Efficient Parallel Execution for "Un-parallelizable" Codes via Coarse-Grain Speculation

Hari K. Pyla
Virginia Tech
harip@vt.edu

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

1. Research Problem

As the number of cores in modern processor architectures keeps growing, programmers must use explicit parallelism to improve performance. Alas, a large body of extant codes are intrinsically unsuitable for mainstream parallelization techniques, due to the execution order constraints imposed by their data and control dependencies. Therefore, realizing the very potential of many-core hinges on our ability to parallelize these so called un-parallelizable codes. *This research solves the challenge of enabling efficient parallel execution of such applications*.

2. Motivation

Multiple application domains posses this dilemma; how to choose the right algorithm when the algorithm's performance is dependent on input data. Graph coloring illustrates this problem. This algorithm underlies forms the foundation of diverse domains including job scheduling, bandwidth allocation, pattern matching, and compiler optimization (register allocation). Several state-of-the-art approaches that solve this problem employ probabilistic and meta-heuristic techniques. The performance of these techniques vary widely with the input parameters including nature of the graph, number of colors, etc. In addition to this sensitivity to the input, algorithms for the graph coloring problem are hard to parallelize due to inherent data dependancies.

Another example is the numerical solution of partial differential equations (PDEs). PDE solvers are a dominant component of large scale simulations arising in 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 are usually solved using preconditioned iterative methods. Unfortunately, the performance of such solvers can vary widely from problem to problem, even for a sequence of problems that may be related in some way, e.g., problems corresponding to discrete time steps in a time-dependent simulation. The problem is that the best iterative method is not known a priori.

Similar examples can be found in widely used combinatorial problems including sorting, searching, permutations and partitions where theoretical algorithmic bounds are well known, but in practice the runtime of an algorithm depends on a variety of factors including the amount of input data (algorithmic bounds assume asymptotic behavior), the sortedness of the input data, and cache locality of the implementation [3].

3. Overview

We present a framework, Anumita (*guess* in Sanskrit) that exploits speculative parallelism to improve the performance of such otherwise hard-to-parallelize applications on multi/manycore architectures. Anumita provides a simple programming interface to express at any arbitrary granularity (e.g., code-blocks, methods, algorithms), the parts of an application that may be executed speculatively.

While the notion of coarse-grain "speculative execution" is relatively straightforward, there are several challenges in the details. Writing correct shared memory parallel programs is a challenging task in itself, and detecting concurrency bugs (e.g., data races, deadlocks, order violations, atomicity violations) is an extremely difficult problem.

Anumita simplifies the notion of speculative parallelism and relieves the programmer from the subtleties of concurrent programming. Anumita consists of a shared library, which implements the framework API for type-unsafe languages including C, C++ and Fortran, and a user-level runtime system that transparently (a) creates, instantiates, and destroys speculative control flows, (b) performs name-space isolation, (c) tracks data accesses for each speculation, (d) committs the memory updates of successful speculations, (e) recovers from memory side-effects of any mis-predictions, and, (f) performs speculation-aware memory management and garbage collection from failed speculations.

In the context of high-performance computing applications, where the OpenMP threading model is widely prevalent, Anumita also provides a new OpenMP pragma to naturally extend speculation into an OpenMP context. *To our knowledge, Anumita is first to introduce the notion of coarse-grain speculation in OpenMP*.

Anumita presents well-defined semantics that ensure program correctness for propagating the memory updates. Anumita is designed to support a wide range of applications (both sequential and parallel) by providing expressive evaluation criteria for speculative execution that go beyond *time to solution* to include arbitrary *quality of solution* criteria. Anumita is implemented as a language independent runtime system and its use requires minimal (around 8-10 lines) modifications to existing application source code. These modifications were short and required little to no understanding of the applications themselves. Another key feature of Anumita is its ability to achieve performance without sacrificing portability and usability. Anumita does not require any modifications to the compiler or the operating system.

We evaluated Anumati using three real applications: a multi-algorithmic PDE solving framework [22], a graph (vertex) coloring problem [16] and a suite of sorting algorithms [26], and several micro benchmarks. Our experimental results demonstrate that Anumita (a) improves the performance of these applications (b) achieves significant speedup over statically chosen alternatives with modest overhead, and is (c) robust in the presence of performance variations or failure.

4. Background and Related Work

None of the existing approaches relying on speculation to achieve parallelism deliver all the desired elements (portability, scalability, usability, and efficiency) required to achieve wide-spread adoption. Anumita comprehensively solves this problem by exploiting coarse-grain speculative parallelism.

Fine-grain Speculation: Hardware and compiler (e.g., branch prediction, prefetching), employ low level fine-grain speculation, relying on value speculation ([18]) to achieve speculative parallelism. At a much coarser granularity, we have —loops. Loop level models [9, 15, 20, 21, 25, 27] achieve parallelism in sequential programs by speculatively executing iterations within loops. Alternatively, several compiler directed frameworks provide support for speculation at the granularity of loops. While most of such models provide transparency, their scope is limited to loops, thus, restricting the amount of parallelism that can be effectively exploited. Additionally, such compile time techniques are suited for applications whose performance is highly input data dependent (known only at runtime).

Coarse-grain Speculation: Software transaction memory systems are premised on optimistic speculative execution of potentially coarse-grain code blocks. Unfortunately, such systems require annotations to variables, special language support, and, memory allocation primitives. Furthermore, fine-grain privatization of updates within transactions is typically achieved by instrumenting load and store operations, which can result in significant impact on application speedup.

In contrast, Anumita does not rely on either value speculation or employ optimistic concurrency to achieve parallelism. Anumita does not require special compilers or rely on program/binary instrumentation or collect runtime program traces. Anumita introduces the notion of a non-deterministic choice operator to imperative programming.

Additionally, in contrast to Ding et al. [12]'s behavior oriented parallelization (BOP) and Kelsey et al. [14]'s Fast Track and Praun et al. [30]'s IPOT, Anumita does not allocate each shared variable in a separate page, or uses a value-based checking algorithm to validate speculation. Furthermore, Anumita does not em-

```
speculation_t *spec_context;
int num_specs=2, rank, value=0;

/* initialize speculation context */
spec_context = init_speculation();

/* begin speculative context */
begin_speculation (spec_context, num_specs, 0);

/* get rank for a speculation */
rank = get_rank (spec_context);

switch (rank)
{
  case 0:
    estimation (...);
    break;

  case 1:
    monte-carlo (...);
    break;

  default:
    printf ("invalid rank\n");
    break;
}

/* commit the speculative composition */
commit_speculation (spec_context);
```

Figure 1. Pseudo code for composing speculations using the programming constructs exposed by Anumita. In the absence of an evaluation function, the fastest surrogate (by time to solution) wins.

ploy binary instrumentation or collects memory traces. Instead, Anumita employs novel runtime techniques to provide transparent name-space isolation. Additionally, unlike [12, 14, 30] Anumita is capable of supporting nested speculations.

Trachsel and Gross [28, 29]'s competitive parallel execution (CPE) execute different variants of a single threaded program competitively in parallel on a multicore system and the variant that finishes first (temporal order) determines the execution time of that phase, thereby reducing the overall execution time. In contrast, Anumita is capable of supporting both sequential and parallel applications and provides expressive evaluation criterion (temporal and qualitative) to evaluate speculations. *To our knowledge, Anumita is the only system that provides such an innovative capability*. In contrast to Cledat et al. [10]'s opportunistic computing, where multiple instances of a single program are generated by varying input parameters to the program, which then compete with each other. Anumita is designed to support speculation at arbitrary granularity as opposed to the entire program.

5. Approach

In this section we briefly discuss Anumita's speculation model, a few of its programming constructs, and key elements of its runtime system.

Speculation Model, Syntax and Semantics: Figures 1 shows pseudocode for composing speculations using Anumita. A speculative composition is *initialized* by a call to `init_speculation`, which returns a speculation context. A composition is *instantiated* by a call to `begin_speculation`, which implements a concurrent continuation of the parent flow. Each speculative flow in the concurrent continuation is exactly identical to its parent flow in that it shares the same view of memory (i.e., global variables, heap and more importantly, the stack.), but is isolated from other concurrent speculative flows. In order to distinguish speculative flows from each other, we associate each speculative flow with a unique rank (similar in principle to MPI and OpenMP). A speculation may query its rank to map a particular unit of work to itself. The parent flow then enters an evaluation context, where it waits (descheduled) for evaluation requests from its speculative flows.

To implement an interface for evaluation, the call to `begin_speculation` takes an argument that specifies the size of a memory region that is used for communication between speculative flows and the parent evaluation context. Each speculative flow receives a distinct memory region of the specified size; this region is shared between a speculative flow and the parent evaluation context.

Periodically, a speculative flow can request an evaluation using the `evaluate_speculation` call, passing the parent intermediate results using the shared memory region. This call synchronously transfers control to the evaluation context (i.e., the idled parent flow), which executes the evaluation function and returns a status indicating whether the speculation calling the evaluation should continue or abort execution. The evaluation context may also use the intermediate results to cancel other speculations based on the results of the current evaluation, for instance, when the progress of one surrogate is significantly better than another within the same composition. In essence, the evaluation mechanism enables pruning of surrogates based on a user-defined notion of result quality.

On completing execution, a surrogate terminates the speculative region by calling `commit_speculation`. The first call to `commit_speculation` succeeds, can-

celing all other speculations in the composition and propagating its execution context to the parent flow, which then resumes execution at the point of commit. Selecting by time to solution (fastest surrogate wins) is trivially implemented by not specifying an evaluation function, as shown in Figure 1. In this case the first surrogate to commit would succeed and cancel its siblings. Anumita also provides an `abort_speculation` call that can be used by a surrogate to terminate itself if it detects that it is not making progress or has reached some failure mode. We also provide a `cancel_speculation` call that can be used by any surrogate to terminate any other surrogate.

Anumita's OpenMP *speculate* pragma is scoped between an open and close brace ({ and }), with an implicit `commit_speculation` at the end of the speculate pragma. In traditional OpenMP programming, name space isolation is achieved through explicit variable scoping (e.g., private, shared, etc.). To simplify programming, the Anumita runtime automatically isolates speculative flows without requiring explicit private scoping.

Runtime System: Neither POSIX thread model (shared address space) nor the process (distinct address space) model satisfies the isolation and selective state sharing requirements imposed by Anumita. Intuitively, we need an execution model that *provides the ability to selectively share state between execution contexts.*

To create the notion of a shared address space among processes, Anumita's runtime (a shared library) traverses through the link map of the application (ELF binary) at runtime and identifies the global data (*.bss* and *.data*) sections, and then unmaps these sections from the loaded binary image in memory, maps them from a SYSV memory mapped shared memory file and reinitializes these sections to the original values.

This mapping to a shared memory file is done by the *main* process before its execution begins at *main*. Speculative flows are then instantiated as processes (we use the *clone()* system call in Linux to ensure that file mappings are shared as well) and a copy of the address space of the parent is created for each instantiation of a speculation. Consequently, the speculations inherit the shared global data mapping. Hence any modifications made by a process to global data are immediately visible to all processes. This novel technique guarantees that all the processes have the same view of global data, similar to a threads model. In essence, this technique creates a set of processes that are semantically identical to threads, but operate in distinct virtual address spaces. By controlling the binding to the shared memory mapping, data can be selectively isolated or shared based on the requirements of the speculation model.

To implement a shared heap, we modified Doug Lea's dlmalloc [13] allocator to operate over shared memory mappings so that the allocated memory is visible to all processes. Anumita's runtime system provides global heap allocation by sharing memory management metadata among processes using the same shared memory backing mechanism used for .data and .bss sections. Additionally, our runtime system ensures that the base address of the stack in a speculative flow is identical to that of the parent speculation. When composing a speculation, the runtime saves the stack frame of the parent speculation and each speculation within a composition uses a copy of this stack frame for execution. *Each speculative flow is now identical to its parent flow, thereby creating a concurrent continuation.*

To create speculative flows, Anumita employs thread pool and to instantiate a speculation, Anumita's runtime propagates the execution context (*setjmp)*) and stack frame of the parent flow before waking up the speculative flows from the pool. The speculative flows adjust their stack and execution context (*longjmp*) before starting execution. To determine the write-set and contain (privatize) the updates of each speculation, Anumita's runtime employs page level protection and privatization of the shared VMA.

To perform the inclusion of updates of the winning speculation, Anumita implements a novel shadowing technique. Anumita's runtime system using the same shared memory objects and memory mapped files (globals, heap) creates an identical secondary (shadow) mapping with the 45th bit set in the virtual address space. This mapping is always shared among speculative flows and hence any updates to it are immediately propagated to all the speculative flows. To perform inclusion the runtime computes and XOR difference between its privatized page and its counterpart in the shadow address space and applies it to the shadow address space.

6. Experimental Results

We evaluated Anumati using three real applications: a multi-algorithmic PDE solving framework [22], a graph (vertex) coloring problem [16] and a suite of sort-

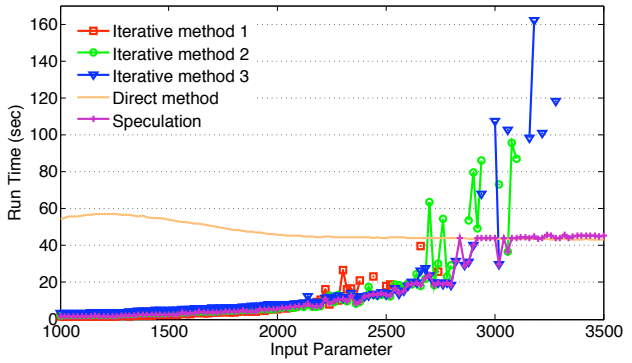

Figure 2. Time to solution for individual PDE solvers and speculation based version using Anumita. Cases that fail to converge in 1000 iterations are not shown. The results show that Anumita has relatively small overhead, allowing the speculation based program to consistently achieve performance comparable to the fastest individual method for each problem.

ing algorithms [26]. We performed all the experiments using a 16 core shared memory machine (NUMA) running Linux 2.6.31-14 with 64GB of RAM. The system contains four 2 GHz Quad-core AMD Opteron processors.

We ran each benchmark under two scenarios. The first scenario uses Anumita to speculatively execute multiple algorithms concurrently. This was done by modifying approximately 8-10 lines of source code in the above benchmarks. Since Anumita guarantees isolation, these modifications were short and required little to no understanding of the algorithms themselves. In the other scenario we ran the vanilla benchmark executing each algorithm individually.

For the graph coloring problem we experimented with over 80 DIMACS [11] data sets. Each data set (graph) has a fixed number of colors that it can use to color a graph. We used a set of 8 sorting algorithms with each algorithm sorting 8GB of input data. Using Anumita we ran all 8 algorithms speculatively creating a footprint of 64GB to stress test Anumita.

Our experimental results [1] indicate that Anumita is capable of significantly improving the performance of hard-to-parallelize and input sensitive applications by leveraging speculative parallelism. For the PDE solver (Figure 2) the speedup ranged from 0.84-36.19 and for the graph coloring the speedup was between 0.95-7.33 and for sorting suite we observed a speedup of 0.84-62.95.

Additionally, Anumita provides resilience to failure of optimistic algorithmic surrogates. In both graph coloring as well as PDE solvers, not all algorithmic surrogates successfully run to completion. In the absence of a system such as Anumita, the alternative is to run the best known algorithmic surrogate and if it fails, retry with a fail-safe algorithm that is known to succeed. While this works for PDE solver with Band Gaussian Elimination being the fail-safe, there is no clear equivalent for graph coloring, with each surrogate failing at different combinations of graph geometry and initial coloring.

Using Anumita it is possible to obtain the best solution among multiple heuristics. We found that in some cases where heuristics failed to arrive at a solution, the use of speculation guaranteed not only a solution but also the one that is nearly as fast as the fastest alternative.

7. Contributions

This research addresses an increasingly important problem of transitioning to many core processors; Achieving efficient parallel execution of so called un-parallelizable codes to that end this research exploits coarse-grain speculative parallelism. We have concretely realized our idea as Anumita, a language independent runtime system that is transparent and simple; It relieve the programmers from the complexities of concurrent programming in programming languages including C, C++, and Fortran and the OpenMP programming model. Experimental evaluation using real applications shows that Anumita achieves significant speedup without sacrificing performance, portability, and usability.

[1] The full paper on Anumita appeared in OOPSLA 2011.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[2] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, 53:90–101, August 2010. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/1787234.1787255`.

[3] 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 *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

[4] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic bombardment for the iterative solution of linear systems: A poly-iterative approach. *Jnl. of Computational & Appl. Math.*, 74: 91–110, 1996.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pages 81–96. ACM, 2009. ISBN 978-1-60558-766-0.

[6] C. Blundell, E. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, 2006. ISSN 1556-6056.

[7] H.-J. Boehm. Threads Cannot be Implemented As a Library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. URL `http://doi.acm.org/10.1145/1065010.1065042`.

[8] Boehm, Hans-J. and Adve, Sarita V. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI 2008, pages 68–78, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. URL `http://doi.acm.org/10.1145/1375581.1375591`.

[9] T. Chen, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI '10: Proceedings of ACM SIGPLAN 2010 conference on Programming Language Design and Implementation*, volume 45, pages 62–73, New York, NY, USA, 2010. ACM.

[10] R. Cledat, T. Kumar, J. Sreeram, and S. Pande. Opportunistic Computing: A New Paradigm for Scalable Realism on Many-Cores. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 5–5, Berkeley, CA, USA, 2009. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1855591.1855596`.

[11] DIMACS. Discrete Mathematics and Theoretical Computer Science, A National Science Foundation Science and Technology Center. `http://dimacs.rutgers.edu/`, April 2011.

[12] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*, volume 42, pages 223–234, New York, NY, USA, 2007. ACM.

[13] Doug Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, April 2011.

[14] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A Software System for Speculative Program Optimization. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 157–168, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0.

[15] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much Parallelism is There in Irregular Applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6.

[16] Marco Pagliari. Graphcol: Graph Coloring Heuristic Tool. `http://www.cs.sunysb.edu/~algorith/implement/graphcol/implement.shtml`, April 2011.

[18] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe Programmable Speculative Parallelism. In *PLDI '10: Proceedings of ACM SIGPLAN 2010 conference on Programming Language Design and Implementation*, volume 45, pages 50–61, New York, NY, USA, 2010. ACM.

[19] H. K. Pyla and S. Varadarajan. Avoiding Deadlock Avoidance. In *PACT 2010: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[20] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS XV: Proceedings of the 15th International conference on Architectural Support for Programming Languages and Operating Systems*, volume 38, pages 65–76, New York, NY, USA, 2010. ACM.

[21] L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel Distributed Systems*, 10(2):160–180, 1999. ISSN 1045-9219.

[22] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer–Verlag, 1985.

[23] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.

[24] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[25] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005. ISSN 0734-2071.

[26] Thomas Wang. Sorting Algorithm Examples. `http://www.concentric.net/~ttwang/sort/sort.htm`, April 2011.

[27] C. Tian, M. Feng, N. Vijay, and G. Rajiv. Copy or Discard execution model for speculative parallelization on multicores. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2836-6.

[28] O. Trachsel and T. R. Gross. Variant-based competitive Parallel Execution of Sequential Programs. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 197–206, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0044-5.

[29] O. Trachsel and T. R. Gross. Supporting Application-Specific Speculation with Competitive Parallel Execution. In *3rd ISCA Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, PESPMA'10, 2010.

[30] C. von Praun, L. Ceze, and C. Caşcaval. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP 2007, pages 79–89, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. URL `http://doi.acm.org/10.1145/1229428.1229443`.

[31] H. Pyla, C. Ribbens, and S. Varadarajan. Exploiting Coarse-Grain Speculative Parallelism. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2011, New York, NY, USA, 2011. ACM.