Chapter 7

Advances in Engineering Software for Multicore Systems

Ali Jannesari

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.72784

Abstract

The vast amounts of data to be processed by today's applications demand higher computational power. To meet application requirements and achieve reasonable application performance, it becomes increasingly profitable, or even necessary, to exploit any available hardware parallelism. For both new and legacy applications, successful parallelization is often subject to high cost and price. This chapter proposes a set of methods that employ an optimistic semi-automatic approach, which enables programmers to exploit parallelism on modern hardware architectures. It provides a set of methods, including an LLVM-based tool, to help programmers identify the most promising parallelization targets and understand the key types of parallelism. The approach reduces the manual effort needed for parallelization. A contribution of this work is an efficient profiling method to determine the control and data dependences for performing parallelism discovery or other types of code analysis. Another contribution is a method for detecting code sections where parallel design patterns might be applicable and suggesting relevant code transformations. Our approach efficiently reports detailed runtime data dependences. It accurately identifies opportunities for parallelism and the appropriate type of parallelism to use as task-based or loop-based.

Keywords: parallelism, multicore/manycore systems, software engineering, code analysis, profiling

1. Introduction

Stagnating single core processor performance caused a new hardware trend in the past years that resulted in the replication of cores and the popularity and ubiquity of multicore and manycore architectures. Many applications and software systems that face growing demand for computational power can leverage this hardware trend for their needs and achieve reasonable performance via software parallelization. The only way for application

developers to speed up an individual application is to match the new hardware cores with thread-level parallelism in the form of task-based or loop-based parallelism. However, successful parallelization is often error prone, difficult, and time-consuming, especially if it is done manually. Further, applying auto-parallelization is generally limited to loops with specific criteria, and it is based on the polyhedral model [1, 2] for compiler optimization. Additionally, auto-parallelization often fails to identify and exploit available parallelism for many applications, since it does not leverage runtime information such as pointers and array indices.

To keep application developers motivated and encourage them to achieve a performance improvement, automated tools and methods are necessary that support them during a semiautomatic parallelization process to reduce the manual efforts and facilitate the parallelization workflow. Hence, effective programming toolchain and methodologies for using an optimistic code-based approach to parallelize software with minimum programming effort and user intervention are in great demand.

There are three major problems in the software parallelization process that often cause the parallelization process to suffer from high complexity and low productivity. The first problem is gaining a thorough and complete understanding of the software code to identify detailed control and data dependences. In order to guarantee the program correctness, the parallelized program must have proper synchronization operations to preserve data dependences and the right order of data accesses to produce the same results as the sequential code does.

The second problem is extracting coarse-grained parallelism. Because of the available hardware parallelism in multicore/manycore processors, they are powerful in executing multiple code sections simultaneously. But the software programming toolchain is not mature to help programmers partition and map their code to the new available cores. Coarse-grained parallelism such as task-based parallelism is expected to be a promising solution for using the available hardware parallelism of the new cores and finding parallelism between arbitrary code sections.

The third problem is generating parallel code which can express this coarse-grained parallelism effectively for a diverse number of target platforms. After generating the parallel code, validation and verification will be applied, and for further performance improvements on specific targets, optimization techniques and auto-tuning methods are necessary.

This work summarizes the results of methods and approaches, which set out to improve the abovementioned problems. The main goal of the work was to make semiautomatic parallelization more feasible and attractive for a broader audience of application developers by providing tools and methods that use an optimistic code-based approach and support key activities of the manual parallelization process in a simpler, more effective and intuitive way than existing tools.

The remainder of the chapter is structured as follows: in the next section, we highlight the main contributions and the essential results of this work. In Section 3, we explain our

approach to dependence profiling and decomposition. In Section 4, we briefly present our methods for task extraction and parallel pattern identification. Sections 5 and 6 deal with code transformation and correctness analysis, respectively. In Sections 7 and 8, several applications of our framework and its limitations are discussed. Section 9 reviews related work. Section 10 concludes the chapter and discusses possible extensions.

2. Contribution summary

The most important goal of this work is to provide a set of methods as an end-to-end solution to support programmers during the semiautomatic parallelization process, from the initial code analysis to code generation and optimization. The methods follow an optimistic code-based approach to effectively analyze different code sections based on the actual runtime dependence analysis. In this way, parallelization opportunities of applications can be identified at an early stage of the code analysis, which maximizes flexibility and also facilitates the parallelization process. The approach is implemented as a tool called Discovery of Potential Parallelism (DiscoPoP) [3] and is based on the LLVM compiler infrastructure.

The main accomplishments, which are illustrated in **Figure 1**, can be summarized as follows:

- **Dependence profiling.** We instrument and execute the program to obtain its control and data dependences with practical overhead [4]. Our data-dependence profiler serves as foundation for different program analyses based on data dependences.
- **Decomposition.** The concept of computational units (CUs) is used to extract the basic blocks for building parallel programs [5]. A CU follows the read-compute-write pattern, which means that a program state is first read from memory, a new state is computed, and finally the new state is written back to memory. We generate the CU graph of a program based on its CUs and the dependences that exist among them.
- **Task extraction and parallel pattern identification.** We search for potential parallelism in the program by merging CUs/partitioning the CU graph [3]. The output is a prioritized list of parallelization opportunities [6]. In a next step, we identify suitable parallel design patterns to support the parallel algorithm structure [7, 8].
- **Code transformation.** In simple cases, the program is automatically transformed into its parallel version based on available parallelism and the identified parallel design patterns [9]. In other cases, suggestions for parallelization are presented.
- **Correctness analysis.** Additionally, an automated method to generate unit tests targeting concurrency bugs such as data races has been developed to validate the resulting code [10, 11].
- Numerous applications and case studies, in which we confirm the functionality of our approach and show its capability as a parallelism discovery tool. In many cases, we can

reproduce manual parallelization strategies. We further demonstrate how DiscoPoP can serve as a basis for different kinds of program analyses such as the exposure of communication patterns [12] or the optimization of transactional memory [13, 14].

In the remainder of the chapter, we describe the above contributions in more detail, followed by a quick look at ongoing developments and a review of related work.

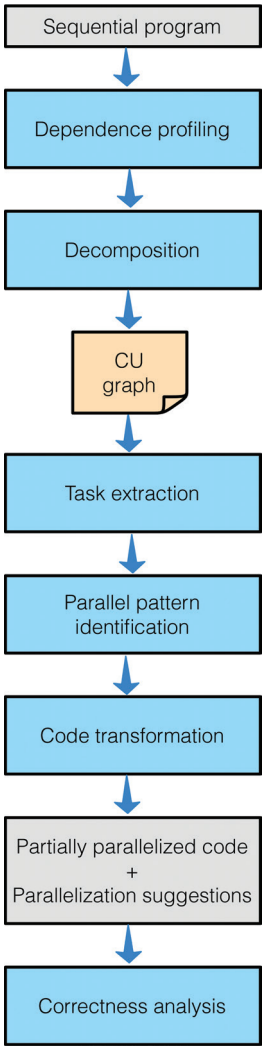

Figure 1. Main elements of the DiscoPoP parallelization approach.

3. Dependence profiling and decomposition

A key result of this work is a profiling method that reports data dependences of the executed program efficiently in terms of time and space. The reported data dependences are used to build the computational units that serve to analyze the program.

3.1. Dependence profiling

In order to parallelize the sequential code, we need to identify the control and data dependences of the program. Data dependences can be obtained in two major ways: static and dynamic analyses. Static approaches determine data dependences at compile time without executing the program. However, many parallelization opportunities are ignored due to the lack of runtime information. In contrast, dynamic dependence profiling instruments the intermediate or binary code and tracks dependences at runtime. It treats the execution of a user program as an instruction stream interrupted by previously inserted calls to instrumentation functions that help detect dependences. Since dynamic profiling tracks only the branches that are actually executed, it is inherently input sensitive, and it identifies control and data dependences for the actual program execution. Despite this, the results are still useful, which is why such profiling forms the basis of many program analysis tools. Moreover, by changing inputs and computing the union of all collected dependences, the input sensitivity can be mitigated.

However, a limitation of data-dependence profiling is high runtime and memory overhead. The time overhead may significantly prolong the analysis, sometimes requiring an entire night [15]. The memory overhead may prevent the analysis completely [16]. This is because dependence profiling requires all memory accesses and locations to be instrumented and recorded. To lower the overhead, current profiling approaches limit their scope to the subset of the dependence information needed for the analysis they have been created for. In this way, they reduce the generality and reusability.

To provide a general foundation for different kinds of analyses, we present a generic data-dependence profiler with practical overhead, capable of supporting a broad range of dependence-based program analysis and optimization techniques for both sequential and parallel programs. The profiler is based on LLVM-IR, and it provides detailed information, including source-code location, variable name, and thread ID.

The proposed profiler is parallelized and utilizes a lock-free design [17] to achieve efficiency. It leverages signatures [18], a concept borrowed from transactional memory to reduce memory consumption. A signature is a data structure that encodes an approximate representation of an unbounded set of elements with a bounded amount of state. It is widely used in transactional memory systems to uncover conflicts [18]. A data dependence is similar to a conflict in transactional memory because it exists only if two or more memory operations access the same memory location in some order. Therefore, a signature is also suitable for detecting data dependences.

We evaluated our approach using the NAS parallel benchmark suite (NAS) [19] and Starbench parallel benchmark suite (Starbench) [20]. The performance results are shown in **Table 1**.

Benchmark	Average slowdown			Average memory consumption (MB)	
	1T	8T	16T	8T	16T
NAS	190	97	78	473	649
Starbench	191	101	93	505	1390

Table 1. Performance results of profiler in DiscoPoP.

While performing an exhaustive dependence search with 16 profiling threads, our lock-free parallel design limited the average slowdown to 78× and 93× for NAS and Starbench, respectively. Using a signature with 10^8 slots, the memory consumption did not exceed 649 MB (NAS) and 1390 MB (Starbench).

3.2. Decomposition

The most difficult and challenging part in parallelizing sequential programs is to identify which code sections are able to run in parallel. While identifying such code sections, most of the current parallelism discovery techniques focus on specific language constructs. In contrast, we propose the concept of computational units (CUs) to concentrate on the computations performed by a program independently of any language constructs.

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 is a collection of instructions following the *read-compute-write* pattern: a set of variables is read and used to perform a computation, and 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. A CU is defined by a read-compute-write pattern because, in practice, tasks communicate with one another by reading and writing variables that are global to them, while computations are performed locally.

We build a *CU graph*, in which vertices are statically generated CUs and edges are dynamic data dependences. 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. Our tool also generates the program execution tree (PET) of a program. This tree contains information about program control dependences and execution paths. Nodes of the tree are control regions of the program. We map the CU graph of a program onto its execution tree to determine CUs for every region. **Figure 2** shows the CU graph of a program mapped onto its PET. PET and CU graphs serve for different kinds of code analyses as they contain the information such as CUs and their corresponding instructions, data and control dependences, etc. In this work we mainly use them for parallelization.

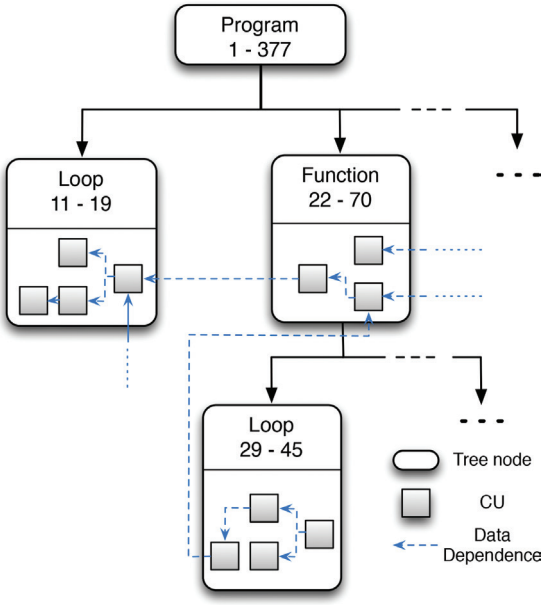

Figure 2. CU graph mapped onto the program execution tree.

4. Task extraction and parallel pattern identification

In the following section, we focus on using PET and CU graphs for parallelism discovery and supporting the parallelization process. Additionally, we describe how to use them to detect parallel design patterns in the given sequential code automatically.

4.1. Task extraction

DiscoPoP suggests parallelism among strongly connected components (SCCs) and chains in the CU graph of a program [21]. An SCC is a subgraph of the CU graph in which every CU is reachable from every other CU. It forms a complex knot of dependences that defy internal parallelization. A chain is a group of CUs that are connected in a row without a branching or joining point in between. We merge chains because a CU contains only a few instructions and there is no benefit in considering each CU as a separate task. The CUs grouped as SCCs or chains could form separate tasks and be executed in parallel, if there are no dependences between them. Parallelism is also possible when dependences are weakly connected. DiscoPoP discovers these parallelization opportunities by calculating affinity between CUs and applying the minimum cut algorithm [22] to the CU graph. It calculates the affinity for every pair of CUs based on the number of dependences and shared instructions between them. DiscoPoP suggests to partition the CU graph with a minimum number of dependences and affected shared instructions.

Finally DiscoPoP ranks parallelization opportunities to prioritize them based on three metrics [21]. The first is instruction coverage. It provides an estimate of how much time is spent in a code section. The second is speedup, which reflects potential speedup if the code section is parallelized. The third is CU imbalance. It reflects when suggested parallelization may lead to a bottleneck. Our experiments with Barcelona OpenMP Tasks Suite (BOTS) [23], NAS, PARSEC benchmark suite (PARSEC) [24], and Starbench showed that all of the code sections identified as parallelizable by our approach are parallelized in the existing parallel versions of the benchmark programs [3, 21, 25, 26].

4.2. Parallel pattern identification

Parallel design patterns are reusable solutions for common problems that occur during the development of parallel programs. They have been developed to help programmers to design and implement parallel applications efficiently [27, 28]. However, identifying a suitable parallel pattern for a specific code region in a sequential application is a difficult task. Also, transforming the application according to structures supporting those parallel patterns is very challenging.

We propose an approach that automatically finds parallel patterns in the algorithm structure design space of sequential applications using template matching [7, 8]. The approach generates a pattern vector, which plays the role of the template to be matched to the program. For each hotspot in the program, we create the pattern-specific graph vector according to the dependences of the corresponding CU graph. The correlation coefficient of the pattern vector and the graph vector of the selected hotspot tells us whether the pattern exists in the selected section or not. So far, we support the detection of pipeline, do-all, task-level parallelism such as master/worker, geometric decomposition, and reduction patterns. Our tool not only indicates whether a parallel design pattern has been found in some section of the program but also shows how the code must be divided to fit the appropriate structure of the pattern.

We evaluated our approach with 17 sequential applications from four different benchmark suites, i.e., Starbench, BOTS, PARSEC, and PolyBench [29]. We successfully detected pipeline, task parallelism, geometric decomposition, fusion, and reduction. We compared the detected patterns with the existing parallel versions of the benchmarks and confirmed our results [7, 8]. For those benchmarks for which the parallel version does not exist, we implemented the detected patterns. We achieved a speedup of 14× with 32 threads for the best case of our hand-implemented parallel version of the *ludcmp* application in PolyBench [8].

5. Code transformation

After finding parallelization opportunities in the program, generating parallel code to run on the hardware is another main step of the parallelization process. The code transformation component [9] in DiscoPoP transforms sequential C/C++ code into parallel code, following

the detected parallel design pattern. Transformation is performed on the AST level using the Clang libraries [30]. The transformation module traverses the Clang AST of the source code to locate simple detected patterns such as do-all or task-level parallelism and the corresponding code sections. Then a source-code rewriting module rewrites the targeted source-code strings in the Clang AST context using Intel TBB parallel constructs—the *parallel_for* and *flow graph* templates. The transformation process does not require users to annotate parallel code sections in advance. The parallel for-loop transformation is automatic, and the parallel-task transformation using the flow graph template requires user assistance. The user needs to specify the buffering policy in the synchronization join node of the flow graph.

We have evaluated applications from NAS, PARSEC, and Intel CnC samples. The obtained results confirm that our approach is able to achieve promising performance with minor user interference. The average speedups of loop parallelization and task parallelization are up to 3.12× and 9.92×, respectively [9].

6. Correctness analysis

The parallelized code is expected to deliver the same output as its sequential counterpart. To assure the correctness, we use automated unit testing [10, 31, 32] and sophisticated race detection methods [11, 33]. For data races, our library-independent race detection approach [33] is applied to the generated parallel code for finding potential data races. Also, automated unit tests based on dynamic and static analyses are generated, which can be used during the parallelization process for finding atomicity violations and verifying the parallel code.

A notorious class of concurrency bugs is race conditions related to nonatomic updates on correlated variables, potentially leading to broken invariants, which make up about 30% of all non-deadlock concurrency bugs. We propose to combine the benefits of automatic parallel unit test generation with the advantages of race detection. To achieve this, our framework uses the existing unit test generator AutoRT [34, 35] to identify possible correlation violations in function pairs accessing correlated variables. We automatically generated 81 parallel unit tests for correlated variables in eight different applications. After analyzing the unit tests, a race detector for correlated variables reported more than 85% of the race conditions violating variable correlations [36]. Furthermore, we were able to reduce the number of redundantly generated tests by up to 50%.

7. Further applications of the DiscoPoP framework

Considering the modern parallel programming models and hardware platforms, communication patterns play an important role for energy efficiency and performance of the generated parallel code. We investigated communication patterns in shared-memory applications, which are useful for applying optimizations and finding performance bottlenecks.

7.1. Communication pattern detection for auto-tuning

Communication patterns extracted from parallel programs can provide a valuable source of information for auto-tuning and runtime workload scheduling on heterogeneous systems. Once identified, such patterns can help find the most promising optimizations. Communication patterns can be detected using different methods, including sandbox simulation, memory profiling, and hardware counter analysis. However, these analyses usually suffer from high runtime and memory overhead, necessitating a tradeoff between accuracy and resource consumption. More importantly, none of the existing methods exploit fine-grained communication patterns on the level of individual code regions.

We extended the DiscoPoP profiler by adding a communication pattern detection component and employed an asymmetric signature memory method to detect communication among threads [12]. Shared-memory systems have fundamental differences in comparison with distributed memory system. Shared-memory applications bring additional irregularity and complexity to data sharing, which imposes further difficulty on finding the communication pattern. Communications are implicit and automatically occur through memory accesses, when one thread writes a value and another one reads it. We experimentally validated our communication pattern detection approach with programs in the SPLASH [37] benchmark suite and successfully identified the typical communication patterns existing in parallel programs [12]. The runtime overhead of our extended profiler is around 225×, while the required amount of memory remains fixed.

7.2. Optimization techniques for transactional memory

Transactional memory (TM) is a promising paradigm that facilitates programming for shared-memory systems. We used DiscoPoP and reported optimization techniques in software transactional memory (STM) [13]. We demonstrated that varying STM parameters such as the size of transaction, readset, and writeset significantly change the execution time of the STM programs. By applying machine learning and using DiscoPoP results, we optimized these parameters. The experimental results with NAS revealed that we are able to improve the performance of STM programs by up to 54.8%. In another work [14], we used DiscoPoP for restricted transactional memory (RTM) on Intel's Haswell processor and showed that the performance of RTM varies across applications. While RTM enhances performance of some applications relative to software transactional memory (STM), it degrades performance in some others. Using DiscoPoP, we proposed an adaptive system that switches between HTM and STM in transaction granularity and predicts the optimal TM system for a given transaction.

8. Limitations

Methods to analyze programs are generally divided into two categories: static and dynamic methods. Static methods analyze source or intermediate code and are restricted to information

that can be obtained before running the program, i.e., at compile time. Static approaches are fast but also conservative because they have limited support for runtime information. In contrast, dynamic approaches identify dependences only if they exist at runtime. Although dynamic approaches relax the conservative assumptions made by static approaches on dynamic data, they are input sensitive, that is, their outcome may depend on the particular program execution. To mitigate this limitation, dynamic approaches generally execute the target program with a range of representative inputs, a practice we adopt as well when using our dynamic dependence profiler. Additionally, we plan to derive conditional correctness guarantees, taking the specific nature of the missing dependences into account. This would allow users to run the program in parallel if the missed dependences are irrelevant for a given input configuration.

Our parallelism detection approach is dependent on the profiler's output. Due to the use of signature technology (as an approximation), the dependence profiler could have a very low rate of false positives and false negatives. If these appear in the profiler results, then our dependence analysis and accordingly the parallelism detection results could be affected.

The pattern detection approach is dependent on the coding style of a programmer. For example in a Starbench program, we found a pipeline pattern because the programmer used pointers to access arrays and used the increment operator (++) on pointers. However, if a loop index variable (loop indexing) had been used in the sequential code, we would have detected a do-all loop pattern based on our pattern detection algorithm (template matching).

MPMD-style task parallelism (Multiple Program Multiple Data) could be found in few evaluated benchmarks, which lead to minor speedups. More intensive investigations are needed to apply such kind of parallelism to real-world applications.

Recently, we developed a prototypical visualization component to display the output of DiscoPoP. However, a more advanced and user-friendly graphical user interface to visually guide application developers in a stepwise manner when parallelizing a program would be desirable. This feature could create higher incentives for developers and make the parallelization workflow easier, particularly for DiscoPoP's code-based semiautomatic parallelization approach.

9. Related work

Profiling and parallelism discovery has always been a central topic in the field of parallel programming. Early approaches mainly analyze source code statically and predict parallelism based on theoretical models [38, 39]. Bobbie [40] presented a method to partition a program for parallelization. The method adopts syntax-driven data-dependence analysis and detects parallelism based on Bernstein's conditions [41]. It uses bipartite graph matching to partition the code.

Compiler-based auto-parallelization based on the polyhedral model [1, 2, 42] is generally restricted to program loops with specific criteria (e.g., *affine* linear loop boundaries and array indices). A dynamically speculative extension of these criteria expands the applicability of this method to a certain extent [43]. Another work [28] that is not restricted to loops only identifies

parallel tasks in the static dependence graphs using integer linear programming. Generally, compiler-based auto-parallelization is often conservative and fails to identify available parallelism for many applications, because runtime information such as the values of pointers and array indices are often not known at compile time. In practice, the parallelization of software usually happens manually, and often, it is more appropriate to follow the provided guidelines for parallel design patterns [44].

Without considering runtime behavior of the target program, some key parallelism usually could remain undetected in static approaches. To overcome this disadvantage, profiling techniques to gather runtime data emerged [45, 46]. Such methods are usually referred to as dynamic parallelism discovery approaches. Additionally, most of these approaches have a cost model to produce results. Kremlin [47] determines the length of the critical path in a given code region. Based on this knowledge, Kremlin calculates a metric called self-parallelism to quantify the parallelism of a code region. The tool reports self-parallelism for each region in a descending order. Alchemist [48] identifies predefined constructs that can be treated as candidates for asynchronous execution in sequential programs. It estimates the effectiveness of parallelizing a certain construct using Valgrind [49]. Kremlin and Alchemist mainly focus on loops, which are easier to profile and quantify.

At the same time, other dynamic parallelism discovery approaches deal with task parallelism as tasking became popular and widely supported in almost all mainstream parallel programming libraries and frameworks. Ketterlin et al. [50] profiles sequential programs and represents them using execution trees. It further attaches data dependences to the nodes of the execution tree and discovers task parallelism where two or more nodes are independent of one another. The SLX Tool Suite, formerly known as MAPS [51], concentrates on parallelism discovery for applications on multiprocessor system on chip (MPSoC). It identifies code sections called *coupled blocks*. These code blocks are identified with constraints requiring that they should be schedulable and should be tightly coupled by data dependences. Each coupled block is considered as a task, and two tasks can run in parallel if there is no data dependence between them.

Tareador [52] provides a set of annotations for marking down tasks in the code. It takes a relatively brute-force approach by enumerating possible decompositions and does not take the control flow into account, which may lead to tasks that are not easy to implement. Intel Advisor XE [53] is a prototyping tool for different programming languages such as C, C++, C#, and Fortran. It also performs a correctness check, which is essentially a data-race detector and has a large time overhead. Also pattern detection and code transformation are not supported by Intel Advisor XE.

The approach presented by Tournavitis et al. [54] uses both static analysis and dynamic profiling to detect potential parallelism. A machine learning-based prediction mechanism maps the parallelism onto different architectures. It generates parallel code using OpenMP annotations and targets loop-based parallelism. However, the code transformation is relatively simple. The tool does not perform high-level code restructuring that could exploit coarse-grained task parallelism. In recent work [55], the tool exploits pipeline parallelism.

OpenRefactoryC [56] is a tool providing many refactoring methods for C programs, but it does not automatically transform sequential code to parallel code. The approach presented in [57] transforms serial C++ code to parallel code using OpenMP directives. However, it requires users to define the high-level abstractions in advance.

Similar to all the dynamic parallelism discovery approaches, the DiscoPoP approach adopts profiling techniques to gather runtime data. However, our method discovers parallelism based on computational units (CUs), which are derived statically, and parallel design patterns. We identify CUs in sequential programs and build the CU graph as the representation of a program. Based on the CU graph, we can perform different analyses and detect parallel design patterns. A CU clearly distinguishes the inputs and outputs of a computation, allowing a direct application of Bernstein's conditions [41]. Bernstein's conditions describe when two program segments are independent and can be executed in parallel. In addition, our method discovers both task-based and loop-based parallelisms using the same framework. A CU in our approach acts as a task, a stage in a pipeline, or an iteration of a loop or a subset of either of these based on the context, which distinguishes our work from related work.

10. Conclusion and outlook

In this chapter, we propose an optimistic code-based approach to assist semiautomatic parallelization in multicore architectures, focusing on general-purpose applications. Our approach is implemented as an integrated tool based on LLVM. Program analysis and parallelism discovery are performed at the LLVM-IR level and are not limited to any programming language or specific language constructs.

The proposed approach presents an alternative to conservative and usually loop-centric autoparallelization. Application developers can take advantage of our methods to identify and exploit parallelism that is not necessarily limited to loop parallelism for many applications. Our semiautomatic approach can reduce the high cost and price of the manual error-prone parallelization process. At the same time, many legacy applications can benefit from the available hardware parallelism using our approach.

There is no doubt that parallel programming is challenging and involves a steep learning curve. Developers must think about the application in new ways. It is possible to work months on parallelizing an application and end up with incorrect results, or the resulting parallel program runs slower than the sequential one. For this reason, the techniques and tools used for parallelism discovery, debugging, and tuning the performance during the parallelization process play a very significant role. Our approach supports application developers during this process. It encourages average programmers to use parallel programming by creating incentives and insight for developers and making the parallelization workflow easier.

Considering the hardware trend and future smart cyber-physical systems (smart factory 4.0), energy-efficient programming is a key feature in improving productivity and efficiency.

Whether we develop an application for mobile devices or data centers, we want to reduce energy, e.g., to increase the battery life of a mobile device or lower the customer's data center utility bill. Thus, the role of programmers to reduce the energy and develop power-efficient applications is very important. Our future work considers energy-efficient software development during the parallelization process. Ongoing work focuses on developing an energy efficiency method to be integrated in our parallelization approach. Energy conservation without performance degradation is challenging and has become an important trend. Our initial results suggest that we can propose energy-efficient task decomposition and programming constructs during the parallelization process. Our preliminary evaluation shows up to 21% improvements of energy consumption after applying our optimizations. Our overarching goal is to improve efficiency while maintaining productivity.

Acknowledgements

I would like to thank all my colleagues and collaborators who contributed to the results described in this chapter. In particular, I would like to thank Ehsan Atoofian (Lakehead University, Canada), Rohit Atre (TU Darmstadt, Germany), Michael Beaumont (RWTH Aachen University, Germany), Daniel Fried (UC Berkeley, USA), Michael Gerndt (TU Munich, Germany), Wolfram Gottschlich (University of Passau, Germany), Kurt Keutzer (UC Berkeley, USA), Nico Koprowski (Daimler AG, Germany), Zhen Li (SAP, Germany), Arya Mazaheri (TU Darmstadt, Germany), Korbinian Molitorisz (Agilent Technologies, Germany), Mohammad Norouzi (TU Darmstadt, Germany), Jochen Schimmel (Karlsruhe Institute of Technology, Germany), Thireshan Jeyakumaran (Lakehead University, Canada), Walter Tichy (Karlsruhe Institute of Technology, Germany), Zia Ul Huda (TU Darmstadt, Germany), Felix Wolf (TU Darmstadt, Germany), Yang Xiao (Lakehead University, Canada), and Bo Zhao (Humboldt University of Berlin).

Author details

Ali Jannesari

Address all correspondence to: jannesari@iastate.edu

Department of Computer Science, Iowa State University, USA

References

[1] Feautrier P. Automatic parallelization in the polytope model. In: The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications. London: Springer-Verlag; 1996. pp. 79-103. [Online]. Available: http://dl.acm.org/citation.cfm?id = 647429.723579

[2] Griebl M, Lengauer C, Wetzel S. Code generation in the polytope model. In: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '98. Washington, DC: IEEE Computer Society; 1998. p. 106. [Online]. Available: http://dl.acm.org/citation.cfm?id=522344.825673

[3] Li Z, Atre R, Huda ZU, Jannesari A, Wolf F. Unveiling parallelization opportunities in sequential programs. Journal of Systems and Software. July 2016;**117**:282-295

[4] Li Z, Jannesari A, Wolf F. An efficient data-dependence profiler for sequential and parallel programs. In: Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Hyderabad, India: IEEE Computer Society; May 2015. pp. 484-493

[5] Atre R, Jannesari A, Wolf F. The basic building blocks of parallel tasks. In: Proceedings of the International Workshop on Code Optimisation for Multi and Many Cores; San Francisco, CA. ACM; February 2015. pp. 1-12

[6] Li Z, Jannesari A, Wolf F. Discovery of potential parallelism in sequential programs. In: Proceedings of the 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI); Lyon, France. October 2013. pp. 1004-1013

[7] Huda ZU, Jannesari A, Wolf F. Using template matching to infer parallel design patterns. ACM Transactions on Architecture and Code Optimization. 21 January 2015;**11**(4):1-64

[8] Huda ZU, Atre R, Jannesari A, Wolf F. Automatic parallel pattern detection in the algorithm structure design space. In Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS); Chicago. IEEE Computer Society; May 2016. pp. 43-52

[9] Zhao B, Li Z, Jannesari A, Wolf F, Wu W. Dependence-based code transformation for coarse-grained parallelism. In: Proceedings of the International Workshop on Code Optimisation for Multi and Many Cores; San Francisco. ACM; February 2015. pp. 1-10

[10] Jannesari A, Wolf F. Automatic generation of unit tests for correlated variables in parallel programs. International Journal of Parallel Programming (IJPP). March 2016;**44**(3):644-662 [Online]. Available: http://dx.doi.org/10.1007/s10766-015-0363-8

[11] Jannesari A. Detection of high-level synchronization anomalies in parallel programs. International Journal of Parallel Programming (IJPP). August 2015;**43**(4):656-678

[12] Mazaheri A, Jannesari A, Mirzaei A, Wolf F. Characterizing loop-level communication patterns in shared memory applications. In: Proceedings of the 44th International Conference on Parallel Processing (ICPP); Beijing, China. September 2015. pp. 759-768

[13] Xiao Y, Li Z, Atoofian E, Jannesari A. Automatic optimization of software transactional memory through linear regression and decision tree. In: Proceedings of 15th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP); Zhangjiajie, China, ser. Lecture Notes in Computer Science, Vol. 9531. Springer International Publishing; November 2015. pp. 61-73

[14] Jeyakumaran T, Atoofian E, Xiao Y, Li Z, Jannesari A. Improving performance of transactional applications through adaptive transactional memory. In: Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP); Heraklion Crete, Greece. February 2016

[15] Rul S, Vandierendonck H, De Bosschere K. A profile-based tool for finding pipeline parallelism in sequential programs. Parallel Computing. September 2010;**36**(9):531-551

[16] Kim M, Kim H, Luk CK. SD3: A scalable approach to dynamic data- dependence profiling. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 43. IEEE Computer Society; 2010. pp. 535-546

[17] Fraser K, Harris T. Concurrent programming without locks. ACM Transactions on Computer System. May 2007;**25**(2). DOI: 10.1145/1233307.1233309

[18] Sanchez D, Yen L, Hill MD, Sankaralingam K. Implementing signatures for transactional memory. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 40. IEEE Computer Society; 2007. pp. 123-133

[19] Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Fa-toohi RA, Frederickson PO, Lasinski TA, Simon HD, Venkatakrishnan V, Weeratunga SK. The NAS parallel benchmarks. The International Journal of Supercomputer Applications. 1991;**5**(3):63-73

[20] Andersch M, Juurlink B, Chi CC. A benchmark suite for evaluating parallel programming models. In: Proceedings of the 24th Workshop on Parallel Systems and, Algorithms, ser. PARS '11. 2011. pp. 7-17

[21] Li Z, Jannesari A, Wolf F. Discovering parallelization opportunities in sequential programs—A closer-to-complete solution. In: Proceedings of the First International Workshop on Software Engineering for Parallel Systems. 2014. pp. 1-10

[22] Luxburg U. A tutorial on spectral clustering. Statistics and Computing. December 2007;**17**(4):395-416 [Online]. Available: http://dx.doi.org/10.1007/ s11222-007-9033-z

[23] Duran A, Teruel X, Ferrer R, Martorell X, Ayguade E. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: Proceedings of the 2009 International Conference on Parallel Processing, ser. ICPP '09. Washington, DC: IEEE Computer Society; 2009. pp. 124-131. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2009.64

[24] Bienia C, Kumar S, Singh JP, Li K. The parsec benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '08. New York: ACM; 2008. pp. 72-81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[25] Li Z, Atre R, Ul-Huda Z, Jannesari A, Wolf F. Discopop: A profiling tool to identify parallelization opportunities. In: Tools for High Performance Computing 2014. Springer International Publishing; August 2015, ch. 3. pp. 37-54

[26] Li Z, Zhao B, Jannesari A, Wolf F. Beyond data parallelism: Identifying parallel tasks in sequential programs. In: Proceedings of 15th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP); Zhangjiajie, China, ser. Lecture Notes in Computer Science, Vol. 9531. Springer International Publishing, November 2015. pp. 569-582

[27] Jahr R, Gerdes M, Ungerer T. A pattern-supported parallelization approach. In: Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores, ser. PMAM '13. New York: ACM; 2013. pp. 53-62. [Online]. Available: http: //doi.acm.org/10.1145/2442992.2442998

[28] Streit K, Doerfert J, Hammacher C, Zeller A, Hack S. Generalized task parallelism. ACM Transactions on Architecture and Code Optimization. April 25, 2015;**12**(1):1-8. [Online]. Available: http://doi.acm.org/10.1145/2723164

[29] http://www.cs.ucla.edu/~pouchet/software/PolyBench/.

[30] http://clang.llvm.org.

[31] Jannesari A, Koprowski N, Schimmel J, Wolf F. Generating classified parallel unit tests. In: Tests and Proofs: Proceedings of the 8th International Conference, TAP 2014, Held as Part of STAF 2014; July 24-25, 2014; York. Springer International Publishing; December 2014. pp. 117-133

[32] Jannesari A, Koprowski N, Schimmel J, Wolf F, Tichy WF. Detecting correlation violations and data races by inferring non-deterministic reads. In: 2013 International Conference on Parallel and Distributed Systems (ICPADS). December 2013. pp. 1-9

[33] Jannesari A, Tichy WF. Library-independent data race detection. IEEE Transactions on Parallel and Distributed Systems (TPDS). 2013;**PP**(99):1-11

[34] Schimmel J, Molitorisz K, Jannesari A, Tichy WF. Automatic generation of parallel unit tests. In: Proceedings of the 8th International Workshop on Automation of Software Test (AST); San Francisco. ACM; May 2013. pp. 40-46

[35] Schimmel J, Molitorisz K, Jannesari A, Tichy WF. Combining unit tests for data race detection. In: Proceedings of 10th IEEE/ACM International Workshop on Automation of Software Test (AST 2015). IEEE; May 2015. pp. 43-47. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819261. 2819275

[36] Jannesari A, Westphal-Furuya M, Tichy WF. Dynamic data race detection for correlated variables. In: Proceedings of the 11th International Conference on Algorithms and architectures for parallel processing—Volume Part I, ser. ICA3PP'11. Berlin, Germany: Springer-Verlag; 2011. pp. 14-26. [Online]. Available: http://dl.acm.org/citation.cfm?id=2075416.2075421

[37] Woo SC, Ohara M, Torrie E, Singh JP, Gupta A. The splash-2 programs: Characterization and methodological considerations. In: Proceedings of the 22Nd Annual International Symposium on Computer Architecture, ser. ISCA '95. New York: ACM; 1995. pp. 24-36. [Online]. Available: http://doi.acm.org/10.1145/223982.223990

[38] Burke M, Cytron R, Ferrante J, Hsieh W. Automatic generation of nested, fork-join parallelism. The Journal of Supercomputing. 1989;**3**(2):71-88 [Online]. Available: http://dx.doi.org/10.1007/BF00129843

[39] Sarkar V. Automatic partitioning of a program dependence graph into parallel tasks. IBM Journal of Research and Development. 1991;**35**(5.6):779-804

[40] Bobbie P. Partitioning programs for parallel execution: A case study in the Intel iPSC/2 environment. International Journal of Mini & Microcomputers. 1997;**19**(2):84-96

[41] Bernstein A. Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers. 1966;**15**(5):757-763

[42] Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '08. New York: ACM; 2008. pp. 101-113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[43] Martinez Caamano JM, Wolff W, Clauss P. Code Bones: Fast and Flexible Code Generation for Dynamic and Speculative Polyhedral Optimization. Cham: Springer International Publishing; 2016. pp. 225-237 [Online]. Available: http://dx.doi.org/10.1007/978-3-319-43659-3_17

[44] Mattson T, Sanders B, Massingill B. Patterns for Parallel Programming. 1st ed. Boston: Addison-Wesley Professional; 2004

[45] Rul S, Vandierendonck H, De Bosschere K. A profile-based tool for finding pipeline parallelism in sequential programs. Parallel Computing. 2010;**36**(9):531-551

[46] Huang J, Jablin TB, Beard SR, Johnson NP, August DI. Automatically exploiting cross-invocation parallelism using runtime information. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE; 2013. 1-11

[47] Garcia S, Jeon D, Louie CM, Taylor MB. Kremlin: Rethinking and rebooting gprof for the multicore age. SIGPLAN Notices. June 2011;**46**(6):458-469. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993553

[48] Zhang X, Navabi A, Jagannathan S. Alchemist: A transparent dependence distance profiling infrastructure. In: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society; 2009. 47-58

[49] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Notices. June 2007;**42**(6):89-100 [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[50] Ketterlin A, Clauss P. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society; 2012. 437-448

[51] Ceng J, Castrillon J, Sheng W, Scharwächter H, Leupers R., Ascheid G, Meyr H, Isshiki T, Kunieda H. Maps: An integrated framework for mpsoc application parallelization. In: Proceedings of the 45th Annual Design Automation Conference, ser. DAC '08. ACM; 2008. pp. 754-759

[52] Subotic V, Ayguadé E, Labarta J, Valero M. Automatic Exploration of Potential Parallelism in Sequential Applications. Cham: Springer International Publishing; 2014. pp. 156-171 [Online]. Available: http://dx.doi.org/10.1007/ 978-3-319-07518-1_10

[53] http://software.intel.com/en-us/intel-advisor-xe.

[54] Tournavitis G, Wang Z, Franke B, O'Boyle MF. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and, Implementation, ser. PLDI '09. ACM; 2009. pp. 177-187. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542496

[55] Tournavitis G, Franke B. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '10. ACM; 2010. 377-388. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854321

[56] Hafiz M, Overbey J, Behrang F, Hall J. Openrefactory/c: An infrastructure for building correct and complex c transformations. In: Proceedings of the 2013 ACM Workshop on Refactoring Tools, ser. WRT '13. ACM; 2013. 1-4. [Online]. Available: http://doi.acm.org/10.1145/2541348.2541349

[57] Quinlan D, Schordan M, Yi Q, de Supinski BR. A c++ infrastructure for automatic introduction and translation of openmp directives. In: OpenMP Shared Memory Parallel Programming. Springer; 2003. pp. 13-25. [Online]. Available: http://dx.doi.org/10.1007/3-540-45009-2_2