Application of rule rewriting system to software automated tuning

Ivanenko P.A., Doroshenko A.Y.

Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov av. 40, 03187 Kyiv, Ukraine

paiv@ukr.net, doroshenkoanatoliy2@gmail.com

Abstract. *This paper presents a technical report on development of auto-tuning framework TuningGenie for parallel programs performance acceleration. The framework works with source code of software and performs code-to-code transformations by utilizing facilities of rule-based rewriting system. Such approach gains higher flexibility comparing to existing solutions. This paper explains the framework's lifecycle, offers toolset for optimization, basics of architecture, and presents demo examples with results of tuning computationally complex parallel program.*

Keywords

Software automated tuning, rule rewriting systems, parallel software optimization.

1 Introduction

Current and emerging scientific and industrial applications require significant computation power provided by parallel platforms such as multicore, clusters, Grids, cloud computing and GPGPU. Therefore, optimizing application code for a given parallel platform is a significant part of development process. The problem is that powerful computation platform does not guarantee application performance: developer effort is required to achieve it.

Programming efficient algorithm has always been a challenging task. It is made even harder by widespread use of multicore processors. The performance growth of a multithreaded program is limited by Amdahl's law [1], so efficient programs should minimize the fraction of sequential computation as well as synchronization and communication overhead. Such optimizations are usually hardware platform-dependent, and the program that was optimized for one problem is likely not optimal on another platform. Therefore there is a need of tools that could perform such adaptation to concrete platform in automated mode.

One traditional solution of this problem is provided by parallelizing compilers [2-3]. The quality of automated parallelization has increased recently; still, such tools are only applicable to programs with simple structure of computation. The reasons include the complexity of static analysis of large codebases, increasing complexity of parallel architectures, as well as their diversity.

An alternative to parallelizing compilers is provided by auto-tuning [4] – a method that has already proven its efficiency and versatility. It can automate the search for the optimal program variant out of set of provided possibilities, by running each candidate and measuring its performance on a given parallel architecture. Its main benefit is high level of abstraction – 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, the parallel program uses subject domain concepts such as number and size of independent tasks, or algorithm details such as data traversal methods. The drawbacks of auto-tuning approach include significant one-time costs of optimization process itself: if the number of program variants is large enough, the optimization process may run for many hours and even days. Also there is an additional development overhead – creation of auto-tuner application.

This paper considers a universal framework for automated generation of auto-tuner applications from source code. Proposed approach solves many of the described drawbacks and is versatile enough to be applied in any subject domain.

2 Framework overview

TuningGenie framework optimizes programs by generating a stand-alone tuner application. This approach allows encapsulating the optimization logic in auto-tuner, simplifying development and maintenance for the target application. To avoid the auto-tuning problems described in Section 1, the framework works with program source code using expert knowledge of a developer and automation facilities from the framework. Developers add metadata to the source code in form of special comments-pragmas or Java annotations. Such expert knowledge reduces number of program variants to be evaluated and therefore increases optimization speed. Also such approach allows TuningGenie to provide domain-independent optimization, unlike many existing specialized tools like ATLAS [5] and FFTW [6].

Our TuningGenie framework is quite similar to Atune-IL [7] – a language extension for auto-tuning. 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 auto-tuning framework.

3 Tuning lifecycle

The software implementation of the auto-tuning system is based on rewriting rules system TermWare[8-9]. TermWare implements a concept of terminal systems, i.e. object structures and rewriting rules with explicit actions that interact with the knowledge base. Thanks to this system, TuningGenie can extract expert knowledge from program source code and generate a new version of program on each tuning iteration. TermWare translates source code into a term and provides transformation tools based on rewriting rules. Therefore TuningGenie can perform structural changes in program computations using declarative style (without explicit changes to source code). This property distinguishes TunigGenie from similar systems, such as AtuneIL[7]; it is quite useful when there is a need to optimize large existing parallel programs.

Current version of TermWare contains components for interaction with Java and C# languages. To support other languages, a parser to translate source code into TermWare language and a pretty-printer should be implemented. Current version of TuningGenie supports Java programs.

The common workflow of the TuningGenie framework is shown on Fig. 1.

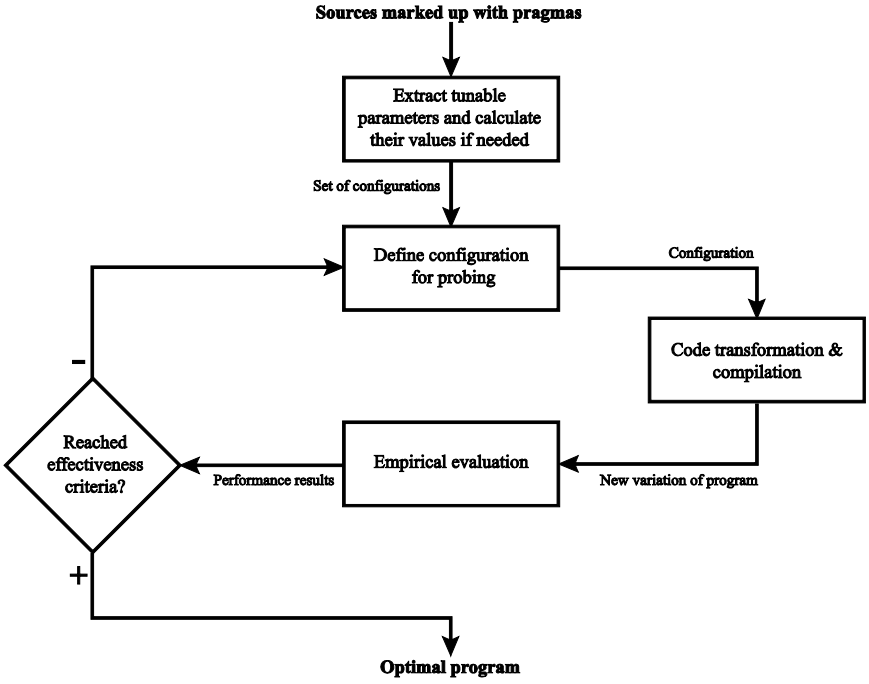

Fig. 1. Tuning workflow

TuningGenie accepts as an input Java source code marked with so-called "pragmas". Pragmas describe configurations and transformations of program that affect its performance. Pragmas are specified manually by program developers, using Java comments of a special form.

During the parsing of source code into a term, the auto-tuner builds a set of configurations based on these expert data. Then these configurations are translated into rewriting rules. Also on this "preliminary" stage some values of program parameters are calculated (see the "calculatedValue" pragma in Section 4). The results of this stage include a

program term, a set of parameterized rewriting rules and a set of rule configurations C that specify the concrete parameter values. Each of these configurations specifies a unique variant of input program.

Then TuningGenie searches for the most efficient configuration $C_{opt} \in C$, by iteratively performing the following steps:

1. Select a configuration to test $C^* \in C$
2. Generate and compile the corresponding program variant. The parameters from a given configuration are substituted into the rules; then these rules are executed on the source term of the parallel program. After the transformation is complete, the term is transformed into Java source code using TermWare facilities. The code is compiled using common JDK tools. The result of this step is a new version of the program ready for performance measurement.
3. Execute the program and evaluate its performance. A small launcher class is generated automatically. It runs the program and measures its execution time, therefore computing $f(C^*) = t$
4. If all configurations from the set C have been evaluated or if the optimization process has used up all the allotted time – go to step 5. Else go to step 1.
5. The optimal configuration is selected from a set of all configuration that have been evaluated: $C_{opt} = C^* : f(C^*) = f_{\min}$. For this configuration, an optimal program version is generated, as in step 2.

The program obtained as a result of steps 1-5 is saved and executed in a target environment. This program is considered optimal for a given architecture.

4 TuningGenie facilities

As already mentioned in section 3, the expert knowledge about subject domain and implementation is saved in source code as a special directives called "pragmas" (by analogy with C language). "Pragmas" are actually comments of a special form, therefore they are ignored by compiler. Adding pragmas does not change the structure of computation, and such instrumented program can be compiled by any compiler without additional libraries.

Currently the TuningGenie framework supports three kinds of pragmas:

- *tuneAbleParam* – specifies a search domain for an optimal value of a numeric variable. E.g. the following pragma sets the possible values for a threadCount variable as a range [50..100] with a step 10:

```
//tuneAbleParam name=threadCount start=50 stop=100 step=10
int threadCount = 1;
```

  The tuneAbleParam pragma is applicable to the algorithms that use geometrical parallelization: it allows to find the optimal decomposition of computations by estimating the size of a block that executes on a single processor. It also can be applied when we need to estimate the optimal number of some limited resources like size of caches or number of used threads to be used in a program. Another use of the tuneAbleParam pragma is to find an optimal threshold value to switch to a different algorithm. Consider a sort algorithm that uses QuickSort or MergeSort for large arrays and applies the same sort recursively to sub-arrays. For some small size of sub-array, it is more efficient to switch to another sorting algorithm, such as InsertionSort. The optimal threshold value depends on processor architecture and can be determined experimentally. TuningGenie framework allows finding this value with minimal changes to source code:

```
…..……
//tuneAbleParam name=threshold start=10 stop=100 step=5
int threshold = 1;
if (high - low < threshold) {
        insertionsort(array, low, high);
} else {
        addPartitionForQuickSort(array, low, high)
}
…..……
```

- *calculatedValue pragma evaluates some function in a target environment and assigns the resulting value to a given variable:*

```
//calculatedValue name=hdReadSpeed method= //"org.tuning.EnvironmUtils.getHDReadSpeed()"
int hdKbPerSec= 1;
```

  This pragma can be used if some algorithm parameters depend on execution environment details, such as execution times of arithmetic operations or memory access on various levels. All such values are

calculated before auto-tuning process starts and saved in TermWare knowledge base. The data stored in knowledge base is accessible from rewriting rules, so calculatedValue can specify the value range for the tuneAbleParam pragma. In this way, the developer can run a small test program to reduce the search domain and improve auto-tuning speed. The similar approach for GPU computing is described in [10].

- *bidirectionalCycle* – can be used to specify loops were the iterations can be run in any order. In the following example TuningGenie will evaluate both increment (0 to SIZE-1) and decrement (SIZE-1 to 0) versions:

```
int[] data = new int[SIZE];
//bidirectionalCycle
for (int i=0; i <SIZE; i++) {
        doSomethingWith(data[i]);
}
```

Changing iteration order can affects cache efficiency and might significantly improve execution time.

5 Case study

5.1 Problem definition

This section presents a case study on the performance tuning of an application for short-term atmospheric circulation modelling (from hours to several days). Full definition of model and method for numerical solution can be found in [11]. Here we'll show simplified problem definition to give a general idea.

Considered model describes macro-scale processes. Their usual horizontal extent is of the order of thousands of kilometers. Typical examples of such processes are equatorial and monsoonal currents, long Rossby waves [12], cyclones and anticyclones, ridges ant etc. Model is based on spherical coordinate system. This work considers two-dimensional simplified case that models wind characteristics (direction and force). Hence model consists of two evolution equations to calculate horizontal components of wind force.

Model takes state of atmosphere $State(T_0)$ at some moment T_0 as an input. Suppose we want to get prediction of state at some moment in future - $State(T_m), T_m > T_0$. To calculate this program performs series of smaller predictions with time step $\Delta t << T_m - T_0$. So we have recurrence relation between iterations: $State(T_m) = \Pr ediction(State(T_{m-1})), T_m = T_{m-1} + \Delta t$. Note that iteration step Δt should be chosen from the interval [5, 20] minutes to provide acceptable accuracy of the result.

For the experiment initial conditions for model were taken from an electronic archive of regional forecasting center "Offenbach" [13].

Method for numerical solution of described model allows parallelize computation on three levels:

- evolution equation level - each of equation can be solved independently in scope of one iteration;
- geometric decomposition – input data can be decomposed for parallel processing;
- by utilizing a modified additive-averaged operator splitting algorithm [14] it is possible to expand independent processing of subtasks to several calculation iterations without significant loss in accuracy of result

Fig. 2 shows how each evolution equation is parallelized.

1
2
3
4
5

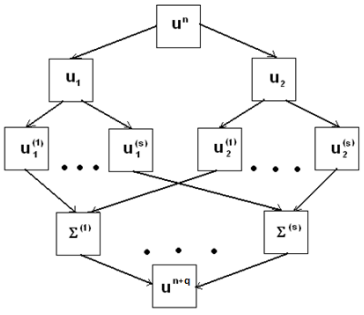

Fig. 2. Parallelization scheme

1. We have some $State(T_n)$ as an input and we want to calculate our prediction for some meteorological characteristic u (humidity, horizontal wind speed etc.). $State(T_n)$ gives us u^n - value of our characteristic at moment T_n.
2. By utilizing modified additive-averaged operator splitting algorithm we are capable to compute prediction gives us possibility to compute prediction u^{n+q} without exchanging intermediate results. So these q steps we are isolated from other sub-tasks. Such approach spares time on synchronization but introduces additional inaccuracy of results. We define boundary conditions for moment T_{n+q} and initial conditions. They are constant for steps 3, 4 and 5. Since our model is two-dimensional, calculation of u is split by special direction on calculation of u_1 and u_2. Each direction is calculated independently.
3. Here geometric decomposition is applied for each direction. Input area is split on s sub-regions so we have $2s$ independent sub-tasks. Each subtask iteratively calculates prognosis for moments $T_n, T_{n+1} ... T_{n+q}$
4. Calculated prognoses for T_{n+q} from previous steps are neutralized by arithmetical mean.
5. Results from previous step are aggregated and we derive u^{n+q}

It's worth mentioning, that model is highly scalable since subtasks with equal size (after decomposition) require equal amount of work independently of what type of physical characteristics is computed.

5.2 Performance tuning

Section "Tuning facilities" provides examples of using tuneAbleParam and calculatedValue pragmas. Let's describe how the third pragma bidirectionalCycle was used in our case study.

Modern strategies of working with processor's cache increase efficiency of traversing application's data by pre-caching "neighbouring" values. For instance, when program reads first element of array – following elements also are fetched to fast L1 cache. So when next iteration starts – required data might be already in cache. That's why sequential iteration over arrays is significantly faster. Count of pre-fetched elements depends on size of element and size of cache line. Basic concepts of CPU caching can be found at [16].

As already said, everything should be good when you access data in array sequentially (even in inverse order). But in some cases, "logically" sequential iteration is not sequential from the data representation point of view. For instance, you can have Matrix<Integer> data structure that represents some usual two-dimensional data (grid) with access by two indexes 'i ' and 'j'. If integers are stored in linearized form in one-dimensional array then one of column-wise and row-wise iteration strategies lead to "jumping" among array elements, hence high rate of cache misses. With "right" strategy simple summation of all elements of square matrix with 4*10^8 integers was constantly faster by 6% on my laptop with Core i7-4850hq CPU (32k L1 cache, 256k L2 cache, 6MB L3 Cache).

So if your algorithm does a lot of iteration over big data collection and reversing cycle's direction can lead to data being accessed in sequential manner – pragma bidirectionalCycle can speed-up it. For our case study, an application for short-term atmospheric circulation modelling, the difference due to such change was almost 15%.

Overall performance of parallel program mostly depends on granularity of data decomposition since it defines amount of calculation per subtask hence size of plain parallel section. It's reasonable to assume that "coarse-grained" decomposition will be the most efficient one. Let's identify number of subdomains as S and number of available processors as N. Search area can be narrowed to values of $S \approx N$, for instance $S \in [Ndiv4, 4 \times N]$. Experiment validated assumption about decomposition $S = N$ being the most efficient one but also revealed other bending point of multiprocessor speedup function:

$$W(N) = \frac{T_n}{T_0}$$

This point represented decomposition of input data into very small subdomains (about a hundred elements). Such result can be explained by suggestion that all data entirely fit processor caches providing additional acceleration of computation. It's worth mentioning that presented framework eases so-called program's "behavior analysis". Since it

empirically evaluates all considered program variations it's easy to discover tuned parameter's impact on overall performance.

5.3 Experiment results

This paper doesn't analyze conformity of derived meteorological prognosis results. Performed research focuses on achieving maximal multiprocessor speedup while preserving acceptable accuracy of results in comparison with sequential version.

Tab.1. Environment configuration

Operating system	*Width*
Processors	2xIntel® Xeon® Processor E5405 Quad Core
Total cores num.	8
L2-cache capacity	12 Mb
RAM	16 Gb

An experiment lasted about two hours. Input data was interpolated up to size 600x600 to load available computational resources. Still, distance between points was about 10 km. Search for optimal value of S was carried by auto-tuner utilizing and the following chart describes how long it takes to calculate 24-hour prognosis depending on value of S. It's significant to mention that with $S=1$ parallel algorithm generates 4 sub-tasks: $2equstions \times 2spartial\_directions$.

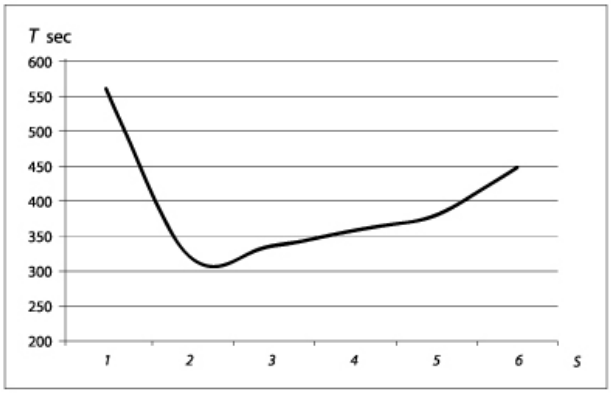

Fig. 3. Time to S relation

Optimized program showed multiprocessor speedup of $W(8)=3.82$ and efficiency $E(8)=47.75\%$.

6 Conclusion

Development of efficient parallel software is a complex task. World-known thesis states that program is symbiosis of algorithms and data structures [15]. Parallel programming model considerably complicates both components and the most efficient programs are derived from their best combination. Variety of modern hardware architectures leads to different combinations being optimal for various computational environments. Thus optimization phase consumes sizeable amount of time and efforts in development of parallel programs. Auto-tuning methodology helps to automate process but in general requires creation of additional software that performs tuning (tuner) or significant changes in software being optimized. This article introduced framework for automated generation of such tuner-apps. It was designed with emphasis on minimization of required source code changes in optimized program. Mostly such changes are declarative and do not affect initial sources. Such flexibility is reached by utilizing rule-rewriting approach for code transformation which significantly differs from related solutions.

Current TuningGenie version contains quite an extensive toolset so reducing time costs of optimization is a main priority in further development. There are several promising directions like introducing functionality for defining relation between pragmas to reduce amount of probed configurations or partial execution of optimized program.

References

[1] Amdahl, Gene M.: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings (30): 483–485, 1967.
[2] Oracle® Solaris Studio http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html
[3] Intel® Parallel Compilers http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers
[4] Naono K., Teranishi K., Cavazos J., Suda R. Software Automatic Tuning From Concepts to State-of-the-Art Results. Springer, 2010. 240p., 2010.
[5] R. Whaley, A. Petitet, J. J. Dongarra, Automated em-pirical optimizations of software and the ATLAS project. Parallel Computing. 2001. 27(1-2), pp.3-35., 2001.
[6] M. Frigo and S. Johnson, FFTW: An adaptive software architecture for the FF. Acoustics, Speech and Signal Processing. 1998. vol. 3, pp. 1381–1384, 1998.
[7] Schaefer C.A., Pankratius V., and Tichy W. F. Atune-IL: An instrumentation language for auto-tuning parallel applications. Euro-Par '09 Proc. 15th Int.Euro-Par Conf. on Parallel Processing Springer-Verlag Berlin, Heidelberg. 2009. pp. 9-20, 2009.
[8] TermWare framework http://www.gradsoft.ua/products/termware_rus.html
[9] Doroshenko, A., Shevchenko, R.: A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae. 72, 1, 95–108, 2006.
[10] Ma W., Krishnamoorthy S., Agrawal G. Parameterized Micro-benchmarking: An Auto-tuning Approach for Complex Applications. Proceedings of the 9th conference on Computing Frontiers. — 2012. — pp.213-222, 2012.
[11] V. A. Prusov, A. E. Doroshenko, R. I. Chernysh, L. N. Guk: Theoretical study of a numerical method to solve a diffusion-convection problem. Cybernetics and Systems Analysis, Volume 44 Issue 2., 2012.
[12] P. N. Belov, E. P. Borisenkov, and B. D. Panin, Numerical Methods of Weather Prediction {in Russian}, Gidrometeoizdat, Leningrad, 1989.
[13] Database of regional meteorological center Offenbach ftp://ftp-outgoing.dwd.de.
[14] V. A. Prusov, A. E. Doroshenko, R. I. Chernysh: A method for numerical solution of a multidimensional convection-diffusion problem. Cybernetics and Systems Analysis, Volume 45 Issue 1., 2009.
[15] Algorithms + Data Structures = Programs Prentice Hall PTR Upper Saddle River, NJ, USA, 1978.
[16] Harvey G. Cragon.: Jones & Bartlett Learning; 1st edition, 1996.