

# Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review

Suejb Memeti<sup>1</sup>  · Sabri Pllana<sup>1</sup> · Alécio Binotto<sup>2</sup> · Joanna Kołodziej<sup>3</sup> · Ivona Brandic<sup>4</sup>

Received: 13 December 2017 / Accepted: 19 April 2018 / Published online: 26 April 2018  
© The Author(s) 2018

**Abstract** While modern parallel computing systems offer high performance, utilizing these powerful computing resources to the highest possible extent demands advanced knowledge of various hardware architectures and parallel programming models. Furthermore, optimized software execution on parallel computing systems demands consideration of many parameters at compile-time and run-time. Determining the optimal set of parameters in a given execution context is a complex task, and therefore to address this issue researchers have proposed different approaches that use heuristic search or machine learning. In this paper, we undertake a systematic literature review to aggregate, analyze and classify the existing software optimization methods for parallel computing systems. We review approaches that use machine learning or meta-heuristics for software optimization at compile-time and run-time. Additionally,

---

✉ Suejb Memeti  
suejb.memeti@lnu.se

Sabri Pllana  
sabri.pllana@lnu.se

Alécio Binotto  
abinotto@br.ibm.com

Joanna Kołodziej  
jokolodziej@pk.edu.pl

Ivona Brandic  
ivona.brandic@tuwien.ac.at

<sup>1</sup> Department of Computer Science, Linnaeus University, 351 95 Växjö, Sweden

<sup>2</sup> IBM Research, São Paulo, Brazil

<sup>3</sup> Cracow University of Technology, 31 155 Cracow, Poland

<sup>4</sup> Electronic Commerce Group, Institute of Software Technology and Interactive Systems, Vienna University of Technology, 1040 Vienna, Austria

we discuss challenges and future research directions. The results of this study may help to better understand the state-of-the-art techniques that use machine learning and meta-heuristics to deal with the complexity of software optimization for parallel computing systems. Furthermore, it may aid in understanding the limitations of existing approaches and identification of areas for improvement.

**Keywords** Parallel computing · Machine learning · Meta-heuristics · Software optimization

**Mathematics Subject Classification** 90C27 Combinatorial optimization · 68T20 Problem solving (heuristics, search strategies, etc.) · 68T05 Learning and adaptive systems [See also 68Q32, 91E40] · 65Y05 Parallel computation

## 1 Introduction

Traditionally, parallel computing [73] systems have been used for scientific and technical computing. Usually scientific and engineering computational problems are complex and resource intensive. To efficiently solve these problems, utilization of parallel computing systems that may comprise multiple processing units is needed. The emergence of multi-core and many-core processors in the last decade led to the pervasiveness of parallel computing systems from embedded systems, personal computers, to data centers and supercomputers. While in the past parallel computing was a focus of only a small group of scientists and engineers at supercomputing centers, nowadays programmers of virtually all systems are exposed to parallel processors that comprise multiple or many cores [49].

The modern parallel computing systems offer high performance capabilities. In recent years, the computational capabilities of supercomputing centers have been increasing very fast. For example, the average performance of the top 10 supercomputers in 2010 was 0.84 PFlops/s, in 2014 the average performance climbed to 11.16 PFlops/s, and in 2016 the average performance capability is 20.63 PFlops/s [94]. With such exciting performance gain, a serious issue of the power consumption of these supercomputing centers arises. For example, according to the TOP 500 list [94], in the years 2010 to 2016, the average power consumption of the top 10 supercomputers has increased from 2.98 to 8.88 MW, that is about 198% increase.

Utilizing these resources to gain the highest extent of performance while keeping low level of energy consumption demands significant knowledge of vastly different parallel computing architectures and programming models. Improving the resource utilization of parallel computing systems (including heterogeneous systems that comprise multiple non-identical processing elements) is important, yet difficult to achieve [50]. For example, for data-intensive applications the limited bandwidth of the PCIe interconnection forces developers to use the resources on the host only, which leads to the underutilization of the system. Similarly, in compute-intensive applications, while utilizing the accelerating device, the host CPUs remain idle, which leads to waste of energy and performance. Approaches that intelligently manage the resources of host CPUs and accelerating devices to address such inefficiencies seem promising [68].

To achieve higher performance, scalability and energy efficiency, engineers often combine Central Processing Units (CPUs), Graphical Processing Units (GPUs), or Field Programmable Gate Arrays (FPGAs). In such environments, system developers need to consider multiple execution contexts with different programming abstractions and run-time systems. There is a consensus that software development for parallel computing systems, especially heterogeneous systems, is significantly more complex than for traditional sequential systems. In addition to the programmability challenges, performance portability of programs to various platforms is essential and challenging for productive software development, due to the differences in architectural level of multi-core and many-core processors [9].

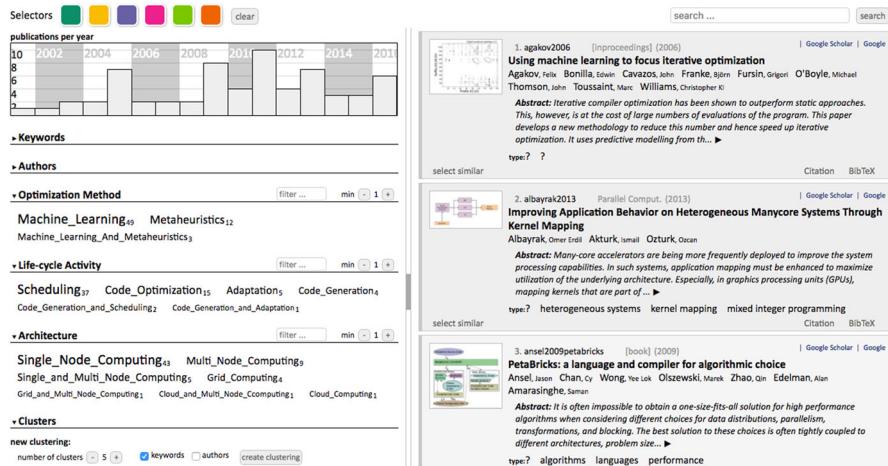
Software development and optimal execution on parallel computing systems expose programmers and tools to a large number of parameters [83] at software compile-time and at run-time. Examples of properties for a GPU-accelerated system include: CPU count, GPU count, CPU cores, CPU core architecture, CPU core speed, memory hierarchy levels, GPU architecture, GPU device memory, GPU SM count, CPU cache, CPU cache line, memory affinity, run-time system, etc. Finding the optimal set of parameters for a specific context is a non-trivial task, and therefore many methods for software optimization that use meta-heuristics and machine learning have been proposed. A systematic literature review may help to aggregate, analyze, and classify the proposed approaches and derive the major lessons learned.

In this paper, we conduct a systematic literature review of approaches for software optimization of parallel computing systems. We focus on approaches that use machine learning or meta-heuristics that have been published since the year 2000. We classify the selected review papers based on the software life-cycle activities (compile-time or run-time), target computing systems, optimization methods, and period of publication. Furthermore, we discuss existing challenges and future research directions. The aims of this systematic literature review are to:

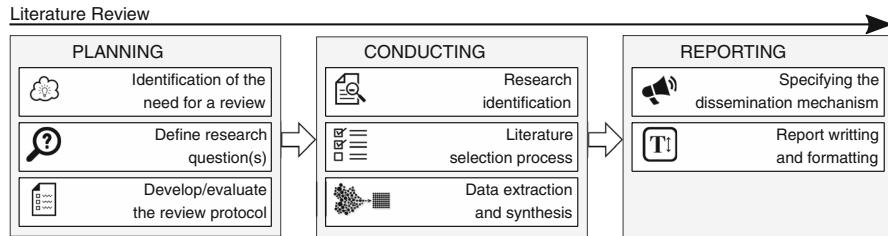
- systematically study the state-of-the-art software optimization methods for parallel computing systems that use machine learning or meta-heuristics;
- classify the existing studies based on the software life-cycle activities (compile-time, and run-time), target computing systems, optimization methods, and period of publication;
- discuss existing challenges and future research directions.

Figure 1 depicts our solution for browsing the results of literature review that we have developed using SurVis [8] literature visualization tool. The browser is available on-line at [www.smemeti.com/sl/](http://www.smemeti.com/sl/) and enables to filter the review results based on the optimization methods, software life-cycle activity, parallel computing architecture, keywords, and authors. A time-line visualizes the number of publications per year. Publications that match filtering criteria are listed on the right-hand side; the browser displays for each publication the title, authors, abstract, optimization method, life-cycle activity, target system architecture, keywords, and a representative figure. The on-line literature browser is easy to extend with future publications that fit the scope of this review.

The rest of the paper is organized as follows. In Sect. 2 we describe the research methodology. In Sect. 3, we give an overview of the parallel computing systems, soft-



**Fig. 1** Our interactive browser of the results of literature review. Results can be filtered by software optimization method, software life-cycle activity, parallel system architecture, keyword, and author name. Results are visualized in the form of time-line that indicates the number of publications per year. The right-hand compartment lists the publications that match the search criteria. The browser is available on-line at [www.smemeti.com/sl/](http://www.smemeti.com/sl/)



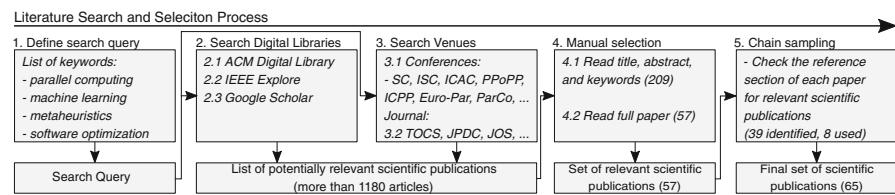
**Fig. 2** Research methodology

ware optimization techniques, and the software optimization at different life-cycle activities. For each of the software life-cycle activities, including *Compile-Time* activities (Sect. 4), and *Run-Time* activities (Sect. 5), we discuss the characteristics of state-of-the-art research, and discuss limitations and future research directions. Finally, in Sect. 6 we conclude our paper.

## 2 Research methodology

We perform a literature review based on guidelines by Kitchenham and Charters [53]. In summary, these guidelines include three stages: Planning, Conducting and Reporting (see Fig. 2).

During the planning stage the following activities are performed: (1) identifying the need for a literature review, (2) defining the research questions of the literature review, and (3) developing/evaluating the protocol for performing the literature review. The activities associated with conducting the literature review include: (1) identifying



**Fig. 3** The process of searching and selecting the relevant literature

the research, (2) literature selection, (3) data extraction and synthesis. The reporting stage includes writing the results of the review and formatting the document. In what follows, we describe in more details the research method and the major activities performed during this study.

## 2.1 Research questions

We have defined the following research questions:

- **RQ1:** Which software optimization goals for parallel computing systems are achieved using meta-heuristics and machine learning?
- **RQ2:** Which are the common algorithms used to achieve such software optimization goals for parallel computing systems?
- **RQ3:** Which features are considered during software optimization of parallel computing systems?

## 2.2 Search and selection of literature

The literature search and selection process are depicted in Fig. 3. Based on the objectives of the study, we have selected an initial set of keywords (see activity 1) that is used to search for articles, such as: *parallel computing*, *machine learning*, *meta-heuristics* and *software optimization*. To improve the result of the search process, we consider synonyms for the keywords during the search. The search query is executed on digital electronic databases (such as, ACM Digital Library, IEEEExplore, and Google Scholar), conference venues (such as, SC, ISC, ICAC, PPoPP, ICDCS, CGO, ICPP, Euro-Par, and ParCo), and scientific journals (such as, TOCS, JPDC, JOS). The outcome of the search process is a list of potentially relevant scientific publications. Manual selection of these publications by reading the title, abstract, and keywords (activity 4.1) first, then the full paper (activity 4.2) is performed, which results in a filtered list of relevant scientific publications. Furthermore, a recursive procedure of searching for related articles is performed using the corresponding related articles section of each digital library (for example, the ACM Digital Library related papers function powered by IBM Watson, or the *Related articles* function of Google Scholar).

The initial automatic search on the ACM digital library (see Fig. 3, activity 2.1) returned a list of total 25970 entries (articles). We sorted the entries by relevance,

such that the most relevant articles will show up first. As expected, the most relevant articles were found in the first part of the list, and after hundreds of articles, the suggested entries were not relevant to our study. Therefore, we decided to consider only the first 1000 articles. Out of these articles, only 130 were selected for further study based on reading the title and abstract (activity 4.1), and after reading the full article (activity 4.2), 22 were selected as relevant articles. The IEEEXplore returned 40 potentially relevant articles (activity 2.2), 20 of them were selected for further study based on reading the title and abstract (activity 4.1), and 16 were selected as relevant after reading the full paper (activity 4.2). The Google Scholar returned 140 potentially relevant articles (activity 2.3), 31 of them were selected after reading the title and abstract (activity 4.1), and 11 were selected as relevant after reading the full paper (activity 4.2). Searching the conference venues (activity 3.1) and scientific journals (activity 3.2), we selected 28 articles based on reading the title and abstract (activity 4.1), and 8 of them were selected as relevant after reading the full paper (activity 4.2). So, out of more than 1180 articles returned from various sources (activity 2 and 3), 209 were selected manually based on reading the title and abstract (activity 4.1), out of which, after reading the full content (activity 4.2), 57 were selected as relevant to the scope of this paper.

Additionally, the chain sampling technique (also known as snowball sampling, see Fig. 3, activity 5) is used to search for related articles. 39 articles were identified using this technique by reading the title and abstract (activity 4.1), and 8 of them were selected as relevant after reading the full paper (activity 4.2). Chain sampling is a recursive technique that considers existing articles, usually found in the references section of the research publication under study [10]. In total, 65 publications are considered in this review.

## 2.3 The focus and scope of the literature review (selection process)

The scope of this literature review includes:

- publications that investigate the use of machine learning or meta-heuristics for software optimization of parallel computing systems;
- publications that contribute to compile-time activities (code optimization and code generation), and run-time activities (scheduling and adaptation) of software life-cycle;
- research published since the year 2000, because in literature, the year 2000 is considered as the starting point of the multi-core era. IBM Power 4 [25], the first industry dual-core processor, is introduced in 2001 [37].

While other optimization methods (such as, linear programming, dynamic programming, control theory), and other software optimization activities (such as, design-time software optimization) may be of interest, they are left out of scope to keep the systematic review focused.

**Table 1** An excerpt of data items collected for each of the selected publications

	Data item	Description
1	Date	Date of the data extraction
2	Bibliographic reference	Author, Year, Title, Research Center, Venue
3	Type of article	Journal article, conference paper, workshop paper, book section
4	Problem, objectives, solution	What is the problem; what are the objectives of the study; how the proposed solution works?
5	Optimization Technique	Which Machine Learning or Meta-heuristic algorithm is used?
6	Considered features	The list of considered features used for optimization
7	Life-cycle Activity	Code Optimization, Code Generation, Scheduling, Adaptation?
8	Target architecture	Single/Multi-node system, Grid Computing, Cloud Computing
9	Findings and conclusions	What are the findings and conclusions?
10	Relevance	Relevance of the study in relation to the topic under consideration

## 2.4 Data extraction

In accordance with the classification strategy (described in Sect. 3.3) and the defined research questions (described in Sect. 2.1), for each of the selected primary studies we have collected information that we consider important to be recorded in order to perform the literature review.

Table 1 shows an excerpt of the data items (used for quantitative and qualitative analysis) collected for each of the selected studies. Data items 1–3 are used for the quantitative analysis related to *RQ1*. Data item 4 is used to answer *RQ2*. Data collected for item 5 is used to answer *RQ3*, whereas data collected for item 6 is used to answer *RQ4*. Data item 7 is used to classify the selected scientific publications based on the software life-cycle activities (see Table 3), whereas data item 8 is used for the classification based on the target architecture (see Fig. 6).

## 3 Taxonomy and terminology

In this section, we provide an overview of the parallel computing systems and software optimization approaches with focus on machine learning and meta-heuristics. Thereafter, we present our approach for classifying the state-of-the-art optimization techniques for parallel computing.

### 3.1 Parallel computing systems

A parallel computing system comprises a set of interconnected processing elements and memory modules. Based on the system architecture, generally parallel computers

can be categorized into shared and distributed memory. *Shared memory* parallel computing systems communicate through a global shared memory, whereas in *distributed memory* systems every processing element has its own local memory and the communication is performed through message passing. While shared memory systems have shown limited scalability, distributed memory systems have demonstrated to be highly scalable. Most of the current parallel computing systems use shared memory within a node, and distributed memory between nodes [6].

According to Top500 [94] in the 90s the commonly used parallel computing systems were *symmetric multi-processing* (SMP) systems and *massive parallel processing* (MPP) systems. SMPs are shared memory systems where two or more identical processing units share other system resources (main memory, I/O devices) and are controlled by a single operating system. MPPs are distributed memory systems where a larger number of processing units (or separate computers) are housed in the same place. The disparate processing units share no system resources, they have their own operating system, and communicate through high-speed network. The main computing models within the distributed parallel computing systems include *cluster* [26, 89], *grid* [13, 32, 82, 86], and *cloud* computing [33, 59, 82].

Nowadays, the mainstream platforms for parallel computing, at their node level consist of multi-core and many-core processors. *Multi-core* processors may have multiple cores (two, four, eight, twelve, sixteen...) and are expected to have even more cores in the future. *Many-core* systems consist of larger number of cores. The individual cores of the many-core systems are specialized to efficiently perform operations such as, SIMD, SIMT, speculations, and out-of-order execution. These cores are more energy efficient because they usually run at lower frequency.

Systems that comprise multiple identical cores or processors are known as *homogeneous systems*, whereas *heterogeneous systems* comprise non-identical cores or processors. As of November 2017, the TOP500 list [94] contains several supercomputers that comprise multiple heterogeneous nodes. For example, a node of Tianhe-2 (2nd most powerful supercomputer) comprises Intel Ivy-Bridge multi-core CPUs and Intel Xeon Phi many-core accelerators; Piz Daint (3rd) consists of Intel Xeon E5 multi-core CPUs and NVIDIA Tesla P100 many-core GPUs [66, 96].

Programming parallel computing systems, especially heterogeneous ones, is significantly more complex than programming sequential processors [78]. Programmers are exposed to various parallel programming languages (often implemented as extensions of general-purpose programming languages such as C and C++), including, OpenMP [72], MPI [42], OpenCL [90], NVIDIA CUDA [70], OpenACC [100] or Intel TBB [97]. Additionally, the programmer is exposed to different architectures with different characteristics (such as the number of CPU/GPU devices, the number of cores, core speed, run-time system, memory and memory levels, cache size). Finding the optimal system configuration that results in the highest performance is challenging. In addition to the programmability challenge, heterogeneous parallel computing systems bring the portability challenge, which means that programs developed for a processor architecture (for instance, Intel Xeon Phi) may not function on another processor architecture (such as, GPU). Manual software porting and performance tuning for various architectures may be prohibitive.

Existing approaches, discussed in this study, propose several solutions that use machine learning or meta-heuristics during compile-time and run-time to alleviate the programmability and performance portability challenges of parallel computing systems.

### 3.2 Software optimization approaches

In computer science selecting the best solution considering different criteria from a set of various available alternatives is a frequent need. Based on what type of values the model variables can take, the optimization problems can be broadly classified in continuous and discrete. Continuous optimization problems are concerned with the case where the model variables can take any value permitted by some given constraints. Continuous optimization problems are easier to solve. Given a point  $x$ , using continuous optimization techniques one can infer information about neighboring points of  $x$  [39].

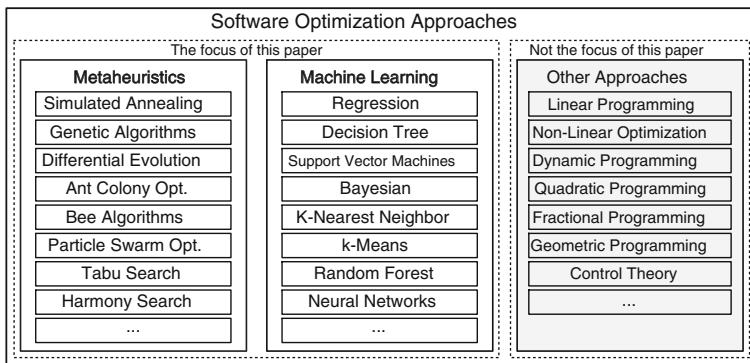
In contrast, in discrete optimization (also known as combinatorial optimization) methods the model variables belong to a discrete set (typically subset of integers) of values. Discrete optimization deals with problems where we have to choose an optimal solution from a finite number of possibilities. Discrete optimization problems are usually hard to solve and only enumeration of all possible solutions is guaranteed to give the correct result. However, enumerating across all available solutions in a large search space is prohibitively demanding.

Heuristic-guided approaches are designed to solve optimization problems more quickly by finding approximate solutions when other methods are too slow or fail to find any exact solution. These approaches select near-optimal solutions within a time frame (that is, they trade-off optimality for speed). While heuristics are designed to solve a particular problem (problem-dependent), meta-heuristics can be applied to a broad range of problems. They can be thought as higher-level heuristics that are designed to determine a near-optimal solution to an optimization problem, with limited computation capacity and knowledge about the problem.

In what follows, we first describe the meta-heuristics and list commonly used algorithms, and thereafter, we describe machine learning in the context of software optimization.

#### 3.2.1 Meta-heuristics

Meta-heuristics are high-level algorithms that are capable to determine a sufficiently satisfactory (near-optimal) solution to an optimization problem with limited domain knowledge and computation capacity. As meta-heuristics are problem-independent they can be used for a variety of problems. Meta-heuristics algorithms are often used for the management and efficient use of resources to increase productivity [79, 101]. In cases where the search space is large, exhaustive search, iterative methods, or simple heuristics are impractical, whereas meta-heuristics can often find good solutions with less computational effort. Meta-heuristics have shown to provide efficient solution to different problems, such as the minimum spanning tree (MST), traveling salesman



**Fig. 4** Classification of the software optimization approaches. While there exist many different optimization approaches, in this study we focus on meta-heuristics and machine learning

problem (TSP), shortest path trees, and matching problems. Selecting the most suitable heuristic for a specific problem is important to reach a near-optimal solution more quickly. However, this process requires consideration of various factors, such as the domain type, search space, computational time, and solution quality [12, 65].

In the context of software optimization, the commonly used meta-heuristics include Genetic Algorithms, Simulated Annealing, Ant Colony Optimization, Local Search, Tabu Search, and Particle Swarm Optimization (see Fig. 4).

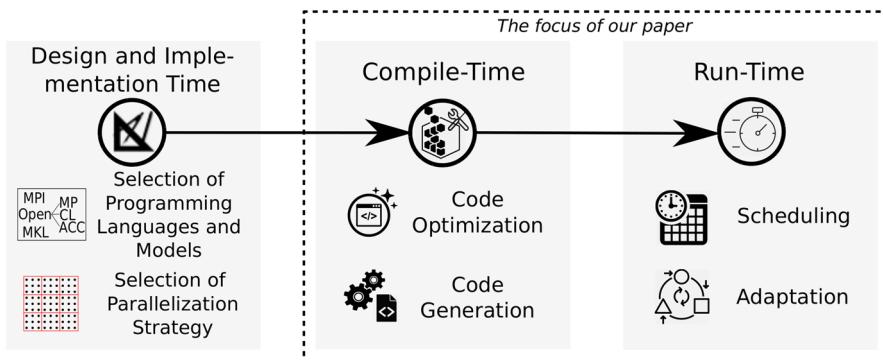
### 3.2.2 Machine learning

Machine Learning is a technique that allows computing systems to learn (that is, improve) from the experience (available data). Mitchell [67] defines Machine Learning as follows, “A computer program is said to learn from experience  $E$  with respect to some class of tasks  $T$  and performance measure  $P$ , if its performance at tasks in  $T$ , as measured by  $P$ , improves with experience  $E$ ”.

Machine learning programs operate by building a prediction model from a set of training data, which later on is used to make data-driven predictions, rather than following hard-coded static instructions. Some of the most popular machine learning algorithms (depicted in Fig. 4) include regression, decision tree, support vector machines, Bayesian inference, random forest, and artificial neural networks.

An important process while training a model is the *feature selection*, because the efficiency of models depends on the selected variables. It is critical to choose features that have significant impact on the prediction model. There are different feature selection techniques that can find features that contain the most useful information to distinguish between classes, for example mutual information score (MIS) [27], greedy feature selection [87], or information gain ratio [45].

Depending on the way the prediction model is trained, machine learning may be *supervised* or *unsupervised*. In supervised machine learning the prediction model learns from examples that are labeled, which means that the input and the output are known in the training data set. Supervised learning uses classification techniques



**Fig. 5** Software life-cycle activities. At design and implementation the selection of the programming languages, models, and the parallelization strategy occurs. We focus on software optimization that occurs during software compile-time (that includes code optimization and generation), and during run-time (that includes scheduling and adaptation)

to predict discrete responses (such as, determining whether an e-mail is genuine or spam, determining whether a tumor is malign or benign), and regression techniques to predict continuous responses (such as, changes in temperature, fluctuations in power demand). The most popular supervised learning algorithms for classification problems include Support Vector Machines, Naive Bayes, Nearest Neighbor, and Discriminant Analysis, whereas for regression problems algorithms such as Linear Regression, Decision Trees, and Neural Networks are used. Selecting the best algorithm depends on the size and type of input data set, the desired output (insight), and how those insights will be used.

The unsupervised machine learning models have no or very little knowledge of how the results should look like. Basically, correct results (that is labeled training data sets) are not used for model training, but the model aims at finding hidden patterns in data based on statistical properties (for instance, intra-cluster variance) of the training data sets. Unsupervised learning can be used for solving data clustering problems in various domains, for example, sequence analysis, market research, object recognition, social network analysis, and astronomical data analysis. Some commonly used algorithms for data clustering include K-Means, Hierarchical Clustering, Neural Networks, Hidden Markov Model, and Density-based Clustering.

### 3.3 Software optimization at different software life-cycle activities

Software optimization can happen during different activities of the software life-cycle. We categorize the software optimization activities by the time of their occurrence: *Design and Implementation-time*, *Compile-time*, *Run-time* (Fig. 5).

During the *design and implementation* activity, decisions such as selection of the programming language/model and selection of the parallelization strategy are considered.

The *compile-time* activities include decisions of selecting the optimal compiler optimization flags and source code transformations (such as loop unrolling, loop nest optimization, pipelining, and instruction scheduling) such that the executable program is optimized to achieve certain goals (performance or energy) on a given context.

The *run-time* activities include decisions of selecting the optimal data and task scheduling on parallel computing systems, as well as taking decisions (such as switching to another algorithm or changing the clock frequency) that help the system to adapt itself during the program execution and improve the overall performance and energy efficiency.

While software design and implementation activities are performed by the programmer, software activities at compile-time and run-time are completed by tools (such as compilers and run-time systems). Therefore, in this paper we focus on tool-supported software optimization approaches that use approximate techniques (machine learning and meta-heuristics) at compile-time and run-time.

For each of the software optimization life-cycle activities, including *Compile-Time* (Sect. 4) and *Run-Time* (Sect. 5), we will describe the context for software optimization goals, discuss the state-of-the-art research, and discuss limitations and future research directions.

### 3.4 Classification based on architecture, software optimization approach, and life-cycle activity

In this section we classify the considered scientific publications based on the architecture, software optimization approach, and life cycle activities.

To provide an overview of the current state of the art, we have grouped the scientific publications that use machine learning and meta-heuristics for software optimization of parallel computing systems in the following time periods: 2000–2005, 2006–2011, and 2012–2017. Each of the periods, correspond to the type of the processors that were used the most in the TOP list during that time. For example, even though the first multi-core processor was introduced in 2001 [37], most of the super computers in TOP500 list during years 2000–2005 comprised multiple single-core processors [94]. Further filtering and classification of the considered scientific publications, and visualization of the results in the form of a time-line can be performed using our on-line interactive tool (see Fig. 1).

*Architecture* Figure 6 shows a classification of the reviewed papers based on the target architecture, including multi-node, single-node, grid, and cloud parallel computing systems. The horizontal axis on the top delineates the common types of processors used during the corresponding time period. For instance, from 2000 to 2005 grids and clusters employed single or multiple sequential processors at node level, whereas during the period from 2006 to 2011 nodes employed multi-core processors. Accelerators combined with multi-core processors can be seen during time period 2012–2017. We may observe that most of the work is focused on optimization of resource utilization at the node level (single-node). Optimization of the resources of multi-node computing systems (including clusters) is addressed by several research studies continuously

	Sequential processors	Multi-core processors	Multi-core and accelerators
Cloud Comp.			[60, 61, 44]
Grid Comp.	[54]	[14, 76]	[43, 36]
Single-Node	[104, 69, 88, 102, 17, 87, 21, 103, 91]	[1, 38, 24, 34, 95, 7, 19, 5, 58, 98, 46, 47, 28, 77, 40, 15, 41, 80, 9, 23, 29, 35, 92]	[52, 55, 16, 51, 11, 30, 4, 57, 99, 31, 71, 62, 63, 64, 65, 48]
Multi-Node	[91, 22, 74, 75]	[18, 34, 93, 85, 35, 92]	[81, 60, 56, 36, 48, 84]
	2000–2005	2006–2011	2012–2017

**Fig. 6** Classification of state-of-the-art work based on the architecture (multi-node, single-node, grid, and cloud computing systems, as described in Sect. 3.1). Please note that a single paper may target more than one architecture (for instance, [36, 60])

**Table 2** Classification of state-of-the-art work based on the intelligent technique (machine learning or meta-heuristics) used during compile-time and/or run-time of software optimization

Machine learning	[17, 22, 54, 69, 87, 91, 102, 103]	[1, 5, 7, 9, 15, 18, 19, 23, 24, 28, 29, 34, 35, 40, 41, 46, 47, 58, 76, 77, 80, 95, 98]	[11, 16, 30, 31, 31, 36, 43, 44, 51, 52, 55, 57, 60, 61, 63– 65, 71, 81, 84, 99]
Meta-heuristics	[2, 21, 74, 75, 88, 104, 105]	[14, 38, 85, 92, 93]	[4, 43, 44, 56, 62, 63, 65]
	2000–2005	2006–2011	2012–2017

during the considered periods of time. The optimization of grid computing systems using machine learning and meta-heuristic approaches has received less attention, whereas optimization of cloud computing systems has received attention during the period 2012–2017.

*Software optimization approach* In Table 2 we classify the selected publications that use intelligent techniques (such as, machine learning and meta-heuristics) for software optimization at compile-time and run-time. We may observe that machine learning is used more often for software optimization during compile-time and run-time compared to meta-heuristics.

*Life-cycle activity* A classification of the reviewed papers based on the software life-cycle activities (including, code optimization, code generation, scheduling, and adaptation) is depicted in Table 3. We may observe that the scheduling life-cycle activity has received the most attention, especially during 2012–2017 period. The use of machine learning and meta-heuristics for code optimization during compile-time has been addressed by many researchers, especially during the period between 2006 and 2011. Similar trend can be observed for research studies that focus on using intelligent approaches to optimize code generation. Optimization of software through adaptation is addressed during the year of 2006–2011.

**Table 3** Classification of state-of-the-art work based on the software life-cycle activities (code optimization, code generation, scheduling, and adaptation)

Code Optimization	[17, 21, 69, 87, 88]	[1, 18, 34, 35, 38, 92, 93, 95]	[57, 99]
Code Generation		[5, 7, 19, 58, 77, 95]	[31, 81]
Scheduling	[2, 22, 54, 74, 75, 102–105]	[7, 9, 15, 23, 24, 40, 41, 76, 80, 85, 98]	[4, 11, 16, 30, 31, 36, 43, 44, 51, 52, 55, 56, 60–65, 71, 84]
Adaptation	[91] 2000–2005	[28, 29, 46, 47, 58] 2006–2011	2012–2017

Please note that a single paper may contribute to more than one software life-cycle activities (for instance, [7, 58])

## 4 Compile-time

Compiling [3] is the process of transforming source code from one form into another. Traditionally, compiler engineers exploited the underlying architecture by manually implementing several code transformation techniques. Furthermore, decisions that determine whether to apply a specific optimization or not were hard-coded manually. At each major revision or implementation of new instruction set architecture, the set of such hard-coded compiler heuristics must be re-engineered (a time-consuming process). In the modern era, the architectures are continuously evolving trying to bring higher performance while keeping shorter time to market, therefore developers do not prefer to do the re-engineering, which requires significant time investment.

Modern parallel computing architectures are complex due to higher core counts, different multi-threading, memory hierarchy, computation capabilities, and processor architecture. This disparity of architecture increases the number of available compiler optimization flags and makes compilers unable to efficiently utilize the available resources. Tuning these parameters manually is not just unfeasible, but also introduces scalability and portability issues. Machine learning and meta-heuristics promise to address compiler problems, such as, selecting compiler optimization flags or heuristic-guided compiler optimizations.

In what follows, we discuss the existing state-of-the-art approaches that use machine learning and meta-heuristics for software optimization for code optimization and code generation. Thereafter, we discuss the limitations and identify possible future research directions.

### 4.1 Code optimization

Code optimization will not change the program behavior but will optimize the code to reach optimization goals (reducing the execution time, energy consumption, or required resources).

Compiler optimization techniques include loop unrolling, splitting and collapsing, instruction scheduling, software pipelining, auto-vectorization, hyper-block forma-

tion, register allocation, and data pre-fetching [88]. Different device-specific code optimization techniques may behave differently in various architectures. Furthermore, choosing more than one optimization technique does not necessarily result in better performance, sometimes combination of different techniques may have negative impact on the final output. Hence, manually writing hard-code heuristics is impractical, and techniques that intelligently select the compiler transformations that result in higher application benefits in a given context are required.

Within the scope of this survey, scientific publications that use machine learning for code optimization at compile time include [1, 17, 34, 35, 57, 69, 87, 95, 99], whereas scientific publications that use meta-heuristics for code optimization include [21, 88, 92, 93]. Table 4 lists the characteristics of the selected primary studies that address code optimization at compile time. Such characteristics include: the algorithm used for optimization, the optimization objectives, the considered features that describe the application being optimized, and type of optimization (on-line or off-line). We may observe that besides the approach proposed by Tiwari and Hollingsworth [92], the rest of them focus on off-line optimization approaches and they are based on historical data (knowledge) that is gathered from previous runs.

#### **RQ1: Software optimization goals for compile-time code optimization:**

- loop unrolling; instruction scheduling; partitioning of irregular and stream applications; determining the best compilation parameters; determining whether parallelism is beneficial; tuning compiler heuristics;

As we mentioned earlier, different optimizations can be performed during compilation. We may see that some researchers focus on using intelligent techniques to identify loops that would potentially execute more efficiently when unrolled [69], or selecting the loop unroll factor that yields the best performance [87]. Instruction scheduling [17], partitioning strategy for irregular [57] and streaming [99] applications, determining the list of compiler optimizations that results in the best performance [21, 35, 92] are also addressed by the selected scientific publications. Furthermore, Tournavitis et al. [95] use SVMs to determine whether parallelization of the code would be beneficial, and which scheduling policy to select for the parallelized code.

#### **RQ2: Software optimization algorithms used for compile-time code optimization:**

- *machine learning* - nearest neighbor classifier; support vector machines; decision trees; ruled set induction; predictive search distribution;
- *meta-heuristics* - genetic algorithms; hill climbing; greedy algorithm; parallel rank order;

With regards to the machine learning algorithms used for code optimization, Nearest Neighbor (NN) classifier [1, 57, 87, 99], Support Vector Machine (SVM) [87, 95], and Decision Tree (DT) [69] are the most popular. Other algorithms, such as Ruled Set Induction (RSI) [17], and Predictive Search Distribution (PSD) [34, 35] are also used for code optimization during compilation. Whereas, approaches that are based on

**Table 4** Characteristics of the approaches that use machine learning or meta-heuristics for code optimization

References	Algorithm	Objectives	Features	On/Off-line
[69]	DT	Identify loops to unroll	Loop characteristics (# memory accesses; # arithmetic operations; # statements; # control statements; # iterations)	Off-line (sup.)
[87]	NN, SVM	Select the most beneficial loop unroll factor	Loop characteristics (# floating point operations; # operands; # memory operations; critical path length; # iterations)	Off-line (sup.)
[17]	RSI	Determine whether to apply instructions scheduling	Code-block characteristics (# instructions; # branches; # calls; # stores; # returns; int/float/sys_func_unit instructions)	Off-line (sup.)
[34,35]	PSD	Determine the most effective compiler optimizations	Static program features (# basic blocks in a method; # normal/critical/abnormal CFG edges; # int/float operations)	Off-line (sup.)
[57]	kNN	Determine the best partitioning strategy of irregular applications	Static program features (# basic blocks; # instructions; loop probability; branch probability; data dependency)	Off-line (sup.)
[99]	NN	Determine the best partitioning strategy of streaming app.	Program features (pipeline depth; split-join width; pipeline/split-join work; # computations; # load/store ops)	Off-line (sup.)
[95]	SVM	Determine whether parallelism is beneficial; select the best scheduling policy	Static program features (# instructions; # load/store; # branches; # iterations); dynamic program features (# data accesses; # instructions; # branches)	Off-line (sup.)
[1]	IMDM; MM; NN	Reduce the number of required program evaluations in iterative compiler optimization; analyze program similarities	Program features (type of nested loop; loop bound; loop stride; # iterations; nest depth; # array references; # instructions; # load/store/compare/branch/divide/call/generic/array/memory copy/other instructions; int/float variables)	Off-line (sup.)

**Table 4** continued

References	Algorithm	Objectives	Features	On/Off-line
[88]	GP	Tuning compiler heuristics	hyper-block formation features; register allocation features; data pre-fetching features.	Off-line (unsupervised)
[21]	GrA; GA; HC RP	Tuning the compilation process through adaptive compilation	–	Off-line
[92,93]	PRO	Tune generated code; determine the best compilation parameters	Architectural parameters (cache capacity; register capacity); application specific parameters	On-line

Please note that, because of space limitation, we do not list all of the considered optimization features

search-based algorithms use Genetic Algorithm (GA), Hill Climbing (HC), Greedy Algorithm (GrA), and Parallel Rank Ordering (PRO) for code optimization during compile-time [21, 92, 93].

### RQ3: Considered features during compile-time code optimization:

- *loop characteristics* - number of memory accesses, arithmetic operations, statements, loop iterations, floating point operations, operands;
- *code-block characteristics* - number of instructions, branches, calls, stores, returns, instructions;
- *program features* - type of nested loop; loop bound; loop stride; nest depth;
- *static program features* - number of basic blocks in a method, CFG edges, operations, load/store operations; data dependency; loop and branch probability;
- *dynamic program features* - number of data accesses, instructions, branches;
- *architectural parameters* - cache capacity; register capacity;
- *application specific parameters; hyper-block formation features; register allocation features; data pre-fetching features;*

To achieve the aforementioned objectives, a representative set of program features is extracted through static code analysis, which are considered to be the most informative with regards to the program behavior. The selection of such features is closely related to the optimization goals. For example, to identify loops that benefit from unrolling, Monsifrot et al. [69] use loop characteristics such as, number of memory accesses, arithmetic operations, code statements, control statements, and loop iterations. Such loop characteristics are also used to determine the loop unroll factor [87]. Characteristics related to a specific code block (such as number of instructions, branches, calls, stores) are used when deciding whether applications benefit from instruction scheduling [17]. Determining the partitioning strategy of irregular applications is based on static program features related to basic block, loop characteristics, and the data dependency [57]. Features such as pipeline depth, load/store operations per instruction, number of computations, and computation-communication ratio are used when determining partitioning strategy of streaming applications [99]. Tiwari and Hollingsworth [92] consider architectural specifications such as cache and register capacity, in addition to the application specific parameters, such as tile size in a matrix multiplication algorithm.

## 4.2 Code generation

The process of transforming code from one representation into another one is called code generation. We call “machine code generation” the code transformation from the high level to low level representation (that is ready for execution), whereas “source code generation” indicates in this paper the source-to-source code transformation.

In the context of parallel computing, a source-to-source compiler is an automatic parallelization compiler that can automatically annotate a sequential code with parallel code annotations (such as, OpenMP pragma directives or MPI code statements).

Source-to-source compilers may alleviate the portability issue, by enabling to automatically translate the code into an equivalent representation of the code that is ready to be compiled and executed on target architectures.

In this section, we focus on source code generation techniques that can:

- generate device-specific code from other code representations,
- generate multiple implementations of the same code, or
- automatically generate parallel code from sequential code.

During the process of porting applications, programmers are faced with the following problems: (1) demand of device-specific knowledge and API; (2) difficulties to predict whether the application will have performance benefits before it is ported; (3) there exist a large number of programming languages and models that are device (types and manufacturer) specific.

To address such issues, researchers have proposed different solutions. In Table 5, we list the characteristics of these solutions such as, optimization algorithm, optimization objectives, and considered features during optimization.

#### RQ1: Software optimization goals for compile-time code generation:

- generating device-specific code; mapping applications to accelerating devices;
- generating multi-threaded loop versions; source-to-source transformations;
- determining the list of program method transformations; enabling writing multiple versions of algorithms and algorithmic choices at language level; auto-tuning algorithmic choices and switching between them during program execution;
- determining optimal work distribution between CPU and GPU.

The optimization objectives are derived from the aforementioned portability challenges. For example, to alleviate the demand for device-specific knowledge, Beach and Avis [7] aim to identify candidate kernels that would likely benefit from parallelization, generate device-specific code from high-level code, and map to the accelerating device that yields the best performance. Similarly, Fonseca and Cabral [31] propose the automatic generation of OpenCL code from Java code. Ansel et al. [5] propose the PetaBricks framework that enables writing multiple versions of algorithms, which are automatically translated into C++ code. The runtime can switch between the available algorithms during program execution. Luk et al. [58] introduce Qilin that enables source-to-source transformation from C++ to TBB and CUDA. It uses machine learning to find the optimal work distribution between the CPU and GPU on a heterogeneous system.

#### RQ2: Software optimization algorithms used for compile-time code generation:

- *machine learning* - decision trees; near neighbors; linear regression;

Decision Trees (DT) [7,31], k-Nearest Neighbor (kNN) [19], Cost Sensitive Decision Table (CSDT), Naive Bayes (NB), Support Vector Machine (SVM), Multi-layer

**Table 5** Characteristics of the approaches that use machine learning or meta-heuristics for code generation

References	Algorithm	Objectives	Features	On/Off-line
[7]	DT	Generate device-specific code from high-level code; map applications to accelerating devices.	Loop (kernel) characteristics (data precision, amount of computation performed and memory access characteristics)	Off-line (sup.)
[19]	kNN	Generate multi-threaded loop versions; select the most suitable one at run-time	Static code features (loop nest depth, # arrays used); dynamic features (data set size)	off-line (sup.)
[31]	NB, SVM, MFL, CSDT, LR	Source-to-source transformation of data-parallel applications; predict the efficiency and select the suitable device.	Static program features (outer/finner access/write; basic operations; ...); dynamic program features (data-io; data-from; ...)	Off-line (sup.)
[5]	–	Enable writing multiple versions of algorithms and algorithmic choices at the language level; auto-tuning of the specified algorithmic choices; switch between the available algorithms during program execution	–	Off-line
[58]	LR	Determine the optimal work distribution between the CPU and GPU	Runtime algorithm parameters (input size) and hardware configuration parameters	On-line
[81]	–	Distribute data-parallel portions of a program across heterogeneous computing resources;	–	–
[77]	LRPR	Determine the list of program method transformations that result in lower compilation time	General program features (# instructions; # load/store operations; # float operations); loop-based features (# loops types; # loop statements)	Off-line (sup.)

Please note that, because of space limitation, we do not list all of the considered optimization features

Perceptron (MPL) [31], Linear Regression (LR) [31, 58], and Logistic Regression (LRPR) [77] machine learning algorithms are used during the code-generation.

**RQ3: Considered features during compile-time code generation:**

- *loop characteristics* - data precision; amount of computation performed; memory access characteristics; loop type; loop statement
- *general program features* - number of instructions; load/store operations; floating point operations
- *static code features* - loop nest depth; number of arrays; outer/inner access/write; basic operations;
- *dynamic features* - data set size; data-to; data-from;
- *runtime algorithm parameters; hardware configuration parameters*

Beach and Avis [7] considered static loop characteristics to achieve their objectives, whereas Chen and Long [19] use both static and dynamic program features to generate the multi-threaded versions of a selected loop, and then select the most suitable loop version at run-time. Combination of static code features (extracted at compile time), and dynamic features (extracted at run-time) are also used to determine the most suitable processing device for a specific application [31]. To determine the best workload distribution of a parallel application, Luk et al. [58] consider algorithm parameters and hardware configuration parameters. Pekhimenko and Brown [77] consider general and loop-based features to determine the list of program method transformation during code generation that would reduce the compilation time.

### 4.3 Observations, challenges, and future directions

In this section, we first discuss the advantages of meta-heuristics and machine learning methods for software optimization at compile-time, followed by a discussion about their limitations. Thereafter, we discuss the future directions.

In Table 6, we list each of the machine learning and meta-heuristic methods used for compile-time software optimization. For each of the used methods, we provide the advantages, such as performance improvement, speedup, and prediction accuracy.

While most of the approaches discussed in this review present significant performance improvement, which is important towards having intelligent compilers that require less engineering effort to provide satisfactory code execution performance, indications that there is still room for improvement can be observed in Stephenson et al. [88] and andWang and O’boyle [99].

Limitations of the compile-time software optimization approaches that use machine learning or meta-heuristics are listed in Table 7, which include: (1) limitation to a specific programming language or model [95], (2) forcing developers to use extra annotations on their code [58], or use not widely known parallel programming languages [5], (3) focusing on single or simpler aspects of optimizations techniques (ex: loop unrolling, unrolling factor, instruction scheduling) [17, 69, 87], whereas more complex compiler optimizations (that are compute-intensive) are not addressed sufficiently.

**Table 6** Advantages of meta-heuristics and machine learning methods for compile-time software optimization

Method	Advantages
<i>Machine learning</i>	
Decision trees	Monsifrot et al. [69] reports up to 3% performance improvement for loop unrolling; Beach and Avis [7] reports performance achievement within 15% of the performance achieved manually, ported code
Support Vector Machines	Stephenson and Amarasinghe [87] report that SVM and NN can predict the optimal unroll factor for a given loop 65% of the time or the near optimal one 79% of the time. Tournavitis et al. [95] uses SVM to decide whether to parallelize loop candidates and achieves 96% of the performance of hand-tuned code. Fonseca and Cabral [31] report 92% of prediction accuracy when using SVM to decide whether kernels should be executed on GPU or CPU
(k) Nearest Neighbor	Liu et al. [57] report 5.41% performance improvement compared to hand-coded compiler optimizations. Wang and O'Boyle [99] use NN to predict the partitioning structure of applications. The authors achieve up to 1.9x speedup compared to the default partitioning strategy, which is 60% of the ideal one. Chen and Long [19] report that 87% of the highest performance improvement can be achieved using NN
Ruled Set Induction	Carvazos and Moss [17] use RSI to determine whether or not to apply instruction scheduling on a code block and reported achievement of 90% performance improvement compared to schedule always method
Regression Based Algorithms	Pekhimenko and Brown [77] use regression techniques to determine the optimal heuristic parameters and report two fold speedup of the compilation process while maintaining the same code quality
Decision Tables	In Fonseca and Cabral [31], the up to 92 % prediction accuracy of DT, NB, and MLP to decide whether kernels should be executed on the GPU or CPU is significant to achieve 65x speedup over sequential Java programs
Predictive Search Distribution	Fursin et al. [34,35] use PSD to select the best compiler optimizations and report 11% performance improvement
<i>Meta-heuristics</i>	
Greedy Algorithm and Hill Climbing	Cooper et al. [21] use meta-heuristics to find the optimal compilation parameters while reducing the number of evaluations during search space exploration from 10000 to a single one using profiling data and estimated virtual execution
Genetic Algorithm	Sandrieser et al. [88] obtain speedup of 23% for hyper-block formation
Parallel Rank Ordering	Tiwari and Hollingsworth [93] and Tiwari et al. [92] use PRO for automatic tuning of compilation process and report 46% performance improvement compared to the original code

**Table 7** Limitations of the existing studies that use machine learning and meta-heuristics for compile-time software optimization

References	Focus	Limitations
[17, 69, 87, 88]	Single aspects of code optimization	Control single/few and simple optimization (such as: loop unrolling; instruction scheduling; hyper-block formation). Considering multiple and more complex compiler optimizations is more challenging
[34, 35, 77, 92, 93]	Determining the most effective compiler optimization	Training is based in random data sampling and it requires large number of samples, which may reduce its effectiveness
[57, 58, 99]	Determining the best partitioning strategy	Assumes that for any two functions with similar features, the same partitioning strategy can be used
[95]	Determining loops that benefit from parallelization and their best scheduling policy	Targets OpenMP loop constructs only. Uses profiling to detect loop candidates, which may significantly increase the compilation time
[1, 21]	Adaptive tuning of the compilation process	Profiling data needs to be collected to perform the virtual executions. Takes too long to find the optimal transformations. Efficient for simple models, but results in lower prediction accuracies for more complex problems
[7, 19, 31]	Device-specific code generation; mapping applications to accelerators	The prediction model requires significant training data for accurate mapping decisions. Lack of training data may result in performance degradation
[19]	Generating multi-threaded versions of the loop	The size of the executable file may dramatically increase for applications with large parallel code, and hardware architectures that consist of multiple multi-core processing units
[5, 31, 81]	Source-to-source transformation	Limited to map-reduce operations. The automatic code generation is limited to specific features of Java code. Not all Java code can be translated to OpenCL
[5]	Auto-tuning; run-time library	PetaBricks requires that developers write their application using non widely known programming languages. The performance of PetaBricks is closely dependent on the architecture where the auto-tuning was performed

Furthermore, optimizations based on features derived from static code analysis provide poor global characterization of the dynamic behavior of the applications, whereas using dynamic features requires application profiling, which adds additional execution overhead to the program under study. This additional time can be considered negligible for applications that are executed multiple times after the optimization, however it represents overhead for single-run applications. Approaches that generate many multi-threaded versions of the code [19] might end up with dramatic code increases that make difficult the applicability to embedded parallel computing systems with limited resources. Adaptive compilation techniques [21] add non-negligible compilation overhead.

Future research should address the identified shortcomings in this systematic review by providing intelligent compiler solutions for general-purpose languages (such as, C/C++) and compilers (for instance, GNU Compiler Collection) that are widely used and supported by the community. Many compiler optimization issues are complex and require human resources that are usually not available within a single research group or project.

## 5 Run-time

The run-time program life-cycle is the time during which the program is running (that is, being executed) and it is also known as execution-time. Software systems that enable running programs to interact with the execution environment are known as run-time systems. The run-time environment contains environment information, such as, the available resources, existing workload, and scheduling policy. A running program can access the execution environment information via the run-time system.

In the past, the choice of architecture and the algorithms was considered during the design and implementation phase of software life-cycle. Nowadays, there are various multi- and many-core processing devices, with different performance and energy consumption characteristics. Furthermore, there is no single algorithm implementation that can exploit the full processing potential of these diverse processing elements. Often it is not possible to know if an application performs better on device X or Y before the execution. The performance of a program is determined by the properties of the execution context (program input, type of available processing elements, current system utilization...) that is known at run-time. Some programs perform better on device X when the input size is large enough, but worse for smaller input sizes. Hence, decisions whether a program should be run on X or Y, or which algorithm to use are postponed to run-time.

In this study, we focus on optimization methods used in different run-time systems that use machine learning or meta-heuristics to optimize the program execution. Such run-time systems may be responsible for partitioning programs into tasks and scheduling these tasks to different processing devices, selecting the most suitable device(s) for a specific task, selecting the most suitable algorithm or the size of the input workload, selecting the number of processing elements or clock frequency, and many more different system run-time configuration parameters to achieve the specified goals including the performance, energy efficiency, and fault tolerance. Specifically, we focus on two major run-time activities: *scheduling* and *adaptation*.

In what follows, we discuss the related state-of-the-art run-time optimization approaches for scheduling and adaptation. Thereafter, we summarize the limitations of the current approaches and discuss possible future research directions.

## 5.1 Scheduling

According to the Cambridge Dictionary,<sup>1</sup> scheduling is “the job or activity of planning the times at which particular tasks will be done or events will happen”. In context of this paper, we use the term scheduling to indicate mapping the tasks onto the processing elements and determining the order of task execution to minimize the overall execution time.

Scheduling may strongly influence the performance of parallel computing systems. Improper scheduling can lead to load imbalance and consequently to sub-optimal performance. Researchers have proposed different approaches that use meta-heuristics or machine learning to find the best scheduling within a reasonable time.

Based on whether the scheduling algorithms can modify the scheduling policy during program execution, generally scheduling algorithms are classified in static and dynamic.

### 5.1.1 Static scheduling

Static scheduling techniques retain an unchanged policy until the end of program execution. Static approaches assume that the number of tasks is fixed, known before execution starts, and that accurate information of their running times is known. Static approaches usually use analytical models to estimate the computation and communication cost, where the work distribution is performed based on these estimations. The program execution time is essential for job scheduling. However, accurately predicting/estimating the program execution time is difficult to achieve in shared environments where system resources can dynamically change over time. Inaccurate predictions may lead to performance degradation [20].

Table 8 lists the characteristics (such as optimization algorithm, objective, and features) of scientific publications that use machine learning and/or meta-heuristics for static scheduling.

#### RQ1: Software optimization goals for run-time static scheduling:

- mapping program parallelism to multi-core architectures; determining the optimal number of threads; mapping applications to the most appropriate processing device; reducing memory latency and contention; mapping threads to specific cores; determining workload distribution on heterogeneous systems; determining near-optimal system configuration parameters;

With regards to static scheduling, the attention of recent research that use machine learning and meta-heuristics is in the following optimization objectives: mapping program parallelism to multi-core architectures [98], mapping applications to the most

<sup>1</sup> Cambridge Dictionary, <http://dictionary.cambridge.org/dictionary/english/scheduling>.

**Table 8** Characteristics of the approaches that use machine learning or meta-heuristics for static scheduling

References	Algorithm	Objectives	Features	On/Off-line
[98]	ANN; SVM	Mapping computations to multi-core CPUs; determine the optimal thread number;	Code features (# static instructions; # load/store operations; # branches); data and dynamic features (L1 data cache miss rate; branch miss rate)	Off-line (sup.)
[40]	SVM	Mapping computations to the suitable processing device	Static code features (# int/float/math operations; barriers; memory accesses; % local/coalesced memory accesses; compute-memory ratio)	Off-line (sup.)
[15]	ID3 DT	Mapping threads to specific cores; reduce memory latency and contention	Program features (transaction time ratio; transaction abort ratio; conflict detection policy; conflict resolution policy; cache misses)	Off-line (sup.)
[71]	L; MP; IB1; IBk; KStar ...	Reducing the training data; select the most informative training data; mapping application to processors	–	Off-line (sup.)
[64]	BDTR	Determine workload distribution of data-parallel applications on heterogeneous systems	Hardware configuration (# threads; # cores; # threads/core; thread affinity); application parameters (input size)	Off-line (sup.)
[62]	SA	Determine near-optimal system configuration parameters of heterogeneous systems	System configuration parameters (#threads/thread_affinity/ workload_fraction on host/device)	Off-line (sup.)
[63, 65]	BDTR; SA	Determine near-optimal system configuration on heterogeneous systems	Available resources; scheduling policy; and the workload fraction	Off-line (sup.)
[2, 14, 104, 105]	GA	Task scheduling	–	Off-line (sup.)

appropriate processing device [40, 71], mapping threads to specific cores [15], and determining workload distribution on heterogeneous parallel computing systems [62–65].

**RQ2: Software optimization algorithms used for run-time static scheduling:**

- *machine learning* - artificial neural networks; support vector machines; (boosted) decision trees; logistic; multi-layer perceptron; IB1; IBk; KStar; Random Forest; LogitBoost; multiclass classifier; NNge; ADTree; random tree;
- *meta-heuristics* - simulated annealing; genetic algorithms;

To achieve the aforementioned optimization objectives, machine learning algorithm such as, Artificial Neural Networks (ANN), Support Vector Machines (SVM), and (Boosted) Decision Trees (BDTR) are used [15, 40, 64, 98]. An approach that combines a number of machine learning algorithms, including, Logistic (L), Multilayer Perceptron (MP), IB1, IBk, KStar, Random Forest, Logit Boost, Multi-Class-Classifier, Random Committee, NNge, ADTree, and RandomTree, to create an active-learning query-committee with the aim to reduce the required amount if training data is proposed by Ogilvie et al. [71]. A combination of Simulated Annealing (SA) and boosted decision tree regression to determine near optimal system configurations is proposed by Memeti and Pllana [63]. The use of Genetic Algorithms (GA) for task scheduling has been extensively addressed by several researchers [2, 14, 104, 105].

**RQ3: Considered features during run-time static scheduling:**

- *static program features* - number of static instructions; number of load/store operations; number of branches; barriers; memory accesses; compute-memory ratio; transaction time ratio; transaction abort ratio; conflict detection and resolution policy;
- *data and dynamic features* - L1 data cache miss rate; branch miss rate;
- *hardware characteristics* - number of threads, cores, threads per core; thread affinity;
- *system configuration parameters* - input size; workload fraction on host and accelerating devices;

The list of considered system features for optimizing of parallel computing systems is closely related to the optimization objectives, target applications and architecture. For example, Castro et al. [15] consider transaction time and abort ratio, conflict detection and resolution policy to map thread to specific cores and reduce memory latency and contention in software transactional memory applications running on multi-core architectures. Static code features, such as number of instruction, memory operations, math operations, and branches, are considered during the mapping of applications to the most suitable processing devices [40, 98]. While such approaches consider application specific features, researchers have demonstrated positive improvement results in approaches that do not require code analysis. Instead, they rely on features such as the available system resources and program input size during the optimization process (that is determining the workload distribution of data-parallel applications) [62–64].

### 5.1.2 Dynamic scheduling

Dynamic scheduling algorithms take into account the current system state and modify themselves during run-time to improve the scheduling policy. Dynamic scheduling does not require prior knowledge of all task properties. To overcome the limitations of the static scheduling, various dynamic approaches are proposed, including work stealing, partitioning and assigning tasks on the fly, queuing systems, and task-based approaches. Dynamic scheduling is usually harder to implement; however, the performance gain may be better than static scheduling.

Table 9 lists the characteristics (such as optimization algorithm, objective, and features) of scientific publications that use machine learning and/or meta-heuristics for dynamic scheduling.

#### RQ1: Software optimization goals for run-time dynamic scheduling:

- mapping tasks to processing devices; partitioning tasks between performance clusters and the cloud; determining resource allocation; predicting thread mapping strategy; predicting the optimal number of threads; improving scheduling algorithms; determining scheduling policy; minimizing the make-span; mapping computation kernels to heterogeneous GPU accelerated systems; determining optimal system configuration; load balancing; determining performance aspects, such as execution time ad power consumption; selecting the best algorithm implementation variant; reducing the number of training data required to build prediction models;

With regards to the optimization objectives, considered scientific publications aim at: (1) determining the optimal number of threads for a given application [30, 41, 102, 103]; (2) determining the application execution time [9, 23, 51, 52, 54]; (3) mapping tasks to processing devices [4, 15, 76, 80]; (4) partitioning tasks between high performance clusters [60]; (5) predicting resource allocation in the cloud [61]; (6) improving scheduling algorithms [36, 43]; (7) minimizing the make-span [11, 24, 44, 74, 75, 85, 104]; (8) selecting near optimal system configurations [56]; and (9) reducing the number of training examples required to build prediction models [55].

#### RQ2: Software optimization algorithms used for run-time dynamic scheduling:

- *machine learning* - artificial neural networks; regression and filtering techniques; support vector machines; (boosted) decision trees; logistic; multi-layer perceptron; IB1; IBk; KStar; Random Forest; LogitBoost; multiclass classifier; NNge; ADTree; random tree; dispatch tables; Naive Bayesian classifier; decision diagrams;
- *meta-heuristics* - (adaptive) greedy algorithm; simulated annealing; genetic algorithms; hill climbing; particle swarm optimization;

Artificial neural network (ANN) [30, 41, 43, 44], regression (LR, QR, PR) [9, 24, 36, 52, 54], support vector machines (SVM) [23, 51, 76], and decisiontrees (DT) [16,

**Table 9** Characteristics of the approaches that use machine learning or meta-heuristics for dynamic scheduling

References	Algorithm	Objectives	Features	On/Off-line
[30]	ANN	Determine the best number of threads	Static features (# load/store ops; # instructions; # branches); dynamic features (# processors; # workload threads; run queue length; ldavg-1; ldavg-5)	Off-line (sup.)
[54]	R&F	Determine the application execution time in shared environments	Program input parameters; # processors; resource status	Off-line (sup.)
[76]	SVM	Mapping tasks to processing devices	# tasks in the queue; the ready times of the machines; computing capabilities of each machine	Off-line (sup.)
[60]	GirA	Evenly partitioning tasks between high performance clusters and the cloud	Estimated execution time determined by monitoring the actual exec. time of data or tasks chunks	On-line
[61]	–	Predicting resource allocation for business processes in the Cloud	Runtime metrics of a process and its behavior	Off-line
[16]	ID3 DT	Predicting a thread mapping strategy for STM applications	Transactional Time/Abort Ratio; Conflict Detection/Resolution Policy; Last-Level Cache Miss	Off-line (sup.)
[43]	ANN	Improve the effectiveness of grid scheduler decisions	Characteristics of the tasks and machines	Off-line
[44]	ANN; GA	Improve the makespan	Security demands, workload of task, and the output size	Off-line & On-line
[36]	LR	Improving the scheduling algorithms using machine learning techniques	Job arrival time; required resources; # running jobs; occupied resources	On-line
[11]	–	Optimize the task scheduling on heterogeneous platforms	Input data; data transfers; task performance; platform features	Off-line & On-line
[41]	ANN	Predict the optimal number of threads	Program features and workload features	Off-line

**Table 9** continued

References	Algorithm	Objectives	Features	On/Off-line
[102, 103]	Adaptive LR	Determine the number of threads and scheduling policy for each parallel region	Inter-thread data locality, instruction mix and load imbalance	–
[74, 75]	GA	Minimize the make-span; dynamic task scheduling in heterogeneous systems	Task properties(arrival time; dependency); system properties (network; processors)	On-line
[4]	Adaptive GrA	Mapping of computation kernels on heterogeneous GPUs accelerated systems.	Profiling information (execution time; data-transfer time); hardware characteristics	Off-line
[56]	HC	Selecting optimal per task system configuration for MapReduce applications	Map-reduce parameters (# mappers; # reducers; slow start; io sort.mbh; # virtual cores)	On-line
[85]	PSO; SA	Dynamic scheduling of heterogeneous tasks on heterogeneous processors; load balancing;	Task properties (execution time; communication cost; fitness function); hardware properties (# processors)	–
[104]	GA	Dynamic load-balancing where optimal task scheduling can evolve at run-time	–	On-line
[80]	–	Mapping tasks to heterogeneous architectures	Architectural trade-offs; computation patterns; application characteristics;	On-line
[24]	PR	Dynamic scheduling and performance optimization for heterogeneous systems	Kernel execution time; machine parameters; input size; input distribution var.; instrumentation data;	Off-line (supervised)
[9, 52]	LR; QR	Prediction of performance aspects (e.g. execution time, power consumption) of implementation variants	System information (resource availability and requirements; estimated performance of implementation variants; input availability)	Off-line (supervised)
[55]	DT	Reducing the number of training data required to build prediction models	Input parameters (e.g. # cores; # accelerators)	Off-line (supervised)
[23, 51]	DT; DD; NB; SVM	Use meta-data from performance aware components to predict the expected execution time; select the best implementation variant and the scheduling policy	Input parameters (e.g. size); system available resources; meta-data	Off-line (supervised)

[23, 51, 55] are the most popular machine learning algorithms used for optimization in the scientific publications considered in this study. Whereas, genetic algorithms (GA) [44, 74, 75, 104], greedy-based algorithms (Gra) [4, 60], hill-climbing (HC) [56], particle swarm optimization (PSO) [85], and simulated annealing (SA) [85] are used as heuristic based optimization approaches for dynamic scheduling.

### RQ3: Considered features during run-time dynamic scheduling:

- *static and dynamic features* - number of load/store operations; number of instructions and branches; number of processors and workload threads; run-queue length;
- *task related features* - number of tasks in the queue; machine ready time; estimated task execution time; task performance; arrival time; dependency;
- *runtime information* - metrics of a process and its behavior; last level cache misses; job arrival time; running jobs; data transfers; inter-thread data locality; instruction mix and load imbalance; execution time; data-transfer time; fitness function;
- *application specific and workload characteristics* - transactional time and abort ratio; conflict detection and resolution policy; required resources; input data; number of mappers and reducers in map-reduce applications;
- *hardware characteristics* - machine computing capability; occupied resources; platform features; network properties; processor properties;

Approaches such as [16, 56, 60, 61, 102, 103] focus on features collected dynamically during program execution, such as, estimated execution time determined through analysis of profiling data, information related to tasks (arrival time, number of currently running tasks). Whereas other approaches combine static features collected at compile-time with dynamic ones collected at run-time [30, 43, 74–76], program input parameters, and hardware related information [11, 24, 41, 54, 80]. Similar to the static scheduling techniques, the selection of such features is closely related to the optimization objectives. For example, Zhang et al. [102, 103] consider the inter-thread data locality when tuning OpenMP applications for hyper-threaded SMPs; Page and Naughton [74, 75] consider task properties, such as, task arrival time and task dependency, when scheduling dynamically tasks in heterogeneous distributed systems. Features such as security demands, workload of tasks, and the output size are considered to train the ANN for optimization of scheduling process and maximization of resource usage in the cloud [44].

## 5.2 Adaptation

According to the Cambridge Dictionary,<sup>2</sup> adaptation is “the process of changing to suit different conditions”. In this paper, we use the term adaptation to refer to the property of systems that are capable of evaluating and changing their behavior to achieve specified goals with respect to performance, energy efficiency, or fault tolerance. In dynamic

<sup>2</sup> Cambridge Dictionary, <http://dictionary.cambridge.org/dictionary/english/adaptation>.

environments, modern parallel computing systems may change their behavior by: (1) changing the number of used processing elements to optimize system resource allocation; (2) changing the algorithm or implementation variant that yields to better results with respect to the specified goals; (3) reducing the quality (accuracy) of the output to meet the performance goals; or (4) changing the clock frequency to reduce energy consumption.

The studied literature in this paper provide examples that adaptation (also referred to as self-adaptation) proved to be an effective approach to deal with the complexity, variability, and dynamism of modern parallel computing systems. Table 10 lists the characteristics (such as, adaptation method and objectives, monitored and tuned parameters) of the scientific publications that use adaptation for software optimization of parallel computing systems.

#### RQ1: Software optimization goals for run-time adaptation:

- selecting the most suitable algorithm implementation; applying user defined actions to change the program behavior; adapting lock's internal implementation mechanisms; determining the ideal data structure knob settings; adaptive mapping of computations to processing elements; adapting applications to meet the user defined goals;

With regards to the adaptation objectives, Thomas et al. [91] use a custom adaptation loop to adaptively select the most suitable algorithm implementation for a given input data set and system configuration. Hoffmann et al. [46, 47] use an observe-decide-act (ODA) feedback loop to adaptively apply user defined actions to change the program behavior in favor of achieving some user-defined goals, such as energy efficiency and throughput. Adaptation methods are used in the smart-locks library [28], which can change its behavior at run-time to achieve certain goals. Similarly, in [29] adaptation methods are used for optimizing data structure knobs. Adaptive mapping of computations to the processing units is proposed by Luk et al. [58]. The Antarex [84] project aims at providing means for application tuning and adaptation for energy efficient heterogeneous high-performance computing systems, by providing a domain specific language that allows specifying adaptation goals at compile-time.

#### RQ2: Software optimization algorithms used for run-time adaptation:

- *machine learning* - decision trees; reinforcement learning; linear regression;
- *other* - custom adaptation loop; observe-act-decide loops; lock acquisition scheduling;

During the process of adaptation, all of the approaches proposed in the considered scientific publications, have at least three components of an adaptation loop, including monitoring, deciding, and acting. For example, Thomas et al. [91] monitor architecture and environment parameters, then uses a decision tree to analyze such information, and perform the required changes (in this case selecting an algorithm implementation). Similarly, Hoffmann et al. [47] use the so called observe-decide-act (ODA) feedback loop to monitor performance related information (retrieved using the

**Table 10** Characteristics of the optimization approaches based on adaptation techniques

References	Method	Adaptation objectives	Monitored parameters	Tuned parameters
[91]	Custom adaptation loop; DT	Select the most suitable algorithm implementation	Architecture and system information (available memory, cache size, # processors); performance characteristics	Algorithm implementation
[46,47]	ODA	Apply user defined actions to change the program behavior	Performance information retrieved using application heartbeats [46]	User defined actions (such as: adjusting the clock speed)
[28]	Lock Acquisition Scheduling; RL	Adapt the lock's internal implementation	Reward signal (heart rate) retrieved using application heartbeats	Change the lock scheduling policy
[29]	RL	Determine the ideal data structure knob settings	Reward signal (throughput heart rate); support for external perf. monitors	Adjusting the <i>scancount</i> value and performance-critical knob of Flat Combining algorithm
[58]	LR	Adaptive mapping of computations to PE	Execution-time of parts of the program	Choosing the mapping scheme (static or adaptive)
[84]	DSL	Adapt applications to meet user defined goals	Contextual information, requirements, resources availability	User defined actions (altering resource alloc. and task mapping)

application heartbeats [46]) and use the heart-rate to take some user defined actions, such as adjusting the clock speed, allocating cores, or change the algorithm. Reinforcement learning (RL), an on-line machine learning algorithm, is used to help with the adaptation decisions in both smart-locks [28] and smart data-structures [29], whereas linear regression (LR) is used by Luk et al. [58] for choosing the mapping scheme of computations to processing elements.

**RQ3: Considered features during run-time dynamic scheduling:**

- *hardware characteristics* - available memory; cache size; number of processors; resource availability
- *performance characteristics* - heartbeat reward signal; throughput; external performance monitors; execution time;
- *contextual information; requirements*;

In Table 10 we list two types of parameters, the monitored parameters, used to evaluate whether adaptation goals have been met, and tuned parameters, which are basically defined actions that will change the program behavior until the desired goals are achieved. For monitoring, architecture and environment variables (such as, available memory, cache size, number of processors), and performance characteristics are considered by Thomas et al. [91]. Performance related information retrieved from the heartbeats monitor are used as monitoring parameters in the following scientific articles [28, 29, 47]. Luk et al. [58] rely on the execution time of parts of the program, whereas the Antarex framework uses contextual information, requirements, and resource availability for monitoring the program behavior. As tuning parameters, the following are considered, selecting the algorithm implementation [47, 91], adjusting the clock speed, core allocation, select algorithm [47], change lock scheduling policy [28], adjust the scandcount [29], change mapping scheme [58], and altering resource allocation and task mapping [84].

### 5.3 Observations, challenges and research directions

In this section, we first discuss the advantages of meta-heuristics and machine learning methods for software optimization at run-time, followed by a discussion about their limitations. Thereafter, we discuss the future directions.

In Table 11, we list each of the machine learning and meta-heuristic methods used for run-time software optimization. For each of the used methods, we provide the advantages, including performance improvement, speedup, or prediction accuracy.

**Table 11** Advantages of meta-heuristics and machine learning methods for run-time software optimization

Method	Advantages
Machine Learning	
Artificial Neural Network	Emani et al. [30] report speedup of up to $3.2\times$ compared to OpenMP default scheme, and $2.3\times$ compared to Hill Climbing on-line adaptation technique. Grzonka et al. [43] show that the ANN can be used to reduce the time required to find the best possible solutions by approximately 30–40%. Grewe et al. [41] show that their neural network is aware of existing workload and can reduce the slowdown to existing workload from 4.5 to 0.5% at a cost of reducing the speedup from $1.66\times$ to $1.59\times$
Support Vector Machines	Grewe and O’Boyle [40] report performance achievement of 80.6% compared to the optimal one. Wang and O’Boyle [98] use ANN and SVM to determine the best number of threads and show performance achievements of up to 96% compared to the optimal performance. Kessler and Löwe [51] show that the SVMs can be used to select the best optimization variant with 0% inaccuracy, however the decision overhead is high
Decision Trees	Castro et al. [15] show performance improvement of up to 18.46% compared to the worst case scenario. Memeti and Pllana [64] can determine a near-optimal workload distribution on heterogeneous system, which results in performance improvement of up to $35.6\times$ compared to sequential version. Thomas et al. [91] show that a performance accuracy between 86 and 100% is capable to dynamically optimize the execution time by choosing the most suitable algorithm in a given context
Regression	Gaussier et al. [36] can predict the execution time, which help to achieve up to 28% makespan reduction. Zhang et al. [103] show performance improvement up to 27% when using regression techniques to predict the optimal number of threads and scheduling policy. Luk et al. [58] use regression techniques to map computations to processing units, which result in performance improvement up to 40% compared to mapping always to CPU, 25% compared to GPU-always, and within 94% of the near optimal mapping
Reinforcement Learning	Eastep et al. [28] reported up to $1.2\times$ speedup compared to other approaches for lock acquisition scheduling. Eastep et al. [29] show the ability to adapt scancount to changing application needs, which result in up to $1.5\times$ speedup compared to state-of-the-art approaches

**Table 11** continued

	Method	Advantages
Meta-heuristics	Simulated Annealing	Memeti and Pllana [63] use simulated annealing to optimize the workload distribution on heterogeneous systems. By evaluating only about 5% of all possible configurations it can achieve average speedup of $1.6\times$ and $2\times$ compared with the host-only and device-only execution
	Genetic Algorithms	Zomaya and Teh [104] show that GA performs better than First Fit for dynamic scheduling using various number of tasks and available processing elements. Page and Naughton [74, 75] show that their evolutionary based scheduler outperforms other schedulers
	Greedy Algorithm	Mantripragada et al. [60] predicts the application execution time, and allows to dynamically shift part of the workload from the cluster to be computed in the cloud, in order to meet the deadline. Albayrak [4] show that nine out of ten times the mapping algorithm based on GrA performs better than single-device mapping
	Hill Climbing	Li et al. [56] shows performance improvement of up to 30% compared to the default configurations used by YARN
	Particle Swarm Optimization	Sivanandam et al. [85] uses PSO and SA for task scheduling. The hybridization of these algorithms outperforms other algorithms, including GA

At run-time, many execution environment parameters influence the performance behavior of the program. Exploring this huge parameter space is time consuming and impractical for programs with long execution times and large demand for system resources. Different computing capabilities and energy efficiency of processing elements of heterogeneous parallel computing systems make the scheduling a difficult challenge. Table 12 lists the limitations of the run-time software optimization approaches for parallel computing systems considered in this paper.

We may observe that some of the existing scheduling techniques often assume that the program is executed on a dedicated system and all system resources are available for use. The approach proposed by Grewe et al. [41] propose a co-scheduling technique, which considers that the resources are shared with other applications. However, the adaptation occurs only when the application is executed, but not during program execution. We believe that better results could be achieved if they consider to adapt to changes while the application is being executed. Another issue is that commonly used scheduling techniques ignore slow processing elements due to their low performance capabilities. Mapping computations always to processing units that offer higher performance capability is not optimal, because slower processing elements may never get work to perform. Furthermore, most of the reviewed approaches target specific features of the code only (for example, loops), or are limited to specific programming models

**Table 12** Limitations of the existing studies that use machine learning and meta-heuristics for run-time software optimization

References	Focus	Limitations
[40, 98]	Mapping computations to the most suitable processing units	The mapping process is dependent on the hardware architecture, which means that a mapping scheme that fits well an architecture may not yield the desired performance on another architecture. It requires to re-learn the prediction model for each new application and architecture configuration
[41, 98, 102, 103]	Determining the optimal number of threads	Require off-line training. Grawe et al. [41] adapts to the workload only when the application starts its execution, but it does not adapt throughout its execution. Zhang et al. [102] focus on applications that consist of single-loops
[15]	Determining thread mapping strategy for TM applications	Since it uses static features, it can not change the mapping strategy when the parallelism degree changes at runtime
[56]	Determining near-optimal system configuration parameters	Requires extensive profiling data collection and analysis, which may introduce significant run-time overhead
[24, 74, 75]	Dynamic task scheduling	The proposed approaches do not consider dependencies between tasks
[80]	Dividing tasks into chunks and scheduling in task-farm way	Task-farm or master-slave like scheduling techniques requires no profiling, however they introduce communication overheads
[2, 104, 105]	Task scheduling	Assume that communication time is known prior execution, processing units have equal computing power and are always ready to perform tasks, and scheduling can be determined off-line and it can not be changed at run-time
[71]	Reducing the number of required training data	This solution requires additional programming investment to build the prediction committee
[4]	Mapping of computation to heterogeneous systems	Collecting profiling information for each kernel on every device represent huge overhead, especially on heterogeneous systems that may comprise larger number of non-identical CPUs and/or GPUs
[91]	Adaptive algorithm selection	Considers hardware characteristics, but not the program input characteristics. Some algorithms perform better for smaller input sizes, whereas others perform better with larger ones
[28, 29, 46, 47, 58, 84]	Self-adaptation	Require adding additional information to the code. Require running the application for a certain amount of time (often with non-optimal parameters) until the framework takes a more optimal decision

and applications (data-bound or compute-bound). Many static scheduling approaches require retraining of the prediction model for each new architecture, limiting their general use because training requires a significant amount of data that is not always available. Approaches that reduce the amount of training data require implementation of multiple machine learning algorithms (for instance, [71]). Approaches that use a single execution [21, 56] by trying various system configurations during the program execution are promising, however the introduced overhead is not negligible. Self-adaptation techniques require the developer to add additional information into the code so that the software would be able to monitor the system and take decisions. Even though such code is not difficult to add for the application programmer, the software development becomes more complex while talking decisions based on these results. Furthermore, such approaches introduce overhead at runtime, because they need to run for a certain amount of time until enough data is collected for the framework to be able to take the most optimal decisions.

Future research should aim at reducing the scheduling and adaption overhead for dynamic approaches. Run-time optimization techniques for heterogeneous systems should be developed that utilize all available computing resources to achieve the optimization goals. There is a need for robust run-time optimization frameworks that are useful for a large spectrum of programs and system architectures. Furthermore, techniques that reduce the amount of data generated from system monitoring are needed in particular for extreme-scale systems.

## 6 Conclusion

In this article, we have conducted a systematic literature review that describes approaches that use machine learning and meta-heuristics for software optimization of parallel computing systems. We have classified approaches based on the software life-cycle activities at compile-time and run-time, including the code optimization and generation, scheduling, and adaptation. We have discussed the shortcomings of existing approaches and provided recommendations for future research directions.

A high-level overview is provided in Table 13, which lists the advantages and limitations of the compile-time and run-time software optimization approaches that use machine learning and meta-heuristics. Our analysis of the reviewed literature suggests that the use of machine learning and meta-heuristic based techniques for software optimization of parallel computing systems is capable of delivering performance comparable to the manual code optimization or task scheduling strategies in specific cases. However, many existing solutions are limited to a specific programming language and model, type of application, or system architecture. There is a need

**Table 13** Advantages and limitations of software optimization approaches that use machine learning and meta-heuristics

	Advantages	Limitations
Compile-time	<ul style="list-style-type: none"> <li>+ Source-to-source code generation and optimization may outperform manually written code</li> <li>+ Selecting the best compiler optimizations results in better performance compared to hard-coded compiler optimizations</li> <li>+ In comparison to techniques that assume that each code block benefits from certain optimizations, performance improvement is observed for approaches that intelligently determine for each code block whether such optimizations result in better performance</li> <li>+ Co-scheduling can reduce the slowdown of other applications</li> </ul>	<ul style="list-style-type: none"> <li>– Some approaches focus on single and simple compiler optimizations. Considering multiple and more complex compiler optimizations is more challenging</li> <li>– Training requires large amount of data or extensive profiling of the code. Lack of training data may result in performance degradation</li> <li>– Some approaches generate multiple versions of the code, and select the most optimal one at runtime. The code-size may dramatically increase</li> <li>– Some of approaches consider only few aspects of the system, such as code features, but not hardware characteristics, system utilization, or task dependencies</li> </ul>
Run-time	<ul style="list-style-type: none"> <li>+ Selecting the best system configurations or algorithm implementation variant for a given execution context may result in better performance and reduce the time required for application tuning</li> <li>+ Adaptation approaches may change the program behavior during execution to meet certain user-defined goals</li> </ul>	<ul style="list-style-type: none"> <li>– Some approaches assume that applications will be executed in isolation, whereas real-world applications may share the same resources with other applications</li> <li>– Some approaches ignore slow processing elements, which may result in overall system underutilization</li> </ul>

for software optimization frameworks that are applicable to a large spectrum of programs and system architectures. Future efforts should focus on developing solutions for widely used general-purpose languages (such as, C/C++) and compilers that are used and supported by the community.

**Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

## References

1. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MF, Thomson J, Toussaint M, Williams CK (2006) Using machine learning to focus iterative optimization. In: Proceedings of the international symposium on code generation and optimization, IEEE Computer Society, pp 295–305
2. Ahmad I, Kwok Y, Ahmad I, Dhadhi M (2001) Scheduling parallel programs using genetic algorithms. Solutions to parallel and distributed computing problems. Wiley, New York, pp 231–254
3. Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: principles, techniques, and tools, 2nd edn. Addison-Wesley Longman Publishing Co., Inc, Boston
4. Albayrak OE, Akturk I, Ozturk O (2013) Improving application behavior on heterogeneous manycore systems through kernel mapping. *Parallel Comput* 39(12):867–878
5. Ansel J, Chan C, Wong YL, Olszewski M, Zhao Q, Edelman A, Amarasinghe S (2009) PetaBricks: a language and compiler for algorithmic choice. ACM, New York
6. Barney B et al (2010) Introduction to parallel computing. Lawrence Livermore National Laboratory, Livermore, p 10
7. Beach TH, Avis NJ (2009) An intelligent semi-automatic application porting system for application accelerators. In: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop, ACM, pp 7–10
8. Beck F, Koch S, Weiskopf D (2016) Visual analysis and dissemination of scientific literature collections with survis. *IEEE Trans Visual Comput Graphics* 22(1):180–189. <https://doi.org/10.1109/TVCG.2015.2467757>
9. Benkner S, Plana S, Träff JL, Tsigas P, Richards A, Namyst R, Bachmayer B, Kessler C, Moloney D, Sanders P (2011) The PEPPHER approach to programmability and performance portability for heterogeneous many-core architectures. In: ParCo
10. Biernacki P, Waldorf D (1981) Snowball sampling: problems and techniques of chain referral sampling. *Sociol Methods Res* 10(2):141–163
11. Binotto APD, Wehrmeister MA, Kuijper A, Pereira CE (2013) Sm@rtConfig: a context-aware runtime and tuning system using an aspect-oriented approach for data intensive engineering applications. *Control Eng Pract* 21(2):204–217
12. Braun TD, Siegel HJ, Beck N, Bölöni LL, Maheswaran M, Reuther AI, Robertson JP, Theys MD, Yao B, Hensgen D et al (2001) A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J Parallel Distrib Comput* 61(6):810–837
13. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Gen Comp Syst* 25(6):599–616. <https://doi.org/10.1016/j.future.2008.12.001>
14. Carretero J, Xhafa F, Abraham A (2007) Genetic algorithm based schedulers for grid computing systems. *Int J Innov Comput Inf Control* 3(6):1–19
15. Castro M, Goes LFW, Ribeiro CP, Cole M, Cintra M, Méhaut JF (2011) A machine learning-based approach for thread mapping on transactional memory applications. In: 2011 18th International conference on high performance computing (HiPC), IEEE, pp 1–10
16. Castro M, Góes LFW, Fernandes LG, Méhaut JF (2012) Dynamic thread mapping based on machine learning for transactional memory applications. In: Euro-Par 2012 Parallel Processing, Springer, pp 465–476
17. Cavazos J, Moss JEB (2004) Inducing heuristics to decide whether to schedule. In: Conference on programming language design and implementation, ACM, New York, NY, USA, PLDI '04, pp 183–194

18. Cavazos J, Fursin G, Agakov F, Bonilla E, Boyle MF, Temam O (2007) Rapidly selecting good compiler optimizations using performance counters. In: International symposium on code generation and optimization, 2007. CGO'07. IEEE, pp 185–197
19. Chen X, Long S (2009) Adaptive multi-versioning for OpenMP parallelization via machine learning. In: 15th International conference on parallel and distributed systems (ICPADS), 2009, IEEE, pp 907–912
20. Chirkov AM, Belloum AS, Kovalchuk SV, Makkes MX, Melnik MA, Visheratin AA, Nasonov DA (2017) Execution time estimation for workflow scheduling. Future generation computer systems. <https://doi.org/10.1016/j.future.2017.01.011>
21. Cooper KD, Grosul A, Harvey TJ, Reeves S, Subramanian D, Torczon L, Waterman T (2005) ACME: adaptive compilation made efficient. In: ACM SIGPLAN notices, ACM 40:69–77
22. Corbalan J, Martorell X, Labarta J (2005) Performance-driven processor allocation. IEEE Trans Parallel Distrib Syst 16(7):599–611
23. Danylenko A, Kessler C, Löwe W (2011) Comparing machine learning approaches for context-aware composition. In: Software composition, Springer, Berlin pp 18–33
24. Diamos GF, Yalamanchili S (2008) Harmony: An execution model and runtime for heterogeneous many core systems. In: Proceedings of the 17th international symposium on high performance distributed Computing, ACM, New York, NY, USA, HPDC '08, pp 197–200. <https://doi.org/10.1145/1383422.1383447>
25. Diefendorff K (1999) Power4 focuses on memory bandwidth. Microprocess. Rep. 13(13):1–8
26. Dongarra J, Sterling T, Simon H, Strohmaier E (2005) High-performance computing: clusters, constellations, mpps, and future directions. Comput. Sci. Eng. 7(2):51–59
27. Duda RO, Hart PE et al (1973) Pattern classification and scene analysis, vol 3. Wiley, New York
28. Eastep J, Wingate D, Santambrogio MD, Agarwal A (2010) Smartlocks: lock acquisition scheduling for self-aware synchronization. In: Proceedings of the 7th international conference on autonomic computing, ACM, pp 215–224
29. Eastep J, Wingate D, Agarwal A (2011) Smart data structures: an online machine learning approach to multicore data structures. In: Proceedings of the 8th international conference on Autonomic computing, ACM, pp 11–20
30. Emani MK, Wang Z, O'Boyle MF (2013) Smart, adaptive mapping of parallelism in the presence of external workload. In: International symposium on code generation and optimization (CGO), IEEE, pp 1–10
31. Fonseca A, Cabral B (2013) *ÆminiumGPU: An Intelligent Framework for GPU Programming*. In: Facing the multicore-challenge III, Springer, pp 96–107
32. Foster I, Kesselman C (2003) The Grid 2: blueprint for a new computing infrastructure. Elsevier, Amsterdam
33. Foster I, Zhao Y, Raicu I, Lu S (2008) Cloud computing and grid computing 360-degree compared. In: 2008 Grid computing environments workshop, pp 1–10. <https://doi.org/10.1109/GCE.2008.4738445>
34. Fursin G, Miranda C, Temam O, Namolaru M, Yom-Tov E, Zaks A, Mendelson B, Bonilla E, Thomson J, Leather H, et al. (2008) MILEPOST GCC: machine learning based research compiler. In: GCC summit
35. Fursin G, Kashnikov Y, Memon AW, Chamski Z, Temam O, Namolaru M, Yom-Tov E, Mendelson B, Zaks A, Courtois E et al (2011) Milepost gcc: machine learning enabled self-tuning compiler. Int J Parallel Prog 39(3):296–327
36. Gaussier E, Glessner D, Reis V, Trystram D (2015) Improving backfilling by using machine learning to predict running times. In: Proceedings of the international conference for high performance computing, networking, storage and analysis, ACM, p 64
37. Geer D (2005) Chip makers turn to multicore processors. Computer 38(5):11–13. <https://doi.org/10.1109/MC.2005.160>
38. Gordon MI, Thies W, Amarasinghe S (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ACM SIGOPS Operating Systems Review, ACM 40:151–162
39. Gould N (2006) An introduction to algorithms for continuous optimization. Oxford University Computing Laboratory Notes
40. Grewe D, OBoyle MF (2011) A static task partitioning approach for heterogeneous systems using opencl. In: Compiler Construction, Springer, Berlin, pp 286–305

41. Grewe D, Wang Z, O'Boyle MF (2011) A workload-aware mapping approach for data-parallel programs. In: Proceedings of the 6th international conference on high performance and embedded architectures and compilers, ACM, pp 117–126
42. Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message-passing interface, vol 1. MIT press, Cambridge
43. Grzonka D, Kolodziej J, Tao J (2014) Using artificial neural network for monitoring and supporting the grid scheduler performance. In: ECMS, pp 515–522
44. Grzonka D, Jakbik A, KoÅodziej J, Pllana S (2017) Using a multi-agent system and artificial intelligence for monitoring and improving the cloud performance and security. Future Generation Computer Systems. 10.1016/j.future.2017.05.046, <http://www.sciencedirect.com/science/article/pii/S016739X17310531>
45. Guyon I, Elisseeff A (2003) An introduction to variable and feature selection. J Mach Learn Res 3:1157–1182
46. Hoffmann H, Eastep J, Santambrogio MD, Miller JE, Agarwal A (2010a) Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In: Parashar M, Figueiredo RJO, Kiciman E (eds) ICAC. ACM, New York City
47. Hoffmann H, Maggio M, Santambrogio MD, Leva A, Agarwal A (2010b) SEEC: A framework for self-aware computing. <http://hdl.handle.net/1721.1/59519>
48. Iakymchuk R, Jordan H, Bo Peng I, Markidis S, Laure E (2016) A particle-in-cell method for automatic load-balancing with the allscale environment. In: The Exascale applications & Software conference (EASC2016)
49. Jeffers J, Reinders J (2015) High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches. Morgan Kaufmann, Burlington
50. Jin C, de Supinski BR, Abramson D, Poxon H, DeRose L, Dinh MN, Endrei M, Jessup ER (2016) A survey on software methods to improve the energy efficiency of parallel computing. In: The international journal of high performance computing applications p 1094342016665471. <https://doi.org/10.1177/1094342016665471>
51. Kessler C, Löwe W (2012) Optimized composition of performance-aware parallel components. Concurr Comput Pract Exp 24(5):481–498
52. Kessler C, Dastgeer U, Thibault S, Namyst R, Richards A, Dolinsky U, Benkner S, Träff JL, Pllana S (2012) Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In: Design, automation & test in Europe conference & exhibition (DATE), 2012, IEEE, pp 1403–1408
53. Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. In: Technical Report EBSE 2007-001, Keele University and Durham University Joint Report
54. Lee BD, Schopf JM (2003) Run-time prediction of parallel applications on shared environments. In: IEEE International conference on cluster computing, 2003. Proceedings. 2003, IEEE, pp 487–491
55. Li L, Dastgeer U, Kessler C (2012) Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In: High performance computing for computational science-VECPAR 2012, Springer, pp 329–345
56. Li M, Zeng L, Meng S, Tan J, Zhang L, Butt AR, Fuller N (2014) Mronline: Mapreduce online performance tuning. In: Proceedings of the 23rd international symposium on High-performance parallel and distributed computing, ACM, pp 165–176
57. Liu B, Zhao Y, Zhong X, Liang Z, Feng B (2013) A Novel Thread Partitioning Approach Based on Machine Learning for Speculative Multithreading. In: IEEE international conference on embedded and ubiquitous computing high performance computing and communications & 2013 (HPCC\_EUC), 2013 IEEE 10th International Conference on, IEEE, pp 826–836
58. Luk CK, Hong S, Kim H (2009) Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture, ACM, New York, NY, USA, MICRO 42, pp 45–55. <https://doi.org/10.1145/1669112.1669121>
59. Malawski M, Juve G, Deelman E, Nabrzyski J (2015) Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. Future Generation Computer Systems 48:1–18. <https://doi.org/10.1016/j.future.2015.01.004>, special Section: Business and Industry Specific Cloud

60. Mantripragada K, Binotto APD, Tizzei LP (2014) A self-adaptive auto-scaling method for scientific applications on HPC environments and clouds. CoRR abs/1412.6392
61. Mastelic T, Fdhila W, Brandic I, Rinderle-Ma S (2015) Predicting resource allocation and costs for business processes in the cloud. In: 2015 IEEE world congress on services, pp 47–54. <https://doi.org/10.1109/SERVICES.2015.16>
62. Memeti S, Pllana S (2016a) Combinatorial optimization of dna sequence analysis on heterogeneous systems. Concurrency and computation: practice and experience pp n/a-n/a. <https://doi.org/10.1002/cpe.4037>, cpe.4037
63. Memeti S, Pllana S (2016b) Combinatorial optimization of work distribution on heterogeneous systems. In: 2016 45th international conference on parallel processing workshops (ICPPW), pp 151–160. <https://doi.org/10.1109/ICPPW.2016.35>
64. Memeti S, Pllana S (2016c) A machine learning approach for accelerating dna sequence analysis. The International Journal of High Performance Computing Applications 0(0):1094342016654,214. <https://doi.org/10.1177/1094342016654214>
65. Memeti S, Pllana S, Kołodziej J (2016) Optimal worksharing of DNA sequence analysis on accelerated platforms. Springer, Cham, pp 279–309. [https://doi.org/10.1007/978-3-319-44881-7\\_14](https://doi.org/10.1007/978-3-319-44881-7_14)
66. Memeti S, Li L, Pllana S, Kolodziej J, Kessler C (2017) Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption. In: Proceedings of the 2017 workshop on adaptive resource management and scheduling for cloud computing, ACM, New York, NY, USA, ARMS-CC '17, pp 1–6. <https://doi.org/10.1145/3110355.3110356>
67. Mitchell TM (1997) Machine learning, 1st edn. McGraw-Hill Inc, New York, NY, USA
68. Mittal S, Vetter JS (2015) A survey of cpu-gpu heterogeneous computing techniques. ACM Comput Surv (CSUR) 47(4):69
69. Monsifrot A, Bodin F, Quiniou R (2002) A machine learning approach to automatic production of compiler heuristics. In: Artificial intelligence: methodology, systems, and applications, Springer, pp 41–50
70. Nvidia C (2015) CUDA C programming guide. NVIDIA Corp 120:18
71. Ogilvie W, Petoumenos P, Wang Z, Leather H (2015) Intelligent heuristic construction with active learning. In: Compilers for parallel computing (CPC'15). London, United Kingdom
72. OpenMP A (2013) OpenMP 4.0 specification, June 2013
73. Padua D (2011) Encyclopedia of parallel computing. Springer, Berlin
74. Page AJ, Naughton TJ (2005a) Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In: 19th International parallel and distributed processing symposium, IEEE, pp 189a–189a
75. Page AJ, Naughton TJ (2005b) Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. Artif Intell Rev 24(3):415–429. <https://doi.org/10.1007/s10462-005-9002-x>
76. Park Yw, Baskiyar S, Casey K (2010) A novel adaptive support vector machine based task scheduling. In: Proceedings the 9th International Conference on Parallel and Distributed Computing and Networks, Austria, pp 16–18
77. Pekhimenko G, Brown AD (2011) Efficient program compilation through machine learning techniques. In: Software Automatic Tuning, Springer, pp 335–351
78. Pllana S, Benkner S, Mehofer E, Natvig L, Xhafa F (2008) Towards an intelligent environment for programming multi-core computing systems. Euro-Par Workshops, Springer, Lecture Notes in Computer Science 5415:141–151
79. Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2007) Numerical recipes 3rd edition: the art of scientific computing, 3rd edn. Cambridge University Press, Cambridge
80. Ravi VT, Agrawal G (2011) A dynamic scheduling framework for emerging heterogeneous systems. In: 18th International conference on high performance computing (HiPC), 2011, IEEE, pp 1–10
81. Rossbach CJ, Yu Y, Currey J, Martin JP, Fetterly D (2013) Dandelion: a compiler and runtime for heterogeneous systems. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles, ACM, pp 49–68
82. Sadashiv N, Kumar SMD (2011) Cluster, grid and cloud computing: A detailed comparison. In: 2011 6th International conference on computer science education (ICCSE), pp 477–482. <https://doi.org/10.1109/ICCSE.2011.6028683>
83. Sandrieser M, Benkner S, Pllana S (2012) Using explicit platform descriptions to support programming of heterogeneous many-core systems. Parallel Comput 38(1–2):52–56

84. Silvano C, Agosta G, Cherubin S, Gadioli D, Palermo G, Bartolini A, Benini L, Martinović J, Palković M, Slaninová K, Bispo Ja, Cardoso JaMP, Abreu R, Pinto P, Cavazzoni C, Sanna N, Beccari AR, Cmar R, Rohou E (2016) The antarex approach to autotuning and adaptivity for energy efficient hpc systems. In: Proceedings of the international conference on computing frontiers, ACM, New York, NY, USA, CF '16, pp 288–293. <https://doi.org/10.1145/2903150.2903470>
85. Sivanandam SN, Visalakshi P (2009) Dynamic task scheduling with load balancing using parallel orthogonal particle swarm optimisation. Int J Bio-Inspired Comput 1(4):276–286. <https://doi.org/10.1504/IJBIC.2009.024726>
86. Smachan S, Indrawan M, Ling S, Enticott C, Abramson D (2013) Scheduling parameter sweep workflow in the grid based on resource competition. Future Gen Comput Syst 29(5):1164–1183. <https://doi.org/10.1016/j.future.2013.01.005>
87. Stephenson M, Amarasinghe S (2005) Predicting unroll factors using supervised classification. In: International Symposium on code generation and optimization, 2005. CGO 2005, IEEE, pp 123–134
88. Stephenson M, Amarasinghe S, Martin M, O'Reilly UM (2003) Meta optimization: improving compiler heuristics with machine learning. SIGPLAN Not 38(5):77–90
89. Sterling T, Becker DJ, Savarese D, Dorband JE, Ranawake UA, Packer CV (1995) Beowulf: A parallel workstation for scientific computation. In: Proceedings of the 24th international conference on parallel processing, pp 11–14
90. Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12(1–3):66–73
91. Thomas N, Tanase G, Tkachyshyn O, Perdue J, Amato NM, Rauchwerger L (2005) A framework for adaptive algorithm selection in STAPL. In: Proceedings of the tenth ACM SIGPLAN symposium on principles and practice of parallel programming, ACM, pp 277–288
92. Tiwari A, Hollingsworth JK (2011) Online adaptive code generation and tuning. In: 2011 IEEE international parallel distributed processing symposium, pp 879–892. <https://doi.org/10.1109/IPDPS.2011.86>
93. Tiwari A, Chen C, Chame J, Hall M, Hollingsworth JK (2009) A scalable auto-tuning framework for compiler optimization. In: Proceedings of the 2009 IEEE international symposium on parallel& distributed processing, IEEE Computer Society, Washington, DC, USA, IPDPS '09, pp 1–12. <https://doi.org/10.1109/IPDPS.2009.5161054>
94. TOP500 (2016) TOP500 Supercomputer Sites. <http://www.top500.org/>. Accessed Jan 2016
95. Tournavitis G, Wang Z, Franke B, O'Boyle MF (2009) Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: ACM Sigplan notices 44:177–187
96. Viebke A, Plana S (2015) The potential of the intel (r) xeon phi for supervised deep learning. In: 2015 IEEE 17th international conference on high performance computing and communications (HPCC), pp 758–765. <https://doi.org/10.1109/HPCC-CSS-ICESS.2015.45>
97. Voss M, Kim W (2011) Multicore desktop programming with intel threading building blocks. IEEE Softw 28(1):23–31. <https://doi.org/10.1109/MS.2011.12>
98. Wang Z, O'Boyle MF (2009) Mapping parallelism to multi-cores: a machine learning based approach. In: ACM Sigplan notices, ACM 44:75–84
99. Wang Z, O'boyle MF (2013) Using machine learning to partition streaming programs. ACM Trans Archit Code Optim (TACO) 10(3):20
100. Wienke S, Springer P, Terboven C, an Mey D (2012) Openacc: First experiences with real-world applications. In: Proceedings of the 18th international conference on parallel processing, Springer-Verlag, Berlin, Heidelberg, Euro-Par'12, pp 859–870
101. Wolsey LA, Nemhauser GL (2014) Integer and combinatorial optimization. Wiley, Hoboken
102. Zhang Y, Burcea M, Cheng V, Ho R, Voss M (2004) An adaptive openmp loop scheduler for hyper-threaded smps. In: ISCA PDCS, pp 256–263
103. Zhang Y, Voss M, Rogers E (2005) Runtime empirical selection of loop schedulers on hyperthreaded smps. In: Proceedings of 19th IEEE International parallel and distributed processing symposium, 2005, IEEE, pp 44b–44b
104. Zomaya AY, Teh YH (2001) Observations on using genetic algorithms for dynamic load-balancing. IEEE Trans Parallel Distrib Syst 12(9):899–911
105. Zomaya AY, Lee RC, Olariu S (2001) An introduction to genetic-based scheduling in parallel processor systems. Solutions to Parallel and Distributed Computing Problems pp 111–133