A Runtime Approach for Dynamic Load Balancing of OpenMP Parallel Loops in LLVM

Jonas H. Müller Korndörfer, Florina M. Ciorba, Akan Yilmaz
Department of Mathematics & Computer Science, University of Basel, Switzerland
firstname.lastname@unibas.ch

Christian Iwainsky
Technische Universität Darmstadt, Germany
christian.iwainsky@sc.tu-darmstadt.de

Johannes Doerfert, Hal Finkel
Argonne National Laboratory
Lemont, IL, USA
[jdoerfert,hfinkel]@anl.gov

Vivek Kale
Brookhaven National Laboratory
Upton, NY, USA
vkale@bnl.gov

Michael Klemm
Intel Deutschland GmbH
Feldkirchen, Germany
michael.klemm@intel.com

ABSTRACT

Load imbalance is the major source of performance degradation in computationally-intensive applications that frequently consist of parallel loops. Efficient scheduling of parallel loops can improve the performance of such programs. OpenMP is the de-facto standard for parallel programming on shared-memory systems. The current OpenMP specification provides only three choices for loop scheduling which are insufficient in scenarios with irregular loops, system-induced interference, or both. Therefore, this work augments the LLVM implementation of the OpenMP runtime library with eleven state-of-the-art plus three new and ready-to-use scheduling techniques. We tested the existing and the added loop scheduling strategies on several applications from the NAS, SPEC OMP 2012, and CORAL-2 benchmark suites. The experimental results show that each newly implemented scheduling technique outperforms the other in certain application and system configurations. We measured performance gains of up to 6% compared to the fastest previously available scheduling techniques. This work establishes the importance of beyond-standard scheduling options in OpenMP for the benefit of evolving applications executing on evolving multicore architectures.

KEYWORDS

Scheduling; dynamic load balancing; OpenMP; LLVM.

1 INTRODUCTION

Parallel and distributed applications in science, engineering, and industry are complex, large, and generally, exhibit irregular and non-deterministic behavior. Moreover, their performance frequently relies on computationally-intensive large parallel loops. High performance computing (HPC) platforms are increasingly complex, large, heterogeneous, and exhibit massive and diverse parallelism. The execution of such applications on existing HPC platforms can suffer from numerous performance degrading phenomena.

Load imbalance is the major source of performance degradation in computationally-intensive applications [1–4]. On shared-memory systems, load imbalance can result from the uneven assignment of work to threads, unequal allocation of threads to processors, or system heterogeneity. The former can be mitigated via scheduling techniques that distribute the work in different manners. It is well known that no single loop scheduling technique can address all sources of load imbalance to effectively optimize the performance of all parallel applications executing on various systems. This poses *the challenge of identifying the most suitable scheduling strategy for a given application-system tuple.* OpenMP is the de-facto parallel programming approach for loops on shared-memory systems offering three scheduling options for work sharing loops: `static`, `guided`, and `dynamic`. These options are insufficient for certain applications-system tuples for which other scheduling strategies can improve performance. Therefore, more scheduling techniques are needed in OpenMP. In this work, we extend the LLVM (llvm.org) OpenMP runtime library (RTL), **libomp**, by eleven state-of-the-art scheduling techniques plus three improved implementations.

We chose the LLVM implementation as it is open-source and widely used in many production and scientific parallel codes. Furthermore, **libomp** is highly compatible with other implementations, such as Intel, GCC, and PGI. The state-of-the-art scheduling techniques added are: Fixed Size Chunking (`fsc`), Factoring (`fac`), Factoring2 (`fac2`), Taper (`tap`), Weighted Factoring (`wf`), Bold (`bold`), Adaptive Weighted Factoring with its 4 variants (`awf_b, awf_c, awf_d, and awf_e`), and Adaptive Factoring (`af`). We also made certain implementation-related improvements to the `fac`, `fac2`, and `af` techniques, hereafter denoted by the suffix "`_a`". We conducted experiments and present in the accompanying poster the results of executing benchmarks from the NAS, SPEC OMP 2012, and CORAL-2 suites with the three standard (static, guided, and dynamic), one non-standard (trapezoidal), and the eleven (plus the three improved implementations) added scheduling techniques.

2 LOOP SCHEDULING IN LLVM OPENMP RTL

Figure 1 illustrates the scheduling process in the LLVM OpenMP RTL. Libomp uses three main functions to perform the scheduling of iterations from a loop onto threads: `init()`, `next()`, and `finish()`. The scheduling techniques

are implemented in **kmp_dispatch.cpp**. The remaining files are needed for the initialization of required environment variables and to make the OpenMP runtime system (RTS) aware of the eleven plus three newly introduced techniques. Upon initialization, each thread enters the `init()` function inside **kmp_dispatch.cpp** whenever a `pragma omp parallel for` with the clause `schedule (runtime)` is encountered inside the application. This function initializes the scheduling technique exported to the environment variable `OMP_SCHEDULE`. The `next()` function is called by every thread to calculate its chunk of iterations following the selected scheduling strategy. Finally, when all loop iterations have been scheduled, every thread calls the `finish()` function which cleans the environment.

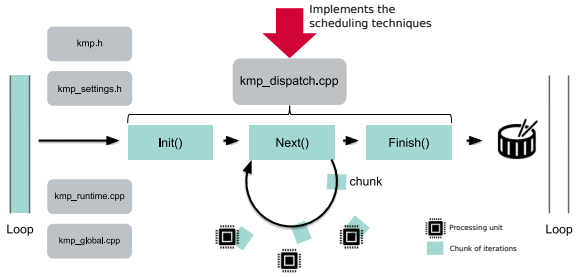

Figure 1: Scheduling process in libomp: kmp_dispatch.cpp implements all scheduling algorithms. The other files are used to select, recognize, and initialize variables for the techniques.

3 EXPERIMENTS AND RESULTS

The experiments were conducted on a system which contains two different node types in separate partitions: Xeon and KNL. A node in the **Xeon** partition is a 10-way dual-socket *Intel Xeon Broadwell E5-2640* processor. A node in the **KNL** partition contains a 64-way *Intel Xeon Phi Knights Landing* processor. We used two pinning strategies denoted **pin1** and **pin2**, which are described in the accompanying poster. All experiments were executed at least 30 times.

Compiler vs. Runtime Library Scheduling

Figure 2 shows the performance of the three OpenMP standard scheduling strategies called via the compiler vs. the runtime library for the well-known EPCC benchmark [1]. The results show that the overhead of calling these scheduling strategies through the LLVM OpenMP RTL is minimal compared to the standard use via the compiler and that it appears only for `dynamic` which makes numerous RTL calls.

Added Scheduling Strategies vs. Standard

The performance of the CORAL-2 Quicksilver benchmark, with 10 million particles, on a node from the Xeon partition using **pin2** is shown in Figure 3. One can see that `wf` outperforms the existing standard scheduling techniques (`static`, `dynamic`, `guided`). The performance results for the other benchmarks from the SPEC2012, NAS, and CORAL-2 suites can be found in the accompanying poster.

[1] www.epcc.ed.ac.uk

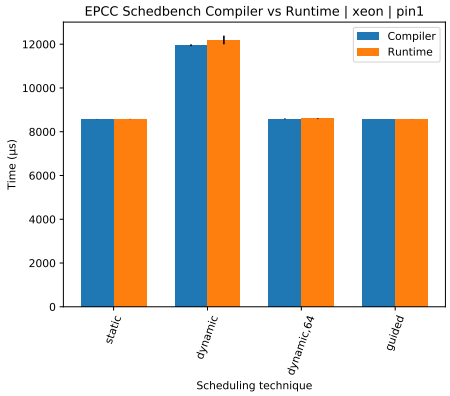

Figure 2: Performance results for executing the EPCC benchmarks using standard OpenMP scheduling strategies via the compiler vs. the LLVM RTL.

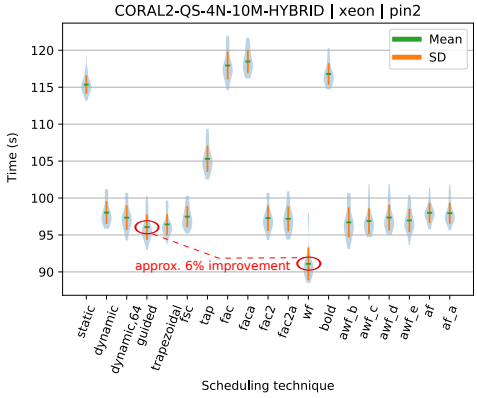

Figure 3: Performance results for executing the CORAL-2 Quicksilver benchmark with 10 million particles on a Xeon node using pin2 using the 3 standard vs. the 11+3 added scheduling techniques.

4 CONCLUSION AND FUTURE WORK

The performance of OpenMP applications can improve beyond the performance delivered by the standard `schedule` clause parameters. These improved performance gains depend, as shown by the experiments, on the degree of load imbalance during the execution of the application on particular systems, which can be mitigated to various extents by the eleven plus three additional scheduling strategies implemented in the LLVM OpenMP runtime library.

The present work establishes the importance and highlights the need for beyond-standard scheduling options in OpenMP. Building upon this and the recent work [1–4], the immediate future work is to develop an interface between the user and the compiler to allow users to define custom scheduling techniques, specialized to the applications' needs.

ACKNOWLEDGMENT

This work is in part supported by the Swiss National Science Foundation in the context of the "Multi-level Scheduling in Large Scale High Performance Computers" (MLS) grant, number 169123, and by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration. improved

REFERENCES

[1] S. Bak, Y. Guo, P. Balaji, and V. Sarkar. 2019. Optimized Execution of Parallel Loops via User-Defined Scheduling Policies. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*. ACM, New York, NY, USA, Article 38, 10 pages.

[2] F. M. Ciorba, C. Iwainsky, and P. Buder. 2018. OpenMP Loop Scheduling Revisited: Making a Case for More Schedules. In *Proceedings of the 2018 International Workshop on OpenMP (iWomp 2018)*. Barcelona.

[3] V. Kale, C. Iwainsky, M. Klemm, J. H. Korndörfer Müller, and F. Ciorba. 2019. Towards A Standard Interface for User-Defined Scheduling in OpenMP. In *Proceedings of the 2019 International Workshop on OpenMP (iWomp 2019)*. New Zealand.

[4] F. Kasielke, R. Tschüter, M. Velten, F. M. Ciorba, C. Iwainsky, and I. Banicescu. 2019. Exploring Loop Scheduling Enhancements in OpenMP: An LLVM Case Study. In *Proceedings of the 18th International Symposium on Parallel and Distributed Computing (ISPDC 2019)*. Amsterdam.