * (y[i - 1] - 2 * y[i] + y[i + 1]) / dx2; // velocity
```

Listing 12: Kernel code fragment of IVP *String-Row*.

```
// Constants: K2, dx2
y[i + 1]; // displacement
K2 * (y[i - 3] - 2 * y[i-1] + y[i + 1]) / dx2; // velocity
```

Listing 13: Kernel code fragment of IVP *String-Mix*.

5 EXPERIMENTAL SETUP

In the following subsection we present the IVPs used for the study of our performance prediction methology. The IVPs considered exhibit different characteristics regarding their execution time (compute-bound, memory-bound, mixed behavior), their access distance (limited, unlimited) and their occupancy rate (sparse, dense), and are therefore suitable for the investigation of our prediction methodology. Afterwards we describe the computer system on which we performed the experiments and general settings that apply to all measurements.

5.1 Characteristics of the IVPs considered.

For large ODE systems with a sparse access pattern, the RHS function evaluation in (2b) only touches a small number of components, thus reducing the inter-node communication traffic required. Hence the computation of the linear combination in (2a) dominates the runtime and there is demand to auto-tune the linear combination. These systems should be quite accurately predictable by our methodology because they tend to have some regular access pattern. Even if the prediction would not be as accurate for more complex sparse ODE systems, still the majority of the runtime will be spent in the LC, thus mitigating the less accurate prediction of the kernel *RHS*.

As examples, we have selected five sparse ODE systems, which are described in detail in the following. Listings 8 to 13 show the code used to replace the RHS function evaluation in the implementation variants considered.

- *Heat2D* is the 2D heat equation and describes the distribution of heat in a given region over time. Its ODE system is derived from a PDE (partial differential equation) by a second order discretization on a $N \times N$ grid and has the dimension $n = N^2$. Heat2D is memory-bound due to the growth of its access distance $d(\mathbf{f}) = N$ with increasing n. In the worst case, loading the required elements of $\mathbf{y}$ takes three single cache loads ($i - N$, $i - 1$ to $i + 1$, $i + N$).
- *InverterChain* is the electric circuit model of an inverter chain [1] and describes a traversing signal through a chain of N concatenated inverters. Its ODE system has dimension $n = N$ and access distance $d(\mathbf{f}) = 1$. InverterChain is compute-bound due to two expensive division operations.
- *Medakzo* is the medical Akzo Nobel problem [14] and describes the penetration of radio-labeled antibodies into a tissue that has been infected by a tumor. Its ODE system is derived from two 1D PDEs by the method of lines and has the dimension $n = 2N$. Its access distance is $d(\mathbf{f}) = N$. The number of cycles required to evaluate the ODE components differs between components derived from the first (memory-bound) and second PDE (compute-bound). The implementation used first stores all N components of the first PDE, then all N components of the second PDE, because this storage scheme is easier to handle for the *kerncraft* tool (see also discussion of *String-Row* and *String-Mix* in Sections 6.6 and 6.7).
- *Wave1D* is the linear convective 1D wave equation [4] and describes the propagation of disturbances at a fixed speed in one direction. Its ODE system is discretized on a uniform grid and the spatial derivative is approximated by a 9 point centered difference scheme of 8th order. The ODE system has the dimension $n = N$ and access distance $d(\mathbf{f}) = 4$. Wave1D is memory-bound.
- *String* is the 1D wave equation and describes the displacement of a vibrating elastic string. Its ODE system is derived from a 2D PDE using the method of characteristics [9] and has the dimension $n = 2N$. Implementation *String-Row* first stores all N components that correspond to displacements of the string, then all N components that correspond to velocities of the string. *String-Row* has access distance $d(\mathbf{f}) = N$. Computing the displacements is memory-bound, while the velocities are compute-bound. Implementation *String-Mix* reduces the access distance to $d(\mathbf{f}) = 3$ by an interleaved storage of displacement and velocity components.

5.2 Testbed

We present experimental results conducted with the different IVPs described above on a Haswell cluster described in depth in Tab. 1. Its nodes are equipped with Intel Xeon E5-2630 v3 2.3 GHz Haswell-EP processors and possess a shared 20 MB L3 cache. A 32 kB L1 data cache as well as a 256 kB L2 cache are available to each of the eight cores on a single node. For this work we had 8 cluster nodes at our disposal.

Table 1: Characteristics of the Haswell cluster.

Microarchitecture	Haswell EP
CPU	Intel Xeon E5-2630 v3
Clock (GHz)	2.3 GHz
Cores (Threads)	8 (16)
L1 cache	32 kB (data)
L2 cache	256 kB
L3 cache	20 MB (shared)
CL size	64 B
Instruction throughput per cycle	
LOAD/STORE	2 / 1
ADD/MUL/FMA	1 / 2 / 2
RAM per node	32 GB
Measured bandwidth (Load only)	51 GB/s
Connection	InfiniBand (56Gbit/s)
Max hops between nodes	2

All codes used in our experiments were compiled with the Intel C/C++ compiler and compiler flags `-O3`, `-fno-alias` and `-xHost`. We ran experiments for a broad range of system dimensions and used *KMP_AFFINITY=granularity=fine,compact* for thread binding. As corrector method we used the 4-stage ($s = 4$) method *Radau II A(7)* and applied $m = 6$ corrector steps per time step. This leads to a total number of 19 OpenMP barriers and 25 communication phases for all implementations considered.

6 EXPERIMENTAL RESULTS

6.1 MPI benchmarks

In this chapter we present the results of the MPI benchmarks we ran on the Haswell cluster.

First, we benchmarked MPI_Allgather, which is required for dense ODE systems, using vector sizes with up to about $3.5 \cdot 10^7$ doubles, which equals approximately 270 MB. As we can see in Fig. 1, the curve follows an almost straight line for large enough vector sizes. Thus, it can be expected that runtimes for even larger vectors can be extrapolated with high enough accuracy.

Second, we created a benchmark that can measure the runtime of the neighborhood communication that is used for ODE systems with limited access distance and which is implemented by MPI_Isend/MPI_Irecv. The results are shown in Fig. 2. Apart from a small number of spikes, that could be filtered out, the curve can be assumed to be linear if the message size is large enough.

6.2 Heat2D

The first problem we address is *Heat2D*, the two dimensional heat equation. We considered grid sizes from $N = 4800$ to 6000, which corresponds to system sizes n of up to 36 million components. Fig. 3 shows the runtimes measured using this IVP and implementations that use MPI_Allgather on 8 nodes with 8 threads each. The runtimes measured are in the range of approximately 1 s to 1.4 s for the smallest system size and 1.6 s to 2 s for the largest system size and increase proportionally to the size of the system. We see that

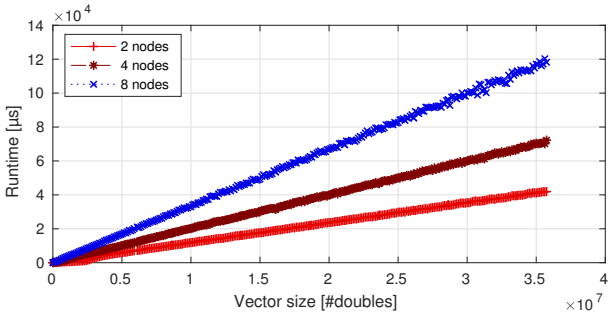

Figure 1: Results of the MPI_Allgather benchmark.

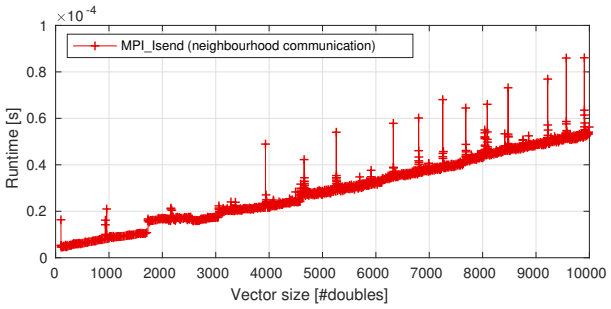

Figure 2: Results of the MPI_Isend neighborhood communication benchmark.

implementation variant *ilj* is performing significantly worse than the other implementation variants and that implementation variant *jil* achieves the best runtimes across all system sizes. Furthermore, we notice that the order of the implementation variants does not change for the system sizes considered. Fig. 4 shows the corresponding runtime predictions where we can see that the qualitative behavior of the curves is similar. Our approach correctly predicts that implementation variant *jil* is performing best and that implementation variants *ilj* and *lij* perform worst. However, we must note that the predicted runtimes are always lower than those actually observed. We show the percentage deviations between measured and predicted values in Fig. 5, which are between 7 and 10% for almost all implementation variants. As in our previous work, the prediction for implementation variant *ilj* is of lower quality (more than 20% deviation), since the special loop order of this implementation is problematic for the *kerncraft* tool in its current version due to the small number of iterations of the two outer loops.

Next, we focus on the implementations that use neighborhood communication. We recognize that the runtimes (Fig. 6) again scale with the system size, but at the same time we notice that they are considerably lower due to the optimized communication. The predictions we make show that the qualitative behavior is again similar and that the predicted runtimes (Fig. 7) are comparable to the measured ones. The deviations shown in Fig. 8 of up to 20% for all implementations except *ilj* suggest that our predictions are noticeably worse than for the MPI_Allgather variants. However, the high percentages are the result of the lower overall runtimes. But even though the predictions are too pessimistic, the qualitative

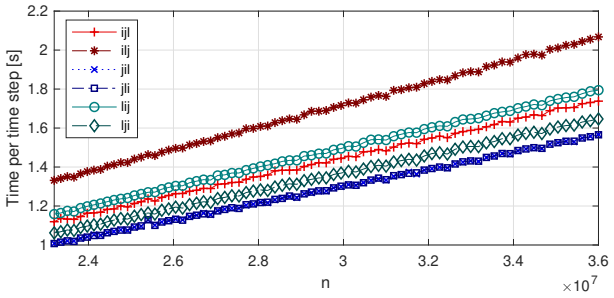

Figure 3: Measured runtimes of IVP Heat2D using multi-broadcast communication (8 nodes - 8 threads each).

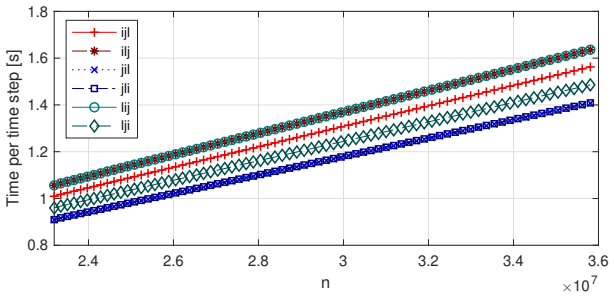

Figure 4: Runtime predictions for IVP Heat2D using multi-broadcast communication (8 nodes - 8 threads each).

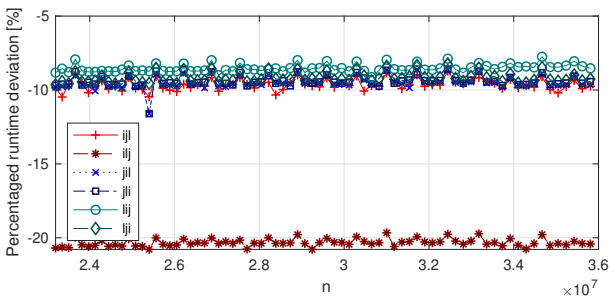

Figure 5: Runtime deviations for IVP Heat2D using multi-broadcast communication (8 nodes - 8 threads each).

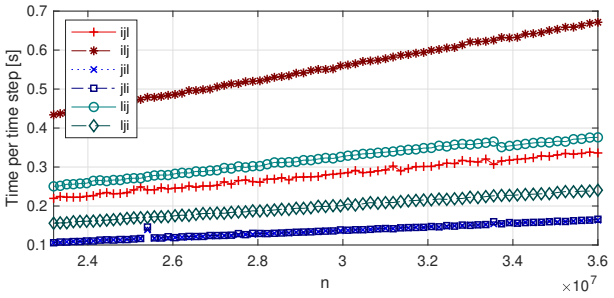

Figure 6: Measured runtimes of IVP Heat2D using neighborhood communication (8 nodes - 8 threads each).

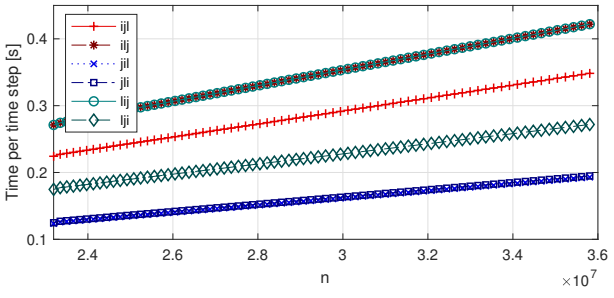

Figure 7: Runtime predictions for IVP Heat2D using neighborhood communication (8 nodes - 8 threads each).

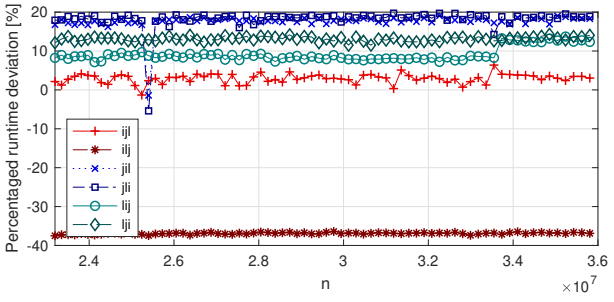

Figure 8: Runtime deviations for IVP Heat2D using neighborhood communication (8 nodes - 8 threads each).

behavior, i.e., the ranking of the implementation variants, allows to identify the best and worst variant again.

An important question is how well our individual estimates for the application and communication shares are performing. To investigate this, we compare the runtime shares of the predictions with those of a measurement made with the *Intel Trace Analyzer and Collector* tool. We choose *ijl* as representative implementation variant and $N = 5984$ as system size. Our prediction says that we expect 0.30 s application runtime and 1.27 s of communication per step if we use multi-broadcast operations. The times determined by the trace analyzer are 0.35 s application runtime and 1.40 s for MPI operations. This means we have a deviation of −15.6% for the application and −9.29% for the communication, which puts them both in the same order of magnitude, see Fig. 12. If we consider neighborhood communication we predict 0.05 s for communication. A measurement with the trace analyzer yielded 0.33 s for the application and 0.02 s of communication, see Fig. 13. The percentage deviation of the communication is over 200%, but has only little impact on the quality of the prediction due to the low share of the total runtime.

Fig. 9 shows the measured runtimes for the IVP *Heat2D* using now 4 nodes with neighborhood communication. The measured runtimes are about twice as long, which shows that the implementations are highly scalable. However, we have to acknowledge that the deviations, shown in Fig. 11 have decreased considerably with the changed number of processes. While we have observed deviations of approximately 0 - 20% for 8 processes, we see deviations of -10 to 10% for 4 processes. This raises the question whether an

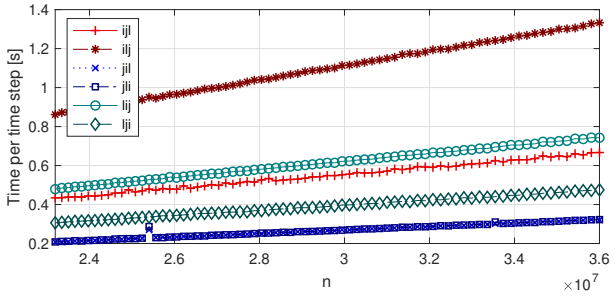

Figure 9: Measured runtimes of IVP Heat2D using neighborhood communication (4 nodes - 8 threads each).

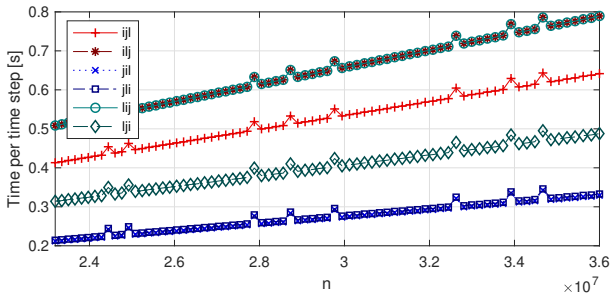

Figure 10: Runtime predictions for IVP Heat2D using neighborhood communication (4 nodes - 8 threads each).

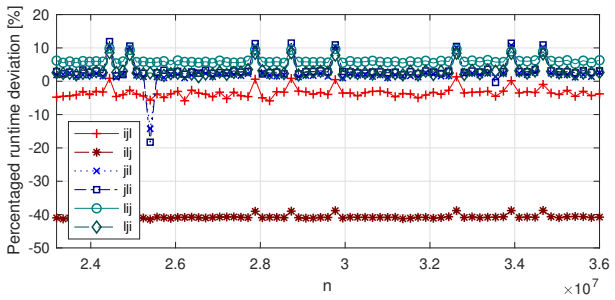

Figure 11: Runtime deviations for IVP Heat2D using neighborhood communication (4 nodes - 8 threads each).

additional number of processors leads to an increased prediction error. We investigated the data for 1 and 2 processes and found that there are no noticeable changes in the prediction error between 1, 2 and 4 processes. Unfortunately, we could not further investigate the behavior of the prediction error depending on the number of processes, since at the time of this work only 8 nodes were available, but plan on doing that in the future.

To summarize the results for *Heat2D*, using the approach proposed it was possible to create reasonable runtime predictions of the different implementation variants and to predict the impact of different communication patterns for this IVP. In particular, poorly performing implementation variants could be identified successfully and, thus, could be separated from good implementation variants, which can help in AT approaches as a pre-selection step.

Table 2: Measured runtimes and runtime predictions for IVP *InverterChain*, n=35,808,256, 8 nodes, 8 threads.

Variant	Prediction [s]	Measured [s]	Deviation [%]
Multi-broadcast communication			
ijl	1.60	1.66	-3.6
ilj	1.68	2.00	-1.6
jil	1.45	1.49	-2.7
jli	1.45	1.49	-2.7
lij	1.68	1.72	-2.3
lji	1.53	1.57	-2.5
Neighborhood communication			
ijl	0.387	0.360	7.5
ilj	0.461	0.695	-33.6
jil	0.232	0.192	20.8
jli	0.234	0.189	23.8
lij	0.461	0.403	14.4
lji	0.311	0.267	16.5

6.3 InverterChain

Similar to *Heat2D*, for all IVPs considered the runtime of a time step increases linearly with the system size. For this reason, we will only consider selected representative system sizes in the following subsections. In this subsection we study the problem *InverterChain* with a system size of $n = 35{,}808{,}256$, which we will use for all upcoming experiments, if not stated otherwise. As in the previous experiments, all following measurements were performed on 8 nodes with 8 threads each. Table 2 compares the predicted rutimes with the runtimes measured and shows the percentage deviations of the implementation variants using multi-broadcast operations and neighborhood communication, respectively. The predictions are very close to the measured runtimes and our approach was again able to distinguish well performing from poorly performing implementation variants. As before, we can measure and predict the impact of the communication used.

6.4 Medakzo

The medical Akzo Nobel problem differs from the IVPs considered so far in that no neighborhood communication is possible. Each process p_i requires a component from its immediate neighbors and all components of process $p_{i+\frac{p}{2}}$ if $i < N$ or $p_{i-\frac{p}{2}}$ if $i > N$. This problem can be considered sparse, but the storage order considered has no limited access distance, which is why we use the communication *sparse*. The communication pattern of *Medakzo* is nevertheless similar to neighborhood communication. Therefore, we estimate the communication costs with the neighborhood benchmark, assuming a message size of n_p. In Tab. 3 we can see that this assessment leads to reasonable runtime predictions. With regard to a possible autotuning, we can state that with the exception of the implementation *ilj*, the order of the variants is correct both for the multi-broadcast implementations and for the sparse implementations.

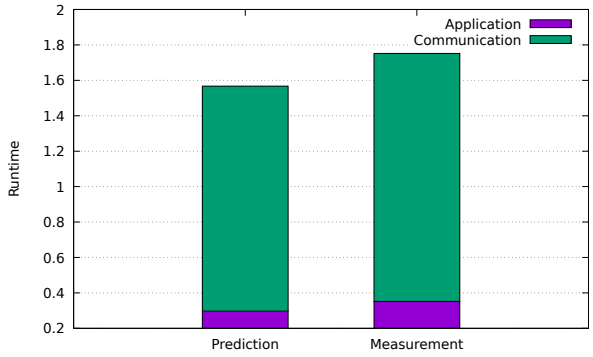

Figure 12: Detdetailed comparison of prediction and measurement for Heat2D, N=5984, implementation variant ijl, 8 nodes, 8 threads, multi-broadcast communication.

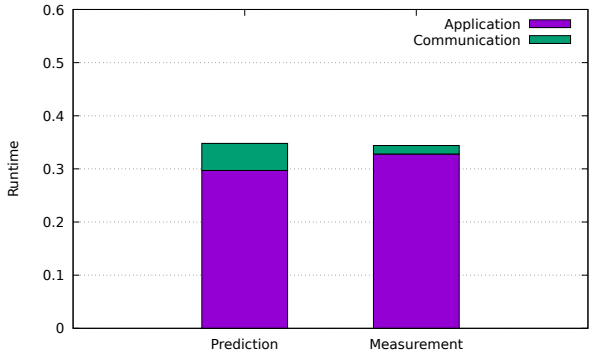

Figure 13: Detailed comparison of prediction and measurement for Heat2D, N=5984, implementation variant ijl, 8 nodes, 8 threads, neighborhood communication.

Table 3: Measured runtimes and runtime predictions for IVP *Medakzo*, n=35,808,256, 8 nodes, 8 threads.

Variant	Prediction [s]	Measured [s]	Deviation [%]
Multi-broadcast communication			
ijl	1.58	1.67	-5.4
ilj	1.65	1.99	-17.1
jil	1.43	1.49	-4.0
jli	1.43	1.49	-4.0
lij	1.66	1.72	-3.5
lji	1.51	1.58	-4.4
Sparse communication			
ijl	0.855	0.967	-11.5
ilj	0.928	0.875	6.1
jil	0.700	0.797	-12.2
jli	0.701	0.813	-13.8
lij	0.928	1.020	-9.0
lji	0.778	0.892	-12.8

Table 4: Measured runtimes and runtime predictions for IVP *Wave1D*, n=35,808,256, 8 nodes, 8 threads.

Variant	Prediction [s]	Measured [s]	Deviation [%]
Multi-broadcast communication			
ijl	1.56	1.64	-4.9
ilj	1.64	1.96	-16.3
jil	1.41	1.47	-4.1
jli	1.41	1.46	-3.4
lij	1.64	1.70	-3.5
lji	1.49	1.55	-3.9
Neighborhood communication			
ijl	0.348	0.336	3.6
ilj	0.421	0.669	-37.1
jil	0.192	0.163	17.8
jli	0.194	0.165	17.6
lij	0.421	0.375	12.3
lji	0.271	0.241	12.4

6.5 Wave1D

Characteristic for this problem, which is derived from a one dimensional PDE, is the discretization using central differences of order 8, which leads to a stencil with an access distance of 4. Looking at Tab. 4, we can observe that the runtimes do not differ significantly from those of the IVP *InverterChain*. The communication costs are determined by MPI_Allgather in the case of multi-broadcast implementations and by the MPI latency in the case of neighborhood communication. Also for this problem, our approach was able to identify both the best and the worst implementation variants.

6.6 String-Row

The *String* problem describes the displacement of a vibrating elastic string via a one dimensional wave equation. Its discretization yields a system of ODEs with two different components (displacements and velocities) that have to be stored in a one dimensional array. The *String-Row* implementation first stores all displacements and then all velocities. The runtimes of the multi-broadcast implementation variants are dominated by the communication costs, so the predictions depend largely on the MPI_Allgather benchmark. Since this benchmark gives good predictions, we can only detect minor deviations from the measured runtimes, see Tab. 5. The *String-Row* implementation is not able to use neighborhood communication so we use the *sparse* communication. Since the communication pattern is similar to neighborhood communication we try to estimate the runtimes using the neighborhood benchmark. Also in Tab. 5 we can see that the deviations for the sparse communication are a bit higher than previously observed.

6.7 String-Mix

String-Mix is another implementation of the forementioned 1D wave equation. In contrast to *String-Row*, this implementation stores

Table 5: Measured runtimes and runtime predictions for IVP *String-Row*, n=35,808,256, 8 nodes, 8 threads.

Variant	Prediction [s]	Measured [s]	Deviation [%]
Multi-broadcast communication			
ijl	1.56	1.66	-6.0
ilj	1.64	1.99	-17.6
jil	1.41	1.49	-5.4
jli	1.41	1.49	-5.4
lij	1.64	1.72	-4.7
lji	1.49	1.57	-5.1
Sparse communication			
ijl	0.731	0.959	-23.8
ilj	0.804	1.290	-37.7
jil	0.576	7.98	-27.8
jli	0.578	0.797	-27.5
lij	0.804	1.010	-20.4
lji	0.655	0.873	-25.0

Table 6: Measured runtimes and runtime predictions for IVP *String-Mix*, n=35,808,256, 8 nodes, 8 threads.

Variant	Prediction [s]	Measured [s]	Deviation [%]
Multi-broadcast communication			
ijl	1.63	1.62	0.6
ilj	1.70	1.95	-14.7
jil	1.47	1.45	1.4
jli	1.47	1.45	1.4
lij	1.70	1.68	1.2
lji	1.55	1.53	1.3
Neighborhood communication			
ijl	0.446	0.530	-15.8
ilj	0.519	1.130	-54.0
jil	0.270	0.290	-6.9
jli	0.272	0.287	-5.2
lij	0.489	0.572	-14.5
lji	0.369	0.423	-12.8

velocity and displacement components in alternation. Thus, the corresponding *RHS* kernel contains a loop with a stride of two, which makes this problem stand out. The behavior of the implementations using MPI_Allgather is similar to *String-Row*. In the variants with neighborhood communication, however, the runtime of the application has a larger share of the total runtime. In Tab. 6 we can see that the predictions generally deviate less than 15% from the measured runtimes. Consequently, we can conclude that *kerncraft* was able to handle the loop with a stride of two reasonably well.

Further, by comparing the predictions obtained for the sparse communication implementations of *String-Row* and *String-Mix*, it can be observed that our methodology correctly predicts *String-Mix* as the more efficient implementation of the *RHS* kernel. In the context of an AT procedure, such an observation could be used to pre-select the most efficient *RHS* kernel and, thus, to narrow down the search by one dimension.

6.8 Assessment of the proposed performance prediction methodology as a pre-selection tool in an AT approach

If *reliable*, a performance prediction methodology can be an excellent instrument inside an AT procedure to support the variant selection process. Implementation variants could be ranked by their performance and *slow* implementation variants filtered out during the offline phase based on the predictions obtained without actually executing them first. In contrast, an unreliable ranking might, have the opposite effect and lead to the dismissal of *performant* implementation variants or the time-consuming execution of *poorly performing* implementation variants during the online phase.

The previous experimental results (Tab. 2 to Tab. 6) indicate that our prediction approach is capable of pre-selecting suitable implementation variants for the system size studied. Nevertheless, it was found that our approach is sensitive to errors in the node-level prediction or the communication benchmarks. The runtime predictions

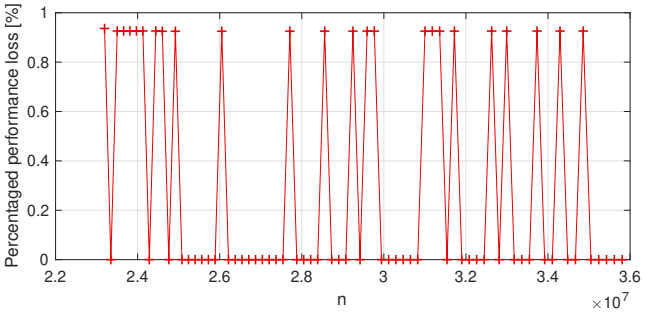

Figure 14: Percentaged performance loss comparing the performance of the predicted best variant versus the experimentally evaluated best variant for IVP *Heat2D*, 8 nodes, 8 threads, neighborhood communication.

differ from the actual runtimes by varying degrees, which means that we do not always identify the best implementation variant, but might sustain a performance loss. We study this phenomenon using the data from the heat equation for grid sizes from $N = 4800$ to 6000, which correspond to system sizes n of up to 36 million components. Fig. 14 depicts the percentaged performance loss sustained by executing the predicted best implementation variant instead of the experimentally evaluated best implementation variant.

Ideally, the performance loss is 0, i.e., our methodology selected the proper variant. It can be observed, that the sustained performance losses for IVP *Heat2D* are marginal (about 0.95% maximum). The runtimes and predictions of the two best implementation variants for *Heat2D* are fairly close (compare Fig. 6 to 7). Hence, minor measurement inaccuracies in the runtimes measured can already lead to the selection of another than the best implementation variant, e.g., the second or third best variant. These performances losses are, however, insignificant since the selected implementation variants are practically equally performant as the best one.

7 CONCLUSION AND FUTURE WORK

In this paper, we have considered performance prediction for scientific applications on cluster systems where MPI is used to communicate between the nodes and OpenMP is used for synchronization within the nodes. To achieve this goal, the methodology developed in [21], which considered node-level predictions on shared-memory systems, was extended to support cluster systems consisting of multiple nodes. This extended performance prediction methodology has been applied successfully to a representative class of explicit ODE methods. Moreover, we have demonstrated that our methodology is capable of deriving a reliable performance ranking of implementation variants for this kind of methods on multi-core cluster systems.

In particular, we have combined our ECM model based node-level prediction approach with cost estimates of the intra-node and inter-node communication costs of the implementation variants and defined a model which predicts the time required to execute a single time step for a particular combination of implementation variant, ODE system and target system. Node-level predictions were determined for different implementation variants of PIRK methods for different ODE systems. Cost estimation models for the communication operations required were deduced from benchmark results. Using the predicted runtimes, we have been able to establish a performance ranking of the implementation variants for each particular combination of target platform and ODE system. Finally, we have validated our predictions by comparing our ranking with actual runtimes measured on the target platform.

Our future work includes expanding our methodology to bigger HPC systems with more nodes and heterogeneous multi-core cluster systems. Hence, we intend to validate our predictions on additional target platforms (Intel IvyBridge, Intel Skylake, AMD Zen) and on heterogeneous systems containing multiple of these different platforms. Further, we plan to study in detail the usability of our prediction methodology for optimizing the load distribution in heterogeneous multi-core cluster systems. In particular, we want to investigate whether our methodology can be exploited for finding the *optimal* workload distribution on a heterogeneous multi-core cluster system during the offline phase of an AT procedure. Further, we intend to apply our methodology to more complex implementation variants using loop tiling and pipeline-like loop structures with stepsize control and plan to study the accuracy of our predictions for more complex ODE systems.

ACKNOWLEDGMENTS

This work is supported by the German Ministry of Science and Education (BMBF) under project number 01IH16012A.

REFERENCES

[1] A. Bartel, M. Günther, R. Pulch, and P. Rentrop. 2002. Numerical Techniques for Different Time Scales in Electric Circuit Simulation. In *High Performance Scientific and Engineering Computing: Proc. 3rd Int. FORTWIHR Conf. HPSEC.* Springer, Berlin, Heidelberg, 343–360.

[2] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. 1997. Optimizing Matrix Multiply using PHiPAC: A Portable High-performance, ANSI C Coding Methodology. In *Proc. 11th Int. Conf. Supercomputing (ICS '97).* ACM, New York, NY, USA, 340–347.

[3] K. Burrage. 1995. *Parallel and Sequential Methods for Ordinary Differential Equations.* Clarendon Press, New York, NY, USA.

[4] M. Calvo, J.M. Franco, and L. Rández. 2004. A new minimum storage Runge–Kutta scheme for computational acoustics. *Journal Comput. Physics* 201 (2004), 1–12.

[5] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proc. of the 25th IEEE Int. Parallel & Distributed Processing Symp.* IEEE, Los Alamitos, USA, 676–687.

[6] S. Das, S. S. Mullick, and P.N. Suganthan. 2016. Recent advances in differential evolution – An updated survey. *Swarm and Evolutionary Computation* 27 (2016), 1–30.

[7] Intel Corporation G. Slavova. 2018. Introducing Intel® MPI Benchmarks. https://software.intel.com/en-us/articles/intel-mpi-benchmarks. (Feb. 2018).

[8] M. Gerndt, E. César, and S. Benkner (Eds.). 2015. *Automatic Tuning of HPC Applications – The Periscope Tuning Framework.* Shaker Verlag.

[9] R. Haberman. 1998. *Elementary Applied Partial Differential Equations: With Fourier Series and Boundary Value Problems* (3rd ed.). Prentice Hall.

[10] E. Hairer, S.P. Nørsett, and Gerhard Wanner. 2000. *Solving Ordinary Differential Equations I: Nonstiff Problems* (2nd rev. ed.). Springer, Berlin, Heidelberg.

[11] E. Hairer and G. Wanner. 2002. *Solving Ordinary Differential Equations II : Stiff and Differential-Algebraic Problems* (2nd rev. ed.). Springer, Berlin, Heidelberg.

[12] J. Hammer, G. Hager, J. Eitzinger, and G. Wellein. 2015. Automatic Loop Kernel Analysis and Performance Modeling with Kerncraft. In *Proc. 6th Int. Workshop Performance Modeling, Benchmarking, & Simulation High Performance Computing Systems (PMBS '15).* ACM, New York, NY, USA, Article 4, 11 pages.

[13] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker. 2015. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation.* Springer International Publishing, Cham, 129–148.

[14] F. Mazzia, C. Magherini, and J. Kierzenka. 2008. Test Set for Initial Value Problem Solvers, Release 2.4. https://archimede.dm.uniba.it/~testset/. (Feb. 2008).

[15] J. A. Nelder and R. Mead. 1965. A Simplex Method for Function Minimization. *Comput. J.* 7, 4 (1965), 308–313.

[16] S. P. Nørsett and H. H. Simonsen. 1989. Aspects of Parallel Runge–Kutta Methods. In *Numerical Methods for Ordinary Differential Equations (Lecture Notes Math.).* Springer, Berlin, Heidelberg, 103–117.

[17] D. Panda. 2018. OSU Micro-Benchmarks 5.4.4, Release 5.4.4. http://mvapich.cse.ohio-state.edu/benchmarks/. (Sep. 2018).

[18] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. 2013. 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 & Implementation (PLDI'13).* ACM, New York, NY, USA, 519–530.

[19] R. H. Reussner, P. Sanders, and J. L. Träff. 2002. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming* 10, 1 (2002), 55–65.

[20] B. A. Schmitt. 2014. Peer Methods with Improved Embedded Sensitivities for Parameter-dependent ODEs. *J. Comput. Appl. Math.* 256 (Jan. 2014), 242–253.

[21] J. Seiferth, C. Alappat, M. Korch, and T. Rauber. 2018. Applicability of the ECM Performance Model to Explicit ODE Methods on Current Multi-core Processors. In *High Performance Computing.* Springer, Berlin, Heidelberg, 163–183.

[22] H. Stengel, J. Treibig, G. Hager, and G. Wellein. 2015. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In *Proc. 29th ACM Int. Conf. Supercomputing (ICS '15).* ACM, New York, NY, USA, 207–216.

[23] N. R. Tallent and J. M. Mellor-Crummey. 2009. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proc. 14th ACM SIGPLAN Symp. Principles & Practice Parallel Programming (PPoPP '09).* ACM, New York, NY, USA, 229–240.

[24] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proc. of the Twenty-third Annual ACM Symp. on Parallelism in Algorithms & Architectures (SPAA '11).* ACM, New York, NY, USA, 117–128.

[25] M. M. Tikir and J. K. Hollingsworth. 2004. Using Hardware Counters to Automatically Improve Memory Performance. In *Proc. ACM/IEEE Conf. Supercomputing (SC '04).* IEEE Computer Society, Washington, DC, USA, 46–.

[26] A. Tiwari and J. K. Hollingsworth. 2011. Online Adaptive Code Generation and Tuning. In *Proc. IEEE Int. Parallel & Distributed Processing Symp.* IEEE, 879–892.

[27] J. Treibig and G. Hager. 2010. Introducing a Performance Model for Bandwidth-Limited Loop Kernels. In *Parallel Processing and Applied Mathematics: 8th Int. Conf., PPAM 2009. Revised Selected Papers, Part I.* Springer, Berlin, Heidelberg, 615–624.

[28] P. J. van der Houwen and B. P. Sommeijer. 1990. Parallel iteration of high-order Runge–Kutta Methods with stepsize control. *J. Comput. Appl. Math.* 29 (1990), 111–127.

[29] R. C. Whaley and J. J. Dongarra. 1999. *Automatically Tuned Linear Algebra Software.* Technical Report. University of Tennessee.

[30] S. Williams, A. Waterman, and D. Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76.