DSL and Autotuning Tools for Code Optimisation on HPC Inspired by Navigation Use Case

Jan Martinovič, Kateřina Slaninová, and Martin Golasowski
IT4Innovations
VŠB - Technical University of Ostrava
Czech Republic
Email: name.surname@vsb.cz

Radim Cmar
Sygic, Slovakia
Email: rcmar@sygic.com

João M. P. Cardoso, and João Bispo
Faculty of Engineering (FEUP)
University of Porto, Portugal
Email: jmpc@fe.up.pt, jbispo@fe.up.pt

Gianluca Palermo, Davide Gadioli, and Cristina Silvano
DEIB
Politecnico di Milano, Italia
Email: name.surname@polimi.it

I. Introduction

When targeting high performance computing (HPC) platforms, improving the performance and scalability of code is often a tedious and time consuming task. The code has to be executed and even compiled many times under different conditions in order to observe its behaviour on the target hardware. These experiments are performed with the goal of estimating an optimal set of the run-time environment parameters to achieve optimal performance and energy efficiency. This optimization task is best performed automatically, but due to the heterogeneous nature of the source codes and the HPC platform, fully automation is often hard to implement.

The domain specific language (DSL) and toolflow approach proposed in the ANTAREX project (*www.antarex-project.eu*) can provide the mechanisms needed for addressing properly the problems previously described. The DSL being developed is based on the LARA DSL [1] and allows to specify strategies for code instrumentation and code transformations, including the required code adaptation for dynamic autotuning [2]. By using instrumentation strategies, the measurements can be integrated seamlessly into the development process, e.g., as a standalone stage in the continuous integration process.

The ANTAREX project plans to demonstrate the usage of the LARA DSL and the autotuning approach on two use cases in cooperation with commercial partners, HPC Accelerated Drug Discovery System (Dompe), and the Self-adaptive Navigation System (Sygic). The poster is focused on the presentation of the DSL and autotuning tools for the self-adaptive navigation system use case, which basic idea is to combine server-side and client-side data knowledge and their routing capabilities to provide the most efficient navigation system in the context of smart cities. In such a use case, we assume a significantly large portion of drivers participating in the system. The efficiency is essential given that the routing system needs to serve many requests requiring potentially huge computation power. Then still, even without limits to be reached, it is desirable to optimize the execution of such a system with respect to the energy efficiency.

The probabilistic Time-Dependent Travel Time Computation algorithm [3] has been selected for the demonstration of DSL and autotuning tools usage in the Self-adaptive Navigation System. The input for the algorithm is a departure time and a selected route composed as a line of road segments. A Monte Carlo simulation (MCS) is used for the computation of the probability distribution of travel time for the selected route. The simulation randomly selects probabilistic speed profiles on road segments and computes travel time at the end of the route. Many MCS iterations are needed to obtain enough travel times for the construction of the probability distribution of travel time. The number of simulation iterations greatly affects the precision of the result.

II. Measuring Performance

The process to measure performance and/or energy consumption consists of several steps. In the first step, the run-time environment is set. In the next step, the code is possibly instrumented and executed to measure a given metric. The execution phase can be further divided into smaller sub-tasks (e.g., data loading phase, execution phase, and post-processing phase). In the next step, the measured metrics are collected and analyzed for each sub-task. These steps are usually repeated for a given set of different run-time parameters (number of threads or processes, compiler flags, etc.).

During this process, there can be a vast number of metrics to be collected (execution time, memory usage, performance counters, power consumption, etc.) each one requiring a different measurement approach and also usually queryable by using their own specialised API, e.g. hardware performance counters API changes on different hardware platforms. Moreover, to obtain measurements for various compiler settings and flags, the user has to often manually change settings of the project build system making it a tedious task. In both cases, the need of custom code and control scripts makes the profiling and optimization step complex and sometimes even impractical.

Various profiling tools can be used to obtain performance measurements of a code on a given hardware platform. However, their use case differs from the problem explained earlier,

since profiling is usually done for a single given environment configuration to tune the code itself and users have little or no control over the actual measurement process. Our approach aims at providing users with methods and tools which allow them to fine tune the application in the context of possibly different target run-time environments while reducing the burden of the necessary code instrumentation and profiling customization.

III. Evaluating Performance and Scalability

The following example is focused on the illustration of typical concerns a developer may face during the preparation of an application (i.e. MCS). To analyze the performance scalability of the simulator, the developer included an external loop able to explore the number of threads from 1 to an upper bound, see Figure 1. The execution time is obtained by inserting a custom code within the application. Then, to consider average execution times, the developer inserted also an inner loop responsible to repeat the execution of the simulator N times, to store each of the execution time measured and to compute the average value. It is this average that is then used to observe the shape of the execution time according to the number of threads. As noticed, these concerns may also expose parameter opportunities, and require the addition of auxiliary code intertwined with the application code increasing the maintenance costs, and are concerns that can be applied to other applications as well.

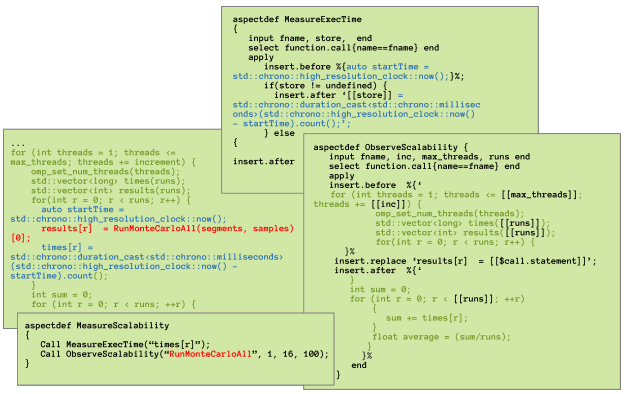

Fig. 1. LARA DSL Aspects

With the ANTAREX approach, the developer would be able to describe these concerns once in the form of LARA aspects and to apply them automatically to a given input application. The LARA aspect shows the modifications in the application in order to observe the scalability using the scheme previously described. Code is inserted in the clean application (i.e. without instrumentation code) before and after the invocation of the MCS. The LARA aspect is specified in a way to promote its reuse by considering a template-based code insertion guided by parameters that can be passed as arguments of the aspect. Another LARA aspect is responsible to insert the code to measure execution time. These two aspects are not optimized in order to increase their reuse as, e.g., one might be interested to measure execution time and not to observe the scalability.

Beside performance monitoring by instrumentation, the aspects can be used to define the optimization logic which would select the best environment parameters according to selected metric (speedup, power/energy consumption, etc.).

IV. DSL and Autotuning for Energy Efficiency

We consider that after the observation regarding the performance scalability, the developer prefers to autotune at runtime the number of threads to minimize the performance instead of defining statically a specific number of threads, or instead of letting the system to use dynamically the number of threads based on the amount of cores, for instance.

Using the ANTAREX approach, the developer programs or reuses aspects to insert the needed code in the application for runtime autotuning of the number of threads, see Figure 2. One aspect is responsible to insert OpenMP directives to parallel loops of the MCS, and the other is able to add code to explore the number of threads and to expose the number of threads as an OpenMP knob. The last aspect inserts code to interface to the autotuner (Margo), to communicate the info needed, and to receive the feedback info (the number of threads).

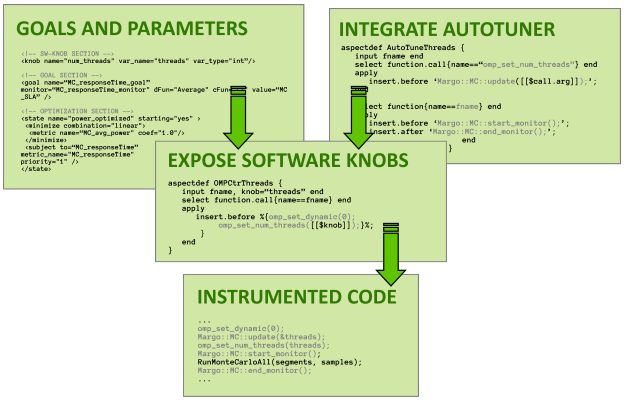

Fig. 2. Autotuning Number of Threads

Summarizing, one of the main advantages of LARA is the modularity provided in this example with invocations of the aspects needed according to the concerns described.

Acknowledgment

This work has been partially funded by ANTAREX, a project supported by the EU H2020 FET-HPC program under grant 671623.

References

[1] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "Lara: an aspect-oriented programming language for embedded systems," in *Proceedings of AOSD*, 2012, pp. 179–190.

[2] D. Gadioli, G. Palermo, and C. Silvano, "Application autotuning to support runtime adaptivity in multicorearchitectures," in *Proceeding of IC-SAMOS 2015*, 2015.

[3] R. Tomis, L. Rapant, J. Martinovič, K. Slaninová, and I. Vondrák, "Probabilistic time-dependent travel time computation using monte carlo simulation," in *Proceedings of HPCSE*, 2016, pp. 1–10.