Optimization of Parallel Software Tuning with Statistical Modeling and Machine Learning

Anatoliy Doroshenko, Pavlo Ivanenko, Oleksandr Novak, Olena Yatsenko

Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov prosp. 40, 03187 Kyiv, Ukraine
doroshenkoanatoliy2@gmail.com, paiv@ukr.net, oayat@ukr.net

Abstract. High-performance computation is the main goal of parallel computers, but the performance of compiled code is often far from the best. Parallel program auto-tuning is the method adjusting some structural parameters (mainly, data structures) of an application program for a target hardware platform to speed-up computation as much as possible. In previous work, the authors have developed a framework intended to automate generation of an auto-tuner from an application source code. However, auto-tuning for complex and nontrivial parallel systems is usually time-consuming due to empirical evaluation of huge amount of parameter values combinations of an initial parallel program in a target environment. In this paper, we propose to improve the auto-tuning method using statistical modeling and neural network algorithms that allow to reduce significantly the space of possible parameter combinations. The resulting optimization is illustrated by an example of tuning a parallel sorting program, that combines several sorting methods. The optimization is done by means of the automatic training of a neural network model on results of "traditional" tuning cycles with subsequent replacement of some auto-tuner calls with an evaluation from the statistical model.

Keywords. Auto-tuning, parallel computation, machine learning, neural network, statistical modeling.

1 Introduction

The problem of optimal use of computing resources has always been important in the process of development of any software — from mobile applications to complex client-server systems. The auto-tuning paradigm [1, 2], which has become a standard for solving the problem of software application optimization over the last decade, allows to fully automatize this process for any computing environment. Its popularity is predefined first by simplicity of use and independence from qualitative characteristics of a computer and operating system. Auto-tuning traditionally uses empirical data for obtaining a qualitative evaluation of optimized code (the quality usually refers to program execution time and accuracy of output results). It automates the search for the optimal program version out of a set of provided possibilities by running each candidate and measuring its performance on a given parallel architecture. Its main

benefit is a high level of abstraction — a program is optimized without explicit knowledge of hardware implementation details, such as number of cores, cache size or memory access speed on various levels. Instead, it needs to use subject domain concepts such as number and size of independent tasks.

In the previous works [3–6], we have developed a theory, methodology and tools for automated program design, synthesis, and auto-tuning, based on Glushkov's systems of algorithmic algebras (SAA) and term rewriting technique. The model for parallel programs optimization and the auto-tuning framework named TuningGenie aimed at automating adjustment of programs to a target platform have been proposed in [6]. The framework works with a source code of parallel software and performs source-to-source transformations by using facilities of a rule-based rewriting system TermWare [3].

The main drawback of the auto-tuning approach is in significant one-time costs of optimization process: if the number of program versions is large enough, the optimization process may run for many hours and even days. In this paper we propose the hybrid approach to auto-tuning using statistical modeling and machine learning technique to reduce the time needed for searching for an optimal program version. The approach consists in automatic training of a neural network model on results of common tuning cycles with subsequent replacement of some auto-tuner calls with an evaluation from the statistical model.

2 Auto-Tuning Software Framework and Machine Learning

In the work [6], TuningGenie framework for automated generation of auto-tuner applications from a source code has been developed. The idea of an auto-tuner consists in empirical evaluation of several versions of input program and selection of the best one where the main evaluation criteria are less execution time of input program and accuracy of results obtained. The framework works with program source code using expert knowledge of a developer and automation facilities from the framework. A developer adds some metadata (parameter names and value ranges) to a source code in the form of special comments-pragmas. Exploiting such expert knowledge (s)he can reduce the number of program versions to be evaluated and therefore increase optimization performance.

The auto-tuning software implementation is based on the rewriting rules system TermWare [3]. TermWare is an open-source implementation of rewriting rules engine written in Java. It provides a language for describing rewriting rules that operate on data structures that are called terms, and a rule engine that interprets rules to transform terms. TuningGenie uses TermWare to extract expert knowledge from program source code and generates a new program version on each tuning iteration. TermWare translates source code into a term and provides transformations according to rewriting rules. The current TermWare version contains components for interaction with Java and C# languages, and the current TuningGenie version supports Java programs.

Application of auto-tuning for complex and nontrivial program systems usually takes a lot of time due to empirical estimating a large number of parameter combina-

tions of input program in a target environment (let us denote the set of parameters combinations as C). In this paper we propose to optimize the auto-tuning method by using statistical modeling and machine learning. The improvement consists in reducing the number of auto-tuner launches by means of building an approximation model which allows dismissing the parameter combinations that are unlikely to be fast. The model approximation often results in a reduction of dimensionality of input parameters of the set C that means significant auto-tuning process speed-up.

Generally, machine learning methods are based on the concept of learning some behavior from data [2, 7]. In the context of auto-tuning, the behavior to be learnt, for example, can be program performance at different settings of program parameters. A machine learning method first evaluates several alternatives within the search space for n different input programs $P_1,...,P_n$, defined by configurations $C_1,...,C_n$. The set of evaluated alternatives is called training data. The process of generating and evaluating the training data and learning behavior from this data is called training. Once the training is completed, and given a new version of program P' to be evaluated, execution of P' is replaced with estimate, obtained from trained model.

Machine learning is closely linked to (and often overlaps with) computational statistics [8]. All statistical algorithms (including machine learning algorithms) require a significant number of statistical data for analysis and model construction. In the context of auto-tuning tasks, the collection of many statistical data can be a long process. Therefore, the problem of selecting the algorithms narrowing the search space at a minimal number of real launches of an auto-tuner is very acute. For a partial solution of the mentioned problem, in this work we use a neural network for data extrapolation (see Section 3). In this case, relatively small number of real launches is required for construction of an approximate model, after which the neural network model can be used by other algorithms according to the black box principle.

3 A Case Study

In the design process, we follow top-down formal transformational style provided by our automated toolkit for designing and synthesis of programs (IDS) [4, 5]. We begin with high-level specification presented as a generalized scheme of the algorithm represented in the algorithmic language of Glushkov's algorithmic algebras [4] that has the advantage to be human-friendly and complete with code in one of the parallel programming languages (Java or C++, in our case).

Below, we consider a case study of performance tuning by the example of a hybrid parallel sorting algorithm which applies a merge sort or an insertion sort depending on a block size (`insertionSortThreshold`) of input numerical array. The initial SAA scheme of the algorithm contains the `tuneAbleParam` pragmas, which specify search domain for optimal values of variables `insertionSortThreshold` and `mergeSortBucketSize`. The resulting algorithm is implemented in Java.

```
"Parallel Hybrid Sorting (arr)"
```

```
==== "Comment(tuneAbleParam name=insertionSortThreshold
              start=10 stop=200 step=10)";
     "Declare a variable (insertionSortThreshold)
      of type (int) with initial value (100)";
     "Comment(tuneAbleParam name=mergeSortBucketSize
              start=5000 stop=1000000 step=5000)";
     "Declare a variable (mergeSortBucketSize)
      of type (int) with initial value (5000)";

     IF 'Length of the array (arr) is less or equal to
         (insertionSortThreshold)'
     THEN "insertionSort(arr)"
     ELSE IF 'Length of the array (arr) is less or equal
              to (mergeSortBucketSize)'
          THEN "sequentialMergeSort(arr)"
          ELSE "concurrentMergeSort(arr)"
          END IF
     END IF
```

In the auto-tuning experiment, the set of 2×10^7 random integer numbers were sorted. The auto-tuner parameters are $C = \{T_{cn}, T_s, T_h\}$, where T_{cn} is a number of parallel threads, T_s is a threshold for block size to be sorted sequentially within the current thread (blocks with $size > T_s$ are split into smaller blocks and assigned to different threads), T_h is a block size at which insertion sort is used.

The experiment was performed in the following environment: 2.7 GHz Intel Core i7 processor (6820HQ) with 4 cores and 8 MB L3 cache; 16 GB 2133 MHz RAM; 512 GB Apple SSD SM0512L; MacOS 10.12.

In a first phase, the auto-tuner was executed without a statistical model to estimate how quick the tuned algorithm can be. In a second phase, the statistical modeling was plugged in to understand how heavily the search space can be pruned while preserving the near-optimum performance of the tuned algorithm.

Let's look at the results of the first phase given in Table 1. Three configurations are listed: *slow* ("default" configuration that behaves almost as classical sequential merge sort); *optimal* (the quickest one that was automatically picked by the auto-tuner) and *intuitive* (values are filled in by intuition with respect to known hardware specifications and algorithms details). *Optimal* configuration is 4.93 times quicker than *slow*. This result is quite good for 4-core processor and was achieved primarily by a combination of two factors: optimal usage of processor caches (by switching to in-place sorting for small data sets) and efficient parallelization schema (merge sort is easy to parallelize with "divide and conquer" method). *Intuitive* combination was 3.1 times faster than *slow* — also a decent result, but it was easy to guess due to relative simplicity of the test algorithm. Usually optimal configurations are not so obvious for real-life parallel programs. *Optimal* configuration is still substantially quicker — by 58%, so we can say that it was worth the time spent on tuning.

Table 1. The results of the first auto-tuning phase.

Configuration	***slow***	***optimal***	***intuitive***
Parallelism level T_{cn}	1 (one thread)	8	4
Insertion sort threshold T_h	0 (do not switch to insertion sort at all)	120	30 (common notion is to set couple dozen as a threshold for this trick)
Threshold for sequential sorting T_s	100 000 000 (it's bigger than the test data size, so no data decomposition is applied)	50 000	10 000
Test data size	20 000 000 integers		
Average sorting time	4432 ms	898 ms	1426 ms

Now let's move to the second phase to see how the auto-tuner's search space can be reduced with the help of statistical analysis methods. T_s parameter is excluded from the model during primary analysis phase because of its minor impact on overall performance: once the number of subtasks after the decomposition of input data is couple times bigger than the parallelism level, it makes almost no difference what value is used. This can be explained by high effectiveness of Java's *RecursiveAction* [9] mechanism that was used in the implementation. *RecursiveAction* is a recursive *ForkJoinTask*, which is "a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a ForkJoinPool, at the price of some usage limitations" [10]. The experiment proved that the computational overhead on executing new *RecursiveAction* is negligible.

The primary analysis of data was performed in Python language with a help of Scikit-learn library [11]. Further analysis was implemented by means of R [12], which is a programming language for statistical computations, analysis and graphical representation of data. The experiment consisted of several stages: preparation and loading of auto-tuner results to R environment, data preparation (including normalization), building a neural network model on a training dataset and checking the model on a test dataset.

The data analysis process is shown in Figure 1. At first, the auto-tuner performs N experiments and saves the result data to a separate file. The data is used by the neural network for training. After the training, the neural network extrapolates the data, generates the new dataset, which is written into a separate file. In the end, both datasets are analyzed and compared by a human. As a neural network, a multilayer perceptron with three input neurons, three hidden layers (20-10-5 neurons per layer) and one output neuron were applied. The rectified linear function $f(x) = \max(0, x)$ was used as an activation function. The backward propagation of errors has been used as a machine learning method and the Broyden-Fletcher-Goldfarb-Shanno algorithm [13] has been applied for optimization of weighting factors.

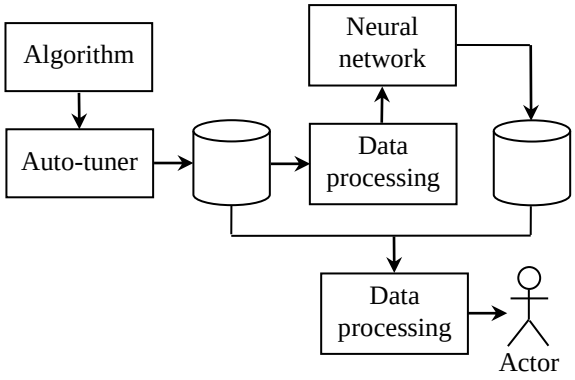

Fig. 1. The process of analysis

The initial neural network was built based on results of 3300 launches. Then it was used for further data generation. The use of the neural network for initial approximation allowed to reduce the search region by 58% (from 10^6 to 4.2×10^5). For estimating the quality of the obtained results, more than 30000 real launches (evenly distributed over the combinations set) of the auto-tuner was performed.

Figure 2 shows the dependency of the model accuracy Acc from 10 neural networks on the ratio of sample data used for training.

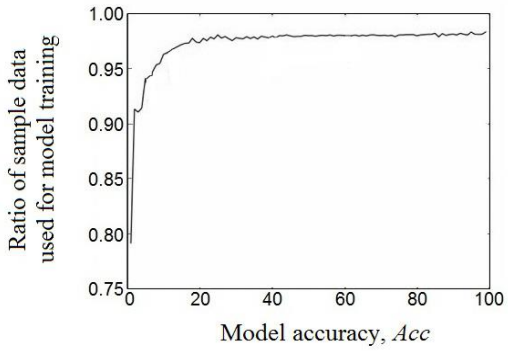

Fig. 2. The dependency of the model accuracy Acc on the ratio of sample data used for model training

The evaluation of the accuracy Acc is based on a confusion matrix [14] and is calculated according to the formula $Acc = \frac{TP + TN}{P + N}$, where TP is the number of true positives; TN is the number of true negatives; P is the number of real positive cases in the data; N is the number of real negative cases.

4 Related Work

Many approaches have been proposed for the problem of auto-tuner development. Well-known examples of auto-tuners are ATLAS [15] and FFTW [16], which are specialized libraries introducing high-performance implementation of some specific functions. Unlike our TuningGenie framework, which provides domain independent optimization, they are tied to domain and language. TuningGenie is quite similar to Atune-IL [17], a language extension for auto-tuning. It also uses pragmas and is not tied to some specific programming language. The main difference of TuningGenie is due to term rewriting engine that is used for source code transformation. Representing program code as a term allows modifying program structure in a declarative way. This feature significantly increases the capabilities of the auto-tuning framework.

There are also auto-tuners based on machine learning techniques [2]. In paper [18], an open-source self-tuning compiler Milepost GCC is described, which exploits machine learning to predict optimal setting of compilation flags for a program at using GCC. In [19] neural networks are used to learn the behavior of a given program transformation (parametric loop tiling) for different values of input parameter (tile size); the model is then used to search for optimal parameter values. In the work [20], a machine learning approach is applied for automatic optimization of task partitioning for OpenCL for different input problem sizes and different heterogeneous architectures consisting of CPUs and GPUs. In our work, we use neural networks for learning on the results of tuning cycles (program execution time at different values of internal program parameters) with subsequent replacement of some auto-tuner calls with an evaluation from the model.

Conclusion

In this paper, we explore the promising method of software auto-tuning improved by using statistical modeling and neural networks. The method allows substantially get rid of the main weakness of the auto-tuning methodology, namely, significantly accelerate the search for an optimal program version by automatic training a neural network model on the results of regular tuning cycles and subsequent replacement of some auto-tuner calls with an evaluation from the model. Furthermore, the use of a perceptron at the primary analysis stage helps to identify the most important input parameters (i.e. which have the largest influence on a final result). The approach is illustrated by the example of performance tuning of a hybrid parallel sorting program that exploits the developed earlier TuningGenie framework. The results of the experiment confirmed the efficiency of the proposed approach and the usefulness of its further development, in particular, the use of more complex approximation functions and conducting experiments with more computationally and semantically complex programs.

References

1. Naono, K., Teranishi, K., Cavazos, J., Suda, R.: Software automatic tuning: from concepts to state-of-the-art results. Springer, Berlin (2010)
2. Durillo, J., Fahringer, T.: From single- to multi-objective auto-tuning of programs: advantages and implications. Scientific Programming, 22(4), 285–297 (2014)
3. Doroshenko, A., Shevchenko, R.: A rewriting framework for rule-based programming dynamic applications. Fundamenta Informaticae, 72(1–3), 95–108 (2006)
4. Andon, F.I., Doroshenko, A.Y., Tseytlin, G.O., Yatsenko, O.A.: Algebra-algorithmic models and methods of parallel programming. Akademperiodyka, Kyiv (2007) (in Russian)
5. Doroshenko, A., Zhereb, K., Yatsenko, O.: Developing and optimizing parallel programs with algebra-algorithmic and term rewriting tools. In: Ermolayev, V., Mayr, H.C., Nikitchenko, M., Spivakovsky, A., Zholtkevych, G. (eds.) ICTERI 2013, Communications in Computer and Information Science, vol. 412, pp. 70–92. Springer, Cham (2013)
6. Ivanenko, P., Doroshenko, A., Zhereb, K.: TuningGenie: auto-tuning framework based on rewriting rules. In: Ermolayev, V., Mayr, H., Nikitchenko, M., Spivakovsky, A., Zholtkevych, G. (eds.) ICTERI 2014. Communications in Computer and Information Science, vol. 469, pp. 139–158. Springer, Cham (2014)
7. Mitchell, T.M.: Machine learning. 1st edn. McGraw-Hill Education, New York (1997)
8. Givens, G.H., Hoeting, J.A.: Computational statistics. 2nd edn. Wiley, Chichester (2012)
9. Class RecursiveAction (Java SE 9 & JDK 9) – Oracle Help Center, https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/RecursiveAction.html, last accessed 2017/12/20.
10. Class ForkJoinTask (Java SE 9 & JDK 9) – Oracle Help Center, https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ForkJoinTask.html, last accessed 2017/12/20.
11. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V. et al.: Scikit-learn: machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830 (2011)
12. Crawley, M.J.: The R book. 1st edn. Wiley, Chichester (2012)
13. Fletcher, R.: Practical methods of optimization. 2nd edn. Wiley, Chichester (2000)
14. Fawcett, T.: An introduction to ROC analysis. Pattern Recognition Letters, 27(8), 861–874 (2006)
15. Whaley, R., Petitet, A., Dongarra, J. J.: Automated empirical optimizations of software and the ATLAS Project. Parallel Computing, 27(1–2), 3–35 (2001)
16. Frigo, M., Johnson, S.: FFTW: an adaptive software architecture for the FF. Acoustics, Speech and Signal Processing, 3, 1381–1384 (1998)
17. Schaefer, C.A., Pankratius, V., Tichy, W.F.: Atune-IL: an instrumentation language for auto-tuning parallel applications. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par'2009. LNCS, vol. 5704, pp. 9–20. Springer, Berlin, Heidelberg (2009)
18. Fursin, G., Kashnikov, Y., Memon, A.W., Chamski, Z. et al.: Milepost GCC: machine learning enabled self-tuning compiler. International Journal of Parallel Programming 39(3), 296–327 (2011)
19. Rahman, M., Pouchet, L.-N., Sadayappan, P.: Neural network assisted tile size selection. In: 5th International Workshop on Automatic Performance Tuning (IWAPT'2010). Springer, Berkeley, CA (2010)
20. Kofler, K., Grasso, I., Cosenza, B., Fahringer, T.: An automatic input-sensitive approach for heterogeneous task partitioning. In: 27th ACM International Conference on Supercomputing (ICS'13), pp. 149–160. ACM, New York (2013)