# Coarse-Grain Speculation for Emerging Processors

Hari K. Pyla
Virginia Polytechnic Institute and State University
harip@cs.vt.edu

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

***General Terms*** Algorithms, Design, Languages, Measurement and Performance

***Keywords*** Speculative Parallelism, Coarse-grain Speculation, Program Analysis, Concurrent Programming and Runtime Systems

## 1. Research Problem

The impending multi/many-core processor revolution requires that programmers leverage explicit concurrency to improve performance. Unfortunately, a large body of applications/algorithms are inherently hard to parallelize due to execution order constraints imposed by data and control dependencies or being sensitive to their input data and not scale perfectly, leaving several cores idle. *The goal of this research is to enable such applications leverage multi/many-cores efficiently to improve their performance.*

## 2. Motivation

The important application domain that will benefit from our approach are multiple equivalent algorithms whose performance differ depending on input data. For example, graph coloring is widely used in domains such as job scheduling, bandwidth allocation, pattern matching, and compiler optimization (register allocation). Existing approaches to this problem rely on probabilistic and meta-heuristic techniques, whose performance varies widely with the input parameters such as the graph's topology and number of colors. In addition to input sensitivity, graph coloring algorithms are hard to parallelize due to inherent data dependencies.

As another example, consider partial differential equations (PDEs) solvers in large scale simulations for computational science and engineering applications such as fluid dynamics, weather and climate modeling, structural analysis, and computational geosciences. The large, sparse linear systems of algebraic equations use preconditioned iterative methods, whose performance varies widely from problem to problem, even for related problem sequences (e.g., discrete time steps in a time-dependent simulation). Unfortunately, the best iterative method is not known a priori.

Yet another example are combinatorial problems including sorting, searching, permutations and partitions with well-known theoretical algorithmic bounds, but whose runtime depends on a variety of factors,s including the amount of input data (algorithmic bounds assume asymptotic behavior), the sortedness of the input data, and cache locality of the implementation [1].

## 3. Approach

This work equips programmers with a powerful tool for exploiting parallelism by means of *coarse-grain speculation*. Speculative execution at coarse granularities (e.g., code-blocks, methods, algorithms) offers a promising programming model for exploiting parallelism for many hard-to-parallelize applications.

Our programming model can express computation at any granularity, so that any application unit can be executed speculatively. Although the idea of coarse-grain "speculative execution" is relatively straightforward, its efficient implementation is strewn with challenges. Shared memory parallel programs are difficult to implement correctly, and so is detecting concurrency bugs (e.g., data races, deadlocks, order violations, atomicity violations) [5]. Hence, the programmer must not be burdened with using the low level threading primitives to create speculative control flows, manage rollbacks, and recover in the event of mis-speculations.

We present a simple speculative programming framework, Anumita (*guess* in Sanskrit), in which coarse-grain speculative code blocks execute concurrently, but the results from only a single speculation modify the program state. Anumita is implemented as a shared library that exposes APIs for common type-unsafe languages including C, C++ and Fortran. Its runtime system transparently (a) creates, instan-

*SPLASH'11 Companion,* October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

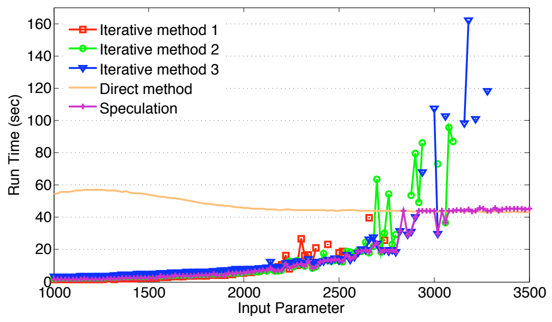

**Figure 1.** PDE solvers (iterative and direct); Y-axis: time-to-solution; X-axis: input parameter. Anumita consistently matches the fastest method for each problem.

tiates, and destroys speculative control flows, (b) performs name-space isolation, (c) tracks data accesses for each speculation, (d) commits the memory updates of successful speculations, and (e) recovers from memory side-effects of any mis-predictions.

Anumita associates each speculation flow's (e.g., an instance of a code block or a function) memory accesses in a speculation composition (loosely, a collection of possible code blocks that execute concurrently) and localizes them, isolating speculation flows through privatization of address space. Ultimately, a single speculation flow within a composition is allowed to modify the program state. We present well-defined semantics that ensures program correctness for propagating the memory updates. Anumita supports a wide range of applications by providing expressive evaluation criteria for speculative execution that go beyond *time to solution* to include arbitrary *quality of solution* criteria. Anumita simplifies speculative parallelism and relieves the programmer from the subtleties of concurrent programming.

Using Anumita requires minimal modifications (8-10 lines on average) to application source code. Additionally, the speculation-aware runtime manages memory and collects garbage from failed speculations. In the context of high-performance computing, with the prevalent OpenMP threading model, Anumita naturally extends speculation to an OpenMP context through a pragma.

## 4. Background and Related Work

Speculative execution is used in a variety of contexts to improve performance, including low level fine-grain speculation in hardware and compilation (e.g., branch prediction, prefetching). Software transaction systems rely on optimistic concurrency. In contrast to existing systems [1–4] Anumita neither relies on value speculation nor employs optimistic concurrency to achieve parallelism. It does not require annotating any variables nor rely on binary instrumentation or collect traces. Anumita introduces the notion of a non-deterministic choice operator to imperative programming.

## 5. Experimental Results

We evaluated Anumati using three real applications: a multi-algorithmic PDE solving framework, a graph (vertex) coloring problem, and a suite of sorting algorithms. Our experimental results indicate that Anumita is capable of significantly improving the performance of hard-to-parallelize and input sensitive applications by leveraging speculative parallelism. For instance, for the PDE solver (Figure 1) the speedup ranged from 0.84-36.19, for the graph coloring problem it ranged from 0.95-7.33, and for the sort benchmark it ranged from 0.84-62.95. With Anumita, it is possible to obtain the best solution among multiple heuristics. In some cases of heuristics failing to arrive at a solution, speculation guaranteed not only a solution but also the one that is nearly as fast as the fastest alternative. We are currently working on improving the performance of Anumita through program analysis and provide support for I/O.

## 6. Contributions

We presented Anumita, a language-independent runtime system for exploiting coarse-grain speculative parallelism in hard to parallelize and/or highly input sensitive applications –*an increasingly important problem in this multi/many-core era*. Our goal is to make speculation a first class parallelization method in such applications. Our research efforts aim at helping adapt and sustain the increasing core counts.

Additionally, Anumita's language transparency and simplicity, will relieve the programmer from the subtleties of concurrent programming and help enable its use with a wide variety of programming languages including C, C++ and Fortran. To our knowledge, Anumita is the first system to provide support for exploiting coarse-grain speculative parallelism in OpenMP based applications. Finally, using real applications we show how our proposed programming constructs achieve significant speedup without sacrificing performance, portability and usability.

## References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *PLDI '09*, pages 38–49, 2009.

[2] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA '09*, pages 81–96. ACM, 2009.

[3] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. In *PLDI '07*, volume 42, pages 223–234.

[4] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe Programmable Speculative Parallelism. In *PLDI '10*, volume 45, pages 50–61, 2010.

[5] W. Zhang, C. Sun, and S. Lu. Conmem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *ASPLOS '10*, spages 179–192, 2010.