Parallel Pattern Detection for Architectural Improvements

Jason A. Poovey
Georgia Institute of Technology
japoovey@gatech.edu

Brian Railing
Georgia Institute of Technology
brian.railing@gatech.edu

Thomas M. Conte
Georgia Institute of Technology
conte@gatech.edu

Abstract

With the shift in general purpose computing to increasingly parallel architectures comes a need for clever architectures to achieve high parallelism on previously sequential or poorly parallelized code. In order to fully utilize the many-core systems of the present and future, a shift must occur in architecture design philosophy to understanding how the parallel programming process affects design decisions.

Parallel patterns provide a way to create parallel code for a wide variety of algorithms. Additionally they provide a convenient classification mechanism that is both understandable to programmers and that exhibit similar behaviors that can be architecturally exploited. In this work we explore the capabilities of pattern driven dynamic architectures as well as detection mechanisms useful for dynamic and static parallel pattern recognition.

1. Introduction

The past decade has brought about a major shift in the primary architecture for general purpose computing. As single-threaded performance becomes increasingly more power intensive, there is a significant shift towards multi-/many- core architectures. These architectures provide the benefit of lower power with high performance for highly parallel applications. However, in order to fully utilize many cores, a shift must occur in architecture design philosophy through new programming paradigms and an increased understanding of the parallel program behavior.

Classifying parallel code in a way that is both understandable to the programmer and useful to low-level designers is a non-trivial task. Recent works in parallel programming patterns [1-3] provide great potential for program classifications that maintain common behaviors exploitable by architects. These patterns span many layers of the design process [1]; however, the algorithmic level patterns provide a good median between the program structure and its low level behaviors. Algorithmic parallel patterns refer to the basic code structure in terms of sharing behavior and thread behavior. For example, many scientific applications such as n-body simulations employ the pattern of geometric decomposition which takes a set of data and splits it up among processing threads. These threads still communicate but sparingly enough to allow speedup from the parallelization.

As parallel programming becomes more well-understood, the method of programming via patterns is becoming increasingly popular [4, 5]. Therefore, it is highly reasonable that parallel programmers will have natural *insights* into the algorithmic structure of their code. Moreover, architects can leverage the properties of these patterns to create adaptable architectures tuned to the various patterns. For example, in a pipeline pattern, data migrates between stages, as opposed to an embarrassingly parallel problem where data remains private to each thread. Therefore, in the case of a pipeline pattern, the architecture with knowledge that the program is using a pipeline may choose to perform optimizations such as running the program on a streaming processor or using a more optimal coherence protocol such as MI.

In this work we will lay the groundwork for a novel method for detecting parallel patterns statically through the use of programmer "insights" as well as dynamically through run-time performance counters. Our results will show that for a set of micro-benchmarks our detection mechanisms are successful in finding patterns in un-annotated parallel code. These results will demonstrate that our current set of metrics provide potential for a robust pattern detection system that could drive many architectural optimizations.

2. Algorithmic Parallel Patterns

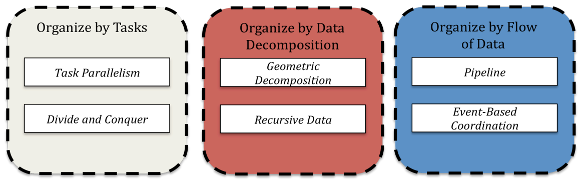


Figure 1 - Algorithmic Parallel Patterns

Several efforts have been made to standardize parallel programming patterns [1, 2]. These efforts codify the standards and characteristics in a manner similar to the programming patterns used in the software engineering community for object-oriented programming. What these standards reveal are six main patterns, defined in [1] and are shown in Figure 1, each with unique architectural characteristics to exploit.

The parallel programming patterns are grouped based on the type of conceptual parallelization performed. When the problem consists of a group of independent tasks or task groups to be run in parallel, the parallel pattern employed is task parallelism. This class of problems has also been generally referred to as embarrassingly parallel. When there is a problem task that naturally subdivides into several smaller tasks that can be done in parallel, then the divide and conquer pattern is applied. Divide and Conquer splits tasks until a work "threshold" is met and the subtask works on a subset of the data serially.

Many parallel problems are solved through the decomposition of data by creating threads to work on the data in parallel. The two standard patterns for data parallelization are geometric decomposition and recursive data. The geometric decomposition pattern operates on data in a regular structure, such as an array, that is split into sub-structures operated on in parallel. This pattern is typically characterized by sharing between threads, particularly threads with neighboring data. If the data is not in a regular structure, but rather a structure such as a graph, data decomposition parallelization is done via the recursive data pattern. This pattern creates parallelism by doing redundant work to decrease communication between threads. For example, an algorithm to find every node's root requires a full graph traversal. A recursive data approach would create a thread for every node in the graph and perform a graph-climbing algorithm independently for each node. This causes some nodes' depths to be calculated more than once, but has performance gains due to the enhanced parallelism.

As programmers continue to shift legacy sequential code to a parallel domain, an increasingly common parallel pattern used is the pipeline pattern [4, 6]. This pattern is performed by taking a flow of data through tasks and splitting it into pipeline stages. The parallelism is achieved by keeping all stages full with data such that each stage can operate simultaneously. Moreover, not all pipeline parallel workloads are completely feed-forward pipelines. Simulations, such as discrete event simulations, leverage the pipeline pattern but with more complex interactions between stages. In these cases the program is classified as an event-based coordination pattern.

3. Architectural Implications of Parallel Patterns

Understanding the parallel pattern used in an application can be very useful for performance improvement. Although the algorithms and details of programs of the same pattern may differ, each pattern has some unique behaviors that have unique architectural implications.

Recently, thread scheduling and balancing has become an increasingly hot topic in architecture conferences [7-11]. The main research thrust is to estimate thread criticality and either provide extra resources to the critical thread or save power by reducing resources for non-critical threads. In [12] we showed how patterns exhibit unique thread behaviors that lend themselves to different criticality metrics. For example, pipeline and divide and conquer are typically imbalanced due to thread complexity differences. However, task parallel and geometric decomposition do not have much imbalance except through a minor contribution due to a non-uniform memory hierarchy.

In addition to thread balancing, patterns can be used to guide coherence protocol design and network design. In patterns with a heavy amount of migratory data, such as the pipeline pattern, it is likely to be more beneficial to use a simpler protocol such as MI to reduce message counts. The reason is that as data moves through the parallel pipeline it may be first read by the next pipeline stage then written which would require coherence messages to transition the block to S then M in an M(O)(E)SI protocol. Other recent works have proposed asymmetric networks that have varying buffer widths throughout the chip [13]. Patterns with high degrees of inter-thread communication such as geometric decomposition would benefit more from intelligent thread placement such as placing high communication threads on the higher bandwidth nodes.

Finally, heterogeneous architectures are an increasingly popular architectural paradigm [7, 9, 14-16]. Some techniques such as thread balancing target single-ISA asymmetry. But in heterogeneous architectures there may exist multiple architectures and multiple ISAs. By understanding the pattern of the running application, one could conceivably optimize a compiler or run-time to place the workload on the optimal architecture. For example, task parallel workloads with many threads would be more suitable for a GPGPU; whereas a pipeline would be better served on a streaming processor.

4. Parallel Pattern Detection Techniques

As has been shown, parallel pattern detection enables many architectural optimizations. This section will define some metrics of interest that are useful for parallel pattern detection. Solutions for dynamic and static collection and analysis are also explored.

4.1. Metrics of Interest

Sharing patterns have been used in past studies for hardware optimization [17]. Additionally, parallel design patterns each tend to use a unique set of sharing patterns. This is both useful for detection and optimization. Researchers have defined many classes of sharing patterns; however, in this study we focus primarily on read-only, producer/consumer, migratory, and private. **Figure 2** illustrates the parallel pattern tendencies for each sharing pattern.

	Task Parallelism	Divide and Conquer	Geometric Decomposition	Recursive Data	Pipeline	Event-based Coordination
Read-Only	★★★		★	★★★		
Migratory	★★	★★★	★★		★★★★	★★★★
Producer/ Consumer		★★	★★★★			
Private	★★★★	★	★	★★★★	★	★

Figure 2 - Parallel Pattern Sharing Behaviors

Private data is common to patterns without much sharing, such as task parallel and recursive data. In a task parallel application the algorithm is typically *embarrassingly parallel,* which means each thread is completely independent. Recursive data reduces sharing by performing redundant work in each thread. Migratory data is common to pipeline and event based coordination. In these patterns, threads are pinned to various stages in a process and data migrates between threads. Finally, producer / consumer sharing is common in patterns with widely shared data such as geometric decomposition.

For pattern detection, sharing behavior is observed for each address in the system. On a "write permission" request, the system records the address as being produced by a thread, a "read permission" is recorded as a consumption. The classification rules for each address are shown in Figure 3.

In addition to sharing behavior, patterns are also defined by their thread behavior. Patterns such as divide and conquer or recursive data have unique thread behaviors that provide a signature pattern. Divide and conquer is characterized by a rising number of threads during the divide phase, some period of leveling for computation, followed by a declining phase as the results are joined. Recursive data, however, begins with many threads and slowly ramps down as the different threads finish execution.

- If #consumers == #producers == 1 then Private
- If #consumers > #producers then Producer/Consumer
- If #consumers <= #producers then Migratory

Figure 3 - Sharing Pattern Classification Rules for each Cache Line

In order to measure thread behavior we monitor thread creation and suspension/exit events. Over the course of a phase or program the average absolute slope of the active threads over time is calculated. In addition the number of rising and falling *events* are counted and added to the evaluation.

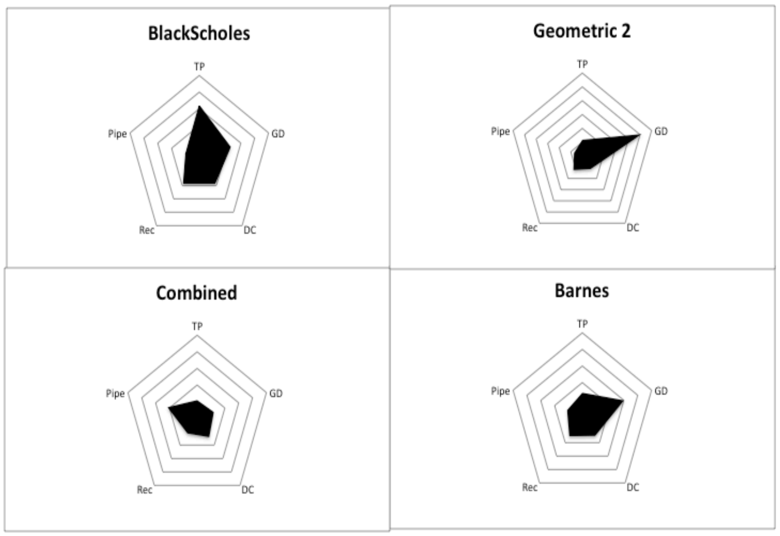
In addition to thread spawn/exit behavior, thread imbalance is unique between patterns. For example, geometric decomposition is typically SIMD or SPMD programming that is more balanced than a recursive data pattern. To measure the imbalance, the number of dynamic instructions per thread is measured to calculate the average number of instructions per thread. If most of the threads have dynamic instruction counts within one standard deviation of the mean, the workload is considered balanced. Otherwise it is considered unbalanced.

The final metric evaluated in this initial pattern detection scheme is the PC uniqueness between threads. This test looks at whether a SPMD or SIMD style programming method was used. The PCs accessed by each thread are counted, and if most PCs are unique this suggests a non-SPMD style pattern such as pipeline. This is measured by measuring for each PC accessed how many threads used that PC. Patterns such as pipeline will have unique PCs for each thread, whereas geometric decomposition will share PCs with most threads.

4.2. Online Detection Mechanisms

The analysis and metrics in Section 4.1 have been collected using a detailed architecture simulator. Some of the data such as a global count of all PCs and a count of producers/consumers for all addresses would be very costly to directly implement in hardware. In this section we will discuss some ideas for how these metrics could be translated into real hardware mechanisms.

Figure 4 - Results of Parallel Pattern Detection



Since sharing behavior is measured using permission requests via the coherence protocol a logical place to measure the producer/consumer relationship is at the directory. For a 32-core machine, 6 bits per entry could be added to maintain the producer/consumer count.[1] The information would be lost on evicted entries, but as long as the working set largely remains on chip, a directory only measurement would provide accurate results.

Thread behavior metrics require a repository of information on thread spawn/exit and balance information. Rather than providing hardware counters an alternative is to instrument the threading code such as pthread_create() in order to monitor this behavior through a run-time monitor. The run-time monitor will then collect all metrics and provide feedback to the hardware on optimizations to enable or disable based upon the pattern detected. Additionally, in future research, the monitor will include the ability to detect program phases so as to reset the pattern detection process.

4.3. Program Invariants for Detection

Understanding the pattern and performance implications cannot be confined to the knowledge available at runtime, as some of the information may be too difficult to detect or otherwise obscured by program behaviors. By extending our scope of usable inputs to include the program's source itself, we can consider the insights that are lost between source and execution. Thus, insights can provide a static mechanism to enhance parallel pattern detection.

Programmer insights are notionally a combination of asserts, pragmas, and invariants. By encompassing these distinct notations in the term "insight", we are able to leverage both the properties that must be true as well as the properties that may be true. The intention is not an attempt to statically analyze the raw code, but rather leverage what the programmer says the code should be doing. This again allows two usages: first, to verify that the application is behaving how the programmer intended. And second, to further optimize the final application by using the insights for pattern detection.

In this work, parallel insights are split into four categories: data sharing, thread communication, general threading and synchronization / concurrency. The first three strongly relate to the parallel pattern metrics in Section 4.1 and can aid in the detection and implementation of patterns. The last category is a look at some insights from past work focused on concurrency bugs that could possibly be leveraged for pattern detection.

In parallel programs, the developer has insight into how the data is intended to be shared. Providing this insight to the compiler / runtime is therefore reasonable and in fact some of these notations are already present in existing frameworks. We consider four sharing types: read-only, migratory, shared, and private. With these, a programmer can express how individual allocations are used within the parallel patterns.

OpenMP [18] provides notations for some of these types with data sharing attribute clauses that can specify data as shared or private. However, the OpenMP clauses do not extend to general data allocations. We propose extending variable

[1] To maintain a range of -32 to +32

classification insights to include specifying data as migratory or read-only.

Data sharing also relates to communication patterns between threads. So beyond determining what data are shared, we can also have insight as to with whom each datum is shared. While a programmer may have trouble expressing exactly which threads will be in communication, sometimes the degree of sharing is clear. Overall, the level of communication may be of several categories: none (terminal thread or minimal communication), few (algorithmically defined), and many (data dependent). Furthermore, knowing the direction of communication is also valuable, whether it be in, out, or bidirectional. These insights could be as simple as specifying for a variable not only if it is shared, but with how many threads and through labels specifying thread ids. Also when specifying a shared variable, certain threads can be labeled as receivers or senders for that variable.

As discussed earlier, another aspect of pattern detection is measuring the number of active threads over time. This can be done by statically analyzing insights such as thread creation or join (i.e. pthread_create() or pthread_join()). Additionally, if insights are made available to the programmer to specify task interaction, a static task graph can be created and analyzed for thread behavior. Combining all of these proposed static insights with the dynamic approaches enhances the accuracy of overall pattern detection.

Past work on parallel insights has focused on finding and addressing potential concurrency bugs. While the focus of this paper has been on the performance of correct algorithms, these works are nonetheless related in a notational sense. In [19], Burnim et al., proposes deterministic sections that are then checked by the run-time for whether different thread interleavings have different results. DeFuse [20] proposes several types of determinism invariants that are leveraged for finding concurrency bugs.

One trade-off in concurrency is between different lock types. In past experience, a programmer would potentially instrument a lock or just "know" the level of contention and types of access required for an individual element of shared data. However, as the platform changes (architecture, core count, etc.), the optimal lock type might also change. Therefore, the compiler / runtime can also benefit from a series of common lock insights like level of contention, types of access (reader versus writer), and relation to other critical sections. Future work will investigate leveraging existing insights to also better understand the relationship to pattern behavior.

5. Experiments

To show how pattern detection can be effective, experiments have been run to investigate detection accuracy. First a set of representative microbenchmarks was created that are considered "golden copies" of the pattern behavior. These benchmarks are short programs that consist of the basic structure of the parallel pattern. Simulations were run to collect the metrics discussed in Section 4.1 for these microbenchmarks. What was found was a set of diverse characteristics for each benchmark. For example pipeline was characterized by using largely migratory data, whereas geometric decomposition uses producer/consumer relationships. Divide and Conquer was unique mostly in thread behavior such as its spawn/exit events.

After collection of results for the microbenchmarks, some real benchmarks were evaluated to detect their parallel pattern. The same metrics from Section 4.1 were collected and a weighted comparison to the results of the microbenchmarks was performed. Repeated experiments were used to determine the weightings.[2] Figure 4 illustrates the results of the weighted comparisons.

In this figure a rating is achieved for each of the benchmarks similarity to each pattern. Blackscholes from the PARSEC suite [5] is shown to be mostly task parallel. This coincides with what was expected. Blackscholes simply performs stock option pricing on many independent stocks in parallel, which is task parallel. A second geometric decomposition workload written by another programmer with another algorithm was detected easily as the correct pattern. Barnes from the SPLASH-2 benchmark set [21] is a geometrically decomposed n-body simulation using the Barnes-Hut algorithm. Our detection mechanism is able to correctly find the geometric decomposition in this workload. Future work such as extending the use of static analysis discussed in Section 4.3 will assist in detection for the *Combined* benchmark. In this benchmark a pipeline algorithm was combined with a divide and conquer which resulted in a mixed result. Through the use of phase detection and boundaries from static and dynamic analysis, these results will improve with future work.

[2] Due to space limitations the full list of weightings is not included but can be added as an appendix to the accepted paper.

References

[1] T. G. S. Mattson, Beverly A.; Massingill, Berna L., Patterns for Parallel Programming: Addison-Wesley, 2004.

[2] J. L. Ortega-Arjona, Patterns for Parallel Software Design. West Sussex: Wiley, 2010.

[3] K. Asanovic, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, Berkeley, CA, Technical ReportDec 2006.

[4] M. A. Suleman, et al., "Feedback-directed pipeline parallelism," presented at the Proceedings of the 19th international conference on Parallel architectures and compilation techniques, Vienna, Austria, 2010.

[5] C. Bienia, et al., "The PARSEC benchmark suite: characterization and architectural implications," presented at the Proceedings of the 17th international conference on Parallel architectures and compilation techniques, Toronto, Ontario, Canada, 2008.

[6] W. Thies, et al., "A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs," presented at the Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007.

[7] J. C. Saez, et al., "A comprehensive scheduler for asymmetric multicore systems," presented at the Proceedings of the 5th European conference on Computer systems, Paris, France, 2010.

[8] T. Li, et al., "Efficient operating system scheduling for performance-asymmetric multi-core architectures," presented at the Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, Nevada, 2007.

[9] N. B. Lakshminarayana, et al., "Age based scheduling for asymmetric multiprocessors," presented at the Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, 2009.

[10] L. De Giusti, et al., "AMTHA: An Algorithm for Automatically Mapping Tasks to Processors in Heterogeneous Multiprocessor Architectures," in Computer Science and Information Engineering, 2009 WRI World Congress on, 2009, pp. 481-485.

[11] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," presented at the Proceedings of the 36th annual international symposium on Computer architecture, Austin, TX, USA, 2009.

[12] J. A. Poovey, Rosier, Michael C., Conte, Thomas M., "Pattern-Aware Dynamic Thread Mapping Mechanisms for Asymmetric Manycore Architectures -- Technical Report No. 2011-1," School of Computer Science, Georgia Institute of Technology2011.

[13] Z. Guz, et al., "Efficient link capacity and QoS design for network-on-chip," presented at the Proceedings of the conference on Design, automation and test in Europe: Proceedings, Munich, Germany, 2006.

[14] M. Pericas, et al., "A Flexible Heterogeneous Multi-Core Architecture," presented at the Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, 2007.

[15] R. J. O. Figueiredo and J. A. B. Fortes, "Impact of heterogeneity on DSM performance," in High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on, 2000, pp. 26-35.

[16] L. Chin and L. Sau-Ming, "An adaptive load balancing algorithm for heterogeneous distributed systems with multiple task classes," in Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on, 1996, pp. 629-636.

[17] J. K. Bennett, et al., "Munin: distributed shared memory based on type-specific memory coherence," SIGPLAN Not., vol. 25, pp. 168-176, 1990.

[18] (2008, OpenMP Application Program Interface v3.0. Available: http://www.openmp.org/mp-documents/spec30.pdf

[19] J. Burnim and K. Sen, "Asserting and checking determinism for multithreaded programs," Commun. ACM, vol. 53, pp. 97-105, 2009.

[20] Y. Shi, et al., "Do I use the wrong definition?: DeFuse: definition-use invariants for detecting concurrency and sequential bugs," presented at the Proceedings of the ACM international conference on Object oriented programming systems languages and applications, Reno/Tahoe, Nevada, USA, 2010.

[21] S. C. Woo, et al., "The SPLASH-2 programs: characterization and methodological considerations," SIGARCH Comput. Archit. News, vol. 23, pp. 24-36, 1995.