Brief Announcement: Meeting the Challenges of Parallelizing Sequential Programs

Rohit Atre
Technische Universität Darmstadt
Darmstadt, Germany
atre@cs.tu-darmstadt.de

Ali Jannesari
University of California, Berkeley
Berkeley, USA
jannesari@berkeley.edu

Felix Wolf
Technische Universität Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

ABSTRACT

Discovering which code sections in a sequential program can be made to run in parallel is the first step in parallelizing it, and programmers routinely struggle in this step. Most of the current parallelism discovery techniques focus on specific language constructs while trying to identify such code sections. In contrast, we propose to concentrate on the computations performed by a program. In our approach, a program is treated as a collection of computations communicating with one another using a number of variables. Each computation is represented as a Computational Unit (CU). A CU contains the inputs and outputs of a computation, and the three phases of a computation: read, compute, and write. Based on the notion of CU, We present a unified framework to identify both loop and task parallelism in sequential programs.

KEYWORDS

parallelism discovery; multicore architectures; static analysis; task parallelism; profiling;

1 INTRODUCTION

Millions of legacy programs are awaiting their parallelization. Programmers are required to solve a lot of problems while trying to parallelize a sequential program but "Which code sections to run in parallel?" is still one of the most difficult questions that needs to be answered first.

Until now, existing parallelism discovery techniques have been built on top of data-dependence analysis, performed either statically [6, 17] or dynamically [9, 16]. The idea of using data dependences to discover parallelism is based on Bernstein's conditions [4]: Let P_i and P_j be two program sections. I_i and O_i are the sets of input and output variables of P_i. Similarly, I_j and O_j are the sets of input and output variables of P_j. P_i and P_j can be executed in parallel if

$$I_j \cap O_i = \emptyset,$$
$$I_i \cap O_j = \emptyset,$$
$$O_i \cap O_j = \emptyset.$$

To discover parallelism, existing techniques check for the dependences that arise when one of these conditions is violated. However, these techniques do not strictly follow Bernstein's conditions. They do not necessarily distinguish between input and output variables of a code section. As a result, they end up applying data dependence analysis on all of the variables. This makes identifying parallelism unnecessarily complex, and it also leads to both false positives and false negatives in identified parallelism of a sequential program.

This paper discusses an approach to identify both loop and task parallelism in sequential programs following Bernstein's conditions. Our method treats a sequential program as a set of computations that communicate with each other. A computation is represented as a *Computational Unit* (CU). It contains three phases: read phase, compute phase, and write phase. Input and output variables of a CU are clearly distinguished and associated with the read phase and the write phase, respectively. A CU is also created in such a way that read phase is guaranteed to happen before write phase. Data dependences among CUs are obtained using dynamic dependence profiling. In the end, a sequential program is represented as a *CU graph*, in which vertices are CUs, and edges are data dependences.

Our approach targets both loop and task parallelism. A loop can easily be parallelized if there are no inter-iteration dependences in it. Such loops are called DOALL loops [12]. Loops that contain inter-iteration dependences and can still be parallelized are called DOACROSS [12] loops. In case of task parallelism, there are two different kinds of tasks: Tasks that are instances of the same code section but process different data (SPMD) [7], and tasks that execute completely different code sections performing unique computations (MPMD) [18]. Following Bernstein's conditions (BC) and with the help of CU graph, all the above kinds of parallelism can be easily identified. In the end, our method produces parallelization opportunities in sequential programs. A CU (or a set of CUs) provides us with the flexibility and versatility to create this framework. In it, a CU (or a set of CUs) can be used as task, an iteration of a loop, a stage in a pipeline, or other parallel constructs.

We have evaluated our approach by comparing the parallelization opportunities identified by our approach with the existing parallel versions. In addition, we also parallelized these opportunities as parallelizable but not parallelized in the parallel version of the benchmark applications. Our experiments on Barcelona OpenMP Tasks Suite (BOTS) [8], NAS parallel benchmarks [3], PARSEC [5] benchmark suite, and Starbench parallel benchmark suite [1] showed that all of the code sections identified as parallelizable by our approach are parallelized in existing parallel versions.

2 APPROACH

We firstly introduce computational unit, which is the most important concept in our method. Then CU graph, the graph we use to represent a sequential program, is introduced.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SPAA '17, July 24-26, 2017, Washington DC, USA
© 2017 Copyright held by the owner/author(s). 978-1-4503-4593-4/17/07.
DOI: 10.1145/3087556.3087592

2.1 Computational Unit

A *computational unit* is a collection of instructions following the *read-compute-write* pattern: a set of variables is read by a collection of instructions and is used to perform a computation, then the result is written back to another set of variables. The two sets of variables are called *read set* and *write set*, respectively. These two sets do not necessarily have to be disjoint. Load instructions reading variables in the read set form the *read phase* of the CU, and store instructions writing variables in the write set form the *write phase* of the CU.

In practice, program tasks communicate with one another by reading and writing variables that are global to them, and computations are performed locally. A CU is defined by read-compute-write pattern for this reason. This is also why we require the variables in a CU's read set and the write set to be global to the CU. The variables local to the CU are part of the compute phase of the CU as they will not be used to communicate with other tasks during parallelization. We perform variable scope analysis to distinguish variables that are global to a code section. It is available in any ordinary compiler. Global variables in the read set and the write set do not have to be global to the whole program. They can be local to an encapsulating code section, but global to the target code section.

2.1.1 Cautious property. A code section is considered to be a CU only if it is *cautious*. Cautious property [15] was previously defined for operators in unordered algorithms. Ot states that an operator is said to be cautious if it reads all the elements of its neighbourhood before it modifies any of them. By adapting it to the CU, we make sure that a code section is cautious if every variable in its read set is read before it is written in the write phase of the CU.

Cautious property guarantees the read-compute-write pattern of the CU. It not only gives a clear way of separating read phase and write phase, but also after parallelism discovery, it allows multiple CUs to be executed speculatively without buffering updates or making backup copies of modified data. This is possible because all conflicts are detected during the read phase. Consequently, tasks extracted based on CUs also do not have any special requirement on runtime frameworks.

A computation may depend on data produced from other computations. To represent such dependences, we use a dynamic dependence profiler DiscoPoP[13]. DiscoPoP profiles detailed data dependences, gathers control-flow information, and identifies hotspots across the target program. We run the profiler multiple times using representative inputs to overcome the input sensitivity of the dynamic dependence analysis. Then we merge the dependence results obtained. These representative inputs are provided along with the benchmarks and they are of varying size and complexity. Next, we build a *CU graph*

2.2 CU Graph

A computation may depend on data produced from other computations. To represent such dependences, we use a dynamic dependence profiler DiscoPoP[13]. DiscoPoP profiles detailed data dependences, gathers control-flow information, and identifies hotspots across the target program. We run the profiler multiple times using representative inputs to overcome the input sensitivity of the dynamic dependence analysis. Then we merge the dependence results obtained. These representative inputs are provided along with the benchmarks and they are of varying size and complexity. Next, we build a *CU graph*, in which vertices are statically generated CUs and edges are dynamic data dependences. Hence, the CU graph combines static and dynamic information to help us discover parallelism. Data dependences in a CU graph are always among instructions in read phases and write phases. Dependences that are local to a CU are hidden because they do not prevent parallelism among CUs according to Bernstein's conditions. Moreover, since the number of global variables to a code section is usually far less than the number of local variables, a CU graph is much simpler than the traditional instruction-based dependence graph. This simplifies the parallelism discovery process. The CU graph is then expanded using runtime information to represent instances of tasks or loops, if necessary.

2.3 Parallelism discovery

There are two kinds of parallelism we can identify: parallelism among different computations, and parallelism among different instances of the same computation. Parallelism among different computations can be easily identified using CU graphs without instantiating these computations.

On the other hand, identifying parallelism among different instances of the same computation requires some additional effort. To discover such parallelism, a CU must be instantiated using real inputs passed into the computation and real outputs it produces. A CU graph consisting instantiated CUs is called an *expanded* CU graph.

2.3.1 ***Task parallelism.*** Task parallelism is discovered based on the following rules:

(1) A CU is instantiated into different instances using their real inputs and outputs with respect to the control flow. Two instances of the same computation can run in parallel if they are independent in the expanded CU graph (SPMD task parallelism).
(2) Two different computations can run in parallel if their corresponding CUs are independent in the CU graph (MPMD task parallelism).

Parallelism within a computation is not covered by our current approach. However, such parallelism can be detected by further applying techniques that track def-use chains [2] on compute phases of CUs. Nevertheless, in rare cases where a code section contains plenty of lines of code and complex computation (which is unlikely to be cautious), analyzing CUs built for it might provide opportunities to break the region down to smaller computations, leading to parallelism that is similar to OpenMP sections [14].

2.3.2 ***Loop parallelism.*** Loop parallelism is discovered based on the following rules:

- Iterations of a loop can run in parallel if for all CUs built for the loop, there are no inter-iteration *read-after-write* (RAW) dependences among the CUs or on a single CU (DOALL parallelism).
- If there are inter-iteration dependences in the loop, the loop may still be analyzed to check if it can be parallelized by using techniques e.g. reduction, privatization, pipeline etc. (DOACROSS parallelism).

DOALL parallelism: In case of DOALL loops, the iterations of loop are independent of each other. This means that the instances of CUs are also independent and as a result they satisfy Bernstein's conditions. However, it is not necessary to analyze all the instances of the CUs of a loop because loops have no parameters to be replaced, unlike function calls and their instances.

DOACROSS parallelism: It is possible to further analyze the dependence distances of the inter-iteration RAW dependences to discover DOACROSS [12] loops. A DOACROSS loop has inter-iteration dependences, but the dependence distance should not be as large as the distance between the first line of an iteration and the last line of the previous iteration. In other words, the first CU of the loop should not depend on the last CU of the loop. This raises the possibility that iterations of a DOACROSS loop can overlap, thus containing parallelism. Based on our CU graph, if the length of the longest dependence is smaller than the distance from the last CU to the first CU, we can classify such loops as candidates for DOACROSS parallelism. All the instances of CUs are not independent of each other in DOACROSS loops as there are inter-iteration dependences, unlike DOALL loops. But a subset of CUs in an iteration can be independent of another subset of CUs in the next iteration. Hence they satisfy Bernstein's conditions and can be run in parallel. Utilizing such parallelism is usually achieved by applying the pipeline pattern [11] [10].

3 CONCLUSION

This paper discusses an approach for discovery of parallelism by identifying code sections called computational units (CU) in sequential programs. A CU follows a read-compute-write pattern. CUs are detected statically from the source code using the cautious property. A CU graph is then created using the CUs and the dynamic dependences. This serves as the basis for parallelism detection. Bernstein's conditions are used to identify the tasks which can run in parallel to each other from a CU graph or an expanded CU graph

REFERENCES

[1] Michael Andersch, Ben Juurlink, and Chi Ching Chi. 2011. A Benchmark Suite for Evaluating Parallel Programming Models. In *Proceedings 24th Workshop on Parallel Systems and Algorithms (PARS '11)*. 7–17.

[2] Rohit Atre, Ali Jannesari, and Felix Wolf. 2015. The Basic Building Blocks of Parallel Tasks. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores (COSMIC '15)*. ACM, New York, NY, USA, Article 3, 11 pages.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications* 5, 3 (1991), 63–73.

[4] Arthur Bernstein. 1966. {Analysis of programs for parallel processing}. *IEEE Transactions on Electronic Computers* 15, 5 (1966), 757–763.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. DOI: http://dx.doi.org/10.1145/1454115.1454128

[6] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. 1989. Automatic generation of nested, fork-join parallelism. *The Journal of Supercomputing* 3, 2 (1989), 71–88. DOI: http://dx.doi.org/10.1007/BF00129843

[7] Frederica Darema. 2001. The spmd model: Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 1–1.

[8] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP'09. International Conference on*. IEEE, 124–131.

[9] Jialu Huang, Thomas B Jablin, Stephen R Beard, Nick P Johnson, and David I August. 2013. Automatically exploiting cross-invocation parallelism using runtime information. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. IEEE, 1–11.

[10] Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. 2016. Automatic Parallel Pattern Detection in the Algorithm Structure Design Space. In *Proc. of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, USA*. IEEE Computer Society, 43–52. DOI: http://dx.doi.org/10.1109/IPDPS.2016.60

[11] Zia Ul Huda, Ali Jannesari, and Felix Wolf. 2015. Using Template Matching to Infer Parallel Design Patterns. *ACM Trans. Archit. Code Optim.* 11, 4, Article 64 (Jan. 2015), Article 64, 21 pages. DOI: http://dx.doi.org/10.1145/2688905

[12] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] Zhen Li, Ali Jannesari, and Felix Wolf. 2015. An Efficient Data-Dependence Profiler for Sequential and Parallel Programs. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS '15)*. 484–493.

[14] OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. (May 2008). http://www.openmp.org/mp-documents/spec30.pdf

[15] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. *SIGPLAN Not.* 46, 6 (June 2011), 12–25. DOI: http://dx.doi.org/10.1145/1993316.1993501

[16] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. 2010. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput.* 36, 9 (2010), 531–551.

[17] Vivek Sarkar. 1991. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development* 35, 5.6 (1991), 779–804.

[18] Barry Wilkinson and Michael Allen. 1999. *Parallel programming*. Vol. 999. Prentice hall New Jersey.