Semantically Ordered, Parallel Execution of Multiprocessor Programs

Gagan Gupta and Gurindar S. Sohi
Department of Computer Sciences, University of Wisconsin-Madison
{gagang, sohi}@cs.wisc.edu.

Emerging Challenges

Emerging approaches to improve efficiency and performance of computers are ushering an era of multiprocessor, "best-effort" systems. These systems are complex to use. Although rich in resources, they may not assure a program's automatic performance scaling or its completion. The onus is on the programmer to scale performance by developing parallel programs and exploiting parallelism. Further, hardware failures, resource (compute, energy, and power) management, and emerging approximate computing techniques can cause *exceptions*, disrupting programs. Consequently, the challenge is to improve multiprocessor usability, including programmability.

Current Approaches

The decades-old nondeterministic parallel programming model, originally developed for supercomputers, is commonly used to program multiprocessors. Nondeterminism complicates system use. But practitioners believe that it maximizes parallelism.

Some proposals simplify nondeterministic program expression, without addressing nondeterminism itself. Others overcome nondeterminism through run-time deterministic execution, but may penalize performance and portability. Yet others avoid nondeterminism by exploiting user-exposed dataflow within the program, which may be hard to apply to irregular parallel algorithms.

Some proposals explore deterministic, ordered parallel programs, but expose limited parallelism. Functional programming has also supported deterministic parallelism, but may not match the performance of imperative approaches.

To recover from exceptions in parallel programs, designers often use checkpoint-and-recovery (CPR). Our analysis shows that due to the overheads CPR may not scale to handle the expected frequent exceptions.

In short, prevailing approaches either do not "cure" nondeterminism, or trade performance for determinism.

Proposal Overview

We propose *ordered*, parallel execution of programs to simplify multiprocessor use and programming. Ordered execution provides a total order to the program's computations, unlike the partial order provided by the deterministic execution. Total order can simplify system use, including programming. We study the utility of this approach and its impact on performance. We apply the approach to two types of multiprocessor programs: the conventional, *explicitly-parallel* multithreaded programs and the *statically-sequential* programs.

Ordered Multithreading

We divide each thread in a conventional, data race-free multithreaded program into sub-threads at communication points therein. A logical order is assigned to the sub-threads, which are then scheduled for execution in that order. This leads to deterministic execution. The proposal is implemented as a runtime system for Pthreads programs [2]. It incurs negligible overheads on average (Figure 1).

Exception Recovery.

The deterministic execution above is made ordered can be applied to different uses by also making it *globally precise-restartable*, analogous to precise-interruptible sequential programs. Ordered execution can be applied to different uses, e.g., to simplify recovery from exceptions.

Briefly, the runtime tracks (i) the order of the program's currently executing sub-threads, (ii) the objects they may modify and (iii) the state of those objects before they are modified. When a sub-thread excepts, objects modified by it and by those "younger" to it can be restored to their pre-modified state, causing the program state to reflect precise, ordered execution up to the exception. The program may restart using this state, and is hence *globally precise-restartable*.

The logical order can be further exploited to selectively re-execute only the excepted sub-thread, without impacting the rest of the program, since only independent computations execute concurrently. This selective restart makes the approach scalable (Figure 2), making it well-suited for the highly exception-prone future systems. Importantly, it can potentially enable new capabilities, as described next.

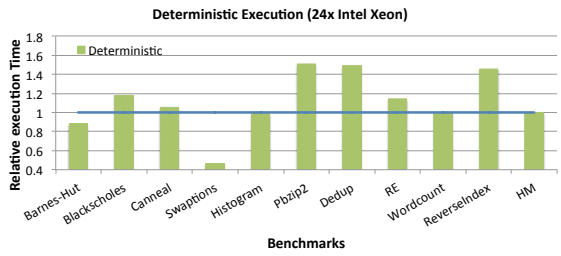

Figure 1. Deterministic execution time relative to nondeterministic Pthreads programs (horizontal line) on a 24-context machine. HM bar gives the harmonic mean.

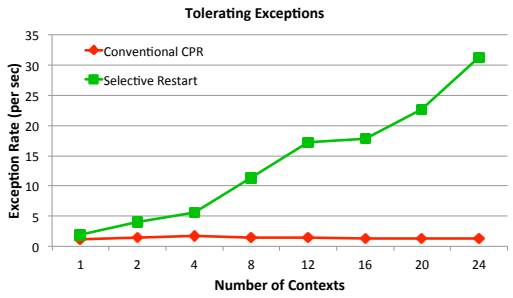

Figure 2. Exception resiliency of conventional CPR and selective restart for Pbzip2 running on a 24-context machine [2]. Selective restart can handle higher rates of exceptions.

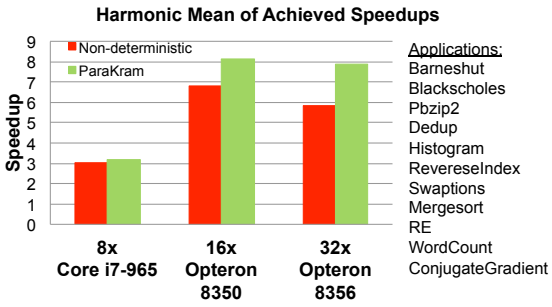

Figure 3. Harmonic mean of speedups achieved for standard parallel programs on an 8-core, a 16-core and a 32-core machine using conventional (Pthreads) and Parakram. Parakram achieves similar or better speedups.

Statically-ordered Programming

Ordered multithreading simplifies some aspects of system use, but not programming since the order introduced is not intuitive. To address this issue, we draw inspiration from out-of-order superscalar processors, which execute instructions concurrently and yet provide an ordered execution semantics. They exploit parallelism using the dataflow principles. Despite the concurrency, superscalar processors *simulate* ordered execution using precise interrupts. Moreover, they use precise interrupts to speculatively explore even higher degrees of parallelism. This overall approach, immensely successful in uniprocessors, simplifies programming, exploits parallelism and handles exceptions efficiently. We envision an analogous approach in multiprocessors.

Programming and Execution Model.

The proposed model, *Parakram*, relies on *statically-ordered*, object-oriented C++ programs in which functions manipulate "hidden" data and use well-defined interfaces to communicate with each other, avoiding side-effects that may impact other functions [1]. Parakram leverages programmers' expertise to develop parallel algorithms but eases the burden of explicitly orchestrating, and hence, reasoning about their parallel execution. This simplifies programming.

Analogous to superscalar processors, Parakram, a software runtime system, sequences through a suitably-written program, seeking to execute user-designated tasks concurrently. It dynamically establishes data dependences between the tasks. Independent tasks are executed concurrently; dependent tasks are serialized. Parallelism beyond stalled dependent tasks is exploited. The resulting execution is naturally deterministic, and can look deep into the program for parallelism, which conventional user-orchestrated execution cannot without enormous efforts. On popular parallel programs Parakram outperforms the conventional method (Figure 3).

Like in ordered multithreading, ordered execution is obtained by using global precise-restartability. Parakram is also similarly highly resilient to exceptions.

Ordered, Speculative Execution.

The above dataflow approach works well for programs (e.g., PARSEC) in which data dependences are easily identified . Computations in some programs (e.g., STAMP and Lonestar) can spend time identifying the data to be computed. This can hamper dataflow execution due to unknown dependences. To overcome this limitation Parakram executes computations speculatively, while maintaining ordered semantics (which current speculative methods do not). Parakram speculatively executes a task when its dependences are unknown. But it checkpoints the state of the objects the task may modify. Once the dependences are established and if a misspeculation is detected, Parakram treats it as an exception and recovers using global precise-restartability. This is similar to misspeculation handling in out-of-order superscalar processors. Using speculation Parakram outperforms the alternative methods, by up to $1.7\times$ in some cases (results not shown).

Conclusion

This work draws from principles proven successful in uniprocessors, to perform parallel, yet semantically ordered execution of multiprocessor programs. Experiments show that the approach simplifies programming without sacrificing performance. It can also potentially simplify other aspects of parallel system use, such as resource management, fault tolerance, security, etc.

References

[1] G. Gupta and G. S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 59–70, New York, NY, USA, 2011. ACM.

[2] G. Gupta, S. Sridharan, and G. S. Sohi. Globally precise-restartable execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 181–192, New York, NY, USA, 2014. ACM.