Exposing Parallelism and Locality in a Runtime Parallel Optimization Framework*

David A. Penry, Daniel J. Richins, Tyler S. Harris, David Greenland, and Koy D. Rehme
Department of Electrical and Computer Engineering, Brigham Young University
Provo, UT 84602
dpenry@ee.byu.edu, {drichins, th268, david_greenland, kdr32}@byu.net

ABSTRACT

Runtime parallel optimization has been suggested as a means to overcome the difficulties of parallel programming. For runtime parallel optimization to be effective, parallelism and locality that are expressed in the programming model need to be communicated to the runtime system. We suggest that the compiler should expose this information to the runtime using a representation that is independent of the programming model. Such a representation allows a single runtime environment to support many different models and architectures and to perform automatic parallelization optimization.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Performance

1. INTRODUCTION

The widespread use of tens to hundreds of processor cores in commodity systems will require widespread deployment of parallel desktop applications, causing a fundamental change in parallel programming; parallel programming techniques must become mainstream.

Unfortunately, bringing parallel programming into the mainstream faces difficulties:

- For a programmer to write quality parallel programs, she needs to know the details of the target system and how to map the application efficiently to that system. Even using advanced programming models such as transactional memory [4], the task remains daunting, perhaps too daunting for the average programmer.
- A widely-distributed program will have no knowledge of its operating environment until runtime. Such a program cannot be efficiently mapped to a target system during development or compilation.
- While future applications will have sufficient parallelism [3], that parallelism may not be apparent during development or compilation due to irregular data access patterns [5].

*This work was supported by a grant from Microsoft Research through the Safe and Scalable Multicore Computing program.

CF'10, May 17–19, 2010, Bertinoro, Italy.
ACM 978-1-4503-0044-5/10/05.

Runtime parallel optimization systems have the potential to overcome these difficulties: a good runtime would know the details of the system, could map the application to the operating environment, and would be able to analyze actual data accesses [7, 9]. Many such runtime systems have been proposed using a variety of techniques such as parallel data structures [1], autoscheduling [8], autotuning [12], inspector-executor [10], and runtime feedback [11].

However, previous runtime systems have suffered from a severe limitation in their usefulness: they have each supported only a limited selection of parallel programming models. We submit that a practical, deployable runtime parallel optimization system should be programming-model-agnostic. To accomplish this goal, a programming model's expressed and implied parallelism and locality must be conveyed to the runtime via a flexible representation. We call such a representation an *exposed parallelism and locality (EPL) representation*.

In this paper, we introduce four properties of EPL representations: *task-based*, *multi-relational*, *hierarchical*, and *concise*. We argue that these properties allow a runtime system to:

- Easily retain the information from many different programming models; and
- Use this information to perform automatic parallelization optimization.

Using these properties, we have created an optimizing runtime known as ADOPAR. Because it incorporates the four properties described above, ADOPAR's representation can support a variety of programming models. ADOPAR automatically detects and performs parallelization and localization optimizations from the EPL representation.

The primary contribution of this paper is the definition of the properties that an exposed parallelism and locality representation should possess; these properties of the representation enable a programming-model-agnostic runtime parallel optimization system.

2. AN EPL REPRESENTATION

When any application is parallelized, either the programmer or the compiler must in some way represent parallelism and locality. Exposing this information to the runtime would allow a runtime system to optimize for parallel execution. An exposed parallelism and locality (EPL) representation requires four properties to be suitable for runtime parallel optimization. It must be *task-based*, *multi-relational*, *hierarchical*, and *concise*. These properties ensure that the representation carries all the useful information from the programming model and that the information can be easily used for parallel optimization. Each property will be described in turn.

2.1 Property 1: Task-based

An EPL representation must retain information expressed by many different programming models. Therefore, it must offer a universal mechanism for representing parallelism. A task graph is such a universal mechanism.

The representation of a task must identify the work to be performed as well as properties of the task. The set of properties includes at least the task type and an estimated execution time to be used in cost functions for mapping. Possible task types include *choice tasks* [2], *loop iterations*, *function invocations*, *reduction operations*, and *general tasks*.

2.2 Property 2: Multi-relational

Much of the parallelism and locality information implied by a programming model involves relationships such as data flow and exclusivity that must be maintained between portions of the program. Other relationships are simply desirable, e.g. locality. An EPL representation needs to represent all of these relationships; we call this property *multi-relational*. Relationships are edges in the task graph.

We have identified several types of relationships that must be supported: data dependence, ordering, exclusiveness, sharing, and nearness. The combination of multiple types of relations with tasks allows a wide variety of programming models to be easily mapped to an EPL representation.

2.3 Property 3: Hierarchical

An EPL representation needs to retain information from the programming model that can help with parallel optimization. One element of many programming models is hierarchy. Well-known optimization techniques that depend upon hierarchy include loop splitting, loop interchange, and loop fusion with multiple sub-tasks.

2.4 Property 4: Concise

While individual tasks and relationships are a useful and general way to represent parallelism and locality, regularity that was present in the programming model becomes obscured and hard to exploit in a task graph. Therefore, while an EPL representation should be task-based and multi-relational, it should represent tasks and relationships concisely as sets of tasks and relationships that are similar to each other. We call these concise representations *descriptors*.

A task descriptor points to two functions: a *task body* and a *task instantiation function*. The task body is the code that the task must run. The task instantiation function, if called at runtime, would generate task instances. A relationship descriptor points to a *relationship instantiation function* that, if called at runtime, would generate relationship instances.

A functional representation of tasks and relationships is very powerful for several reasons: it is general, dynamic, exact, easily automated, and can be optimized and evaluated using standard compiler techniques.

Note that while the instantiation functions could be called at runtime to generate a task graph, we do *not* see instantiation as the primary mode of operation of a runtime system. Instead, parallel optimization techniques would analyze the instantiation functions and task bodies and make choices about how to generate parallel code. Some examples of such optimization techniques might be:

- Recognition and exploitation of DOALL loops
- Specialized hardware selection
- Dynamic task scheduling
- Helper threading
- Thread-level speculation
- Lock insertion or software transactional memory

3. ADOPAR

We are currently implementing a general runtime parallel optimization infrastructure that uses an EPL representation. This infrastructure is called ADOPAR, for *ADaptive Online PARallel optimization* [7]. Developers write applications in a variety of programming languages and models, making no assumptions about the target system. The compilers perform parallelism and locality discovery and produce a binary augmented with exposed parallelism and locality information. The runtime system uses the EPL, runtime application data, and online performance feedback to map the application to the architecture and available resources.

ADOPAR can be seen as a generalization of the inspector-executor model of parallelization. ADOPAR uses the LLVM compiler framework [6] as its underlying compilation infrastructure. EPL is represented in the low-level IR through functions and hooks.

4. CONCLUSION

Parallel programming has heretofore remained the domain of highly-skilled programmers with extensive experience. With the advent of commodity multiprocessors, it is becoming increasingly important to enable the majority of programmers to participate in parallel programming. The EPL representation begins to surmount the difficulties that have precluded the acceptance of parallel programming as a mainstream technique. It does so by removing the need for the programmer to program to a specific architecture, allowing a runtime to actively adapt a program to its running environment, and feeding information regarding actual data access patterns to the runtime for online parallelization.

5. REFERENCES

[1] G. Agrawal, A. Sussman, and J. Saltz. Compiler and runtime support for structured and block structured applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing'93*, pages 578–587, 1993.

[2] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, 2007.

[3] Y.-K. Chen, J. Chhugani, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, and M. Smelyanskiy. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE*, 96(5):790–807, May 2008.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[5] K. Kennedy. Compiler technology for machine-independent parallel programming. *International Journal of Parallel Programming*, 22(1):79–98, Feb 1994.

[6] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[7] D. A. Penry. Multicore diversity: A software developer's nightmare. *Operating Systems Review*, 43(2):100–101, Apr 2009.

[8] C. D. Polychronopoulos. Toward auto-scheduling compilers. *Journal of Supercomputing*, 2:297–330, 1988.

[9] L. Rauchwerger. Run-time parallelization: Its time has come. *Parallel Computing*, 24(3–4):527–556, July 1998.

[10] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[11] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the 13th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 277–286, 2008.

[12] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.