SEMANTICALLY ORDERED PARALLEL EXECUTION OF MULTIPROCESSOR PROGRAMS

By

Gagan Gupta

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2015

Date of final oral examination: 6/4/2015

The dissertation is approved by the following members of the Final Oral Committee:

Gurindar S. Sohi, Professor, Computer Sciences
Mark D. Hill, Professor, Computer Sciences
Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Karthikeyan Sankaralingam, Associate Professor, Computer Sciences
Michael M. Swift, Associate Professor, Computer Sciences
David A. Wood, Professor, Computer Sciences

Dedicated to Sanika, Megan, Ragini, Mom, and Dad

Acknowledgments

All that follows in the following pages is a synthesis of what I have learnt from my teachers at UW-Madison. This work could not have been possible without help from my colleagues and support from my family.

From Gurindar Sohi I learnt the art of analyzing problems, conducting research, drawing insights, and presenting the work. He was more than generous with his time, advice, coaching, and financial support. He permitted me to recruit as many as nine students to use the runtime systems I built as I explored multiple directions. I could not have asked for a better advisor.

Much of this work started in classes taught by David Wood, Mark Hill, and Jeff Naughton. David Wood and Mark Hill have supported the work enthusiastically throughout its duration. They periodically advised on presentation methods and research questions to answer. David Wood progressively raised the bar for me to achieve, whether on purpose I do not know. Mark Hill encouraged the work during the tough periods by reminding me of its utility. He also patiently guided me in making my claims precise.

I received guidance from Michael Swift on system issues related to this research. He also encouraged and helped me to widen the scope of this work to include concurrent programs. From Karthikeyan Sankaralingam I learnt to analyze and present related work. Mikko Lipasti made suggestions on future hardware-assist for the ideas here. He has also kindled my interest in neuro-inspired computing.

Ben Liblit helped me learn the basics of operational semantics. He patiently oversaw my application of it to describe the execution model the runtimes employ, although the description did not make it to the dissertation. The late Susan Horwitz guided work on static analysis of programs,

a direction that Rathijit Sen and I were unable to pursue to fruition for the lack of time.

Srinath Sridharan was my partner in several research crimes, in the course of which he gave me insightful feedback and perspectives on ideas, designs, and writings. My predecessor, Matthew Allen, left behind an excellent infrastructure that formed the launching pad for the runtimes I eventually built. With Rathijit Sen I had many enriching philosophical discussions on specific problems, defining terms, the claims I made, and research in general. I received much programming help from Marc de Kruijf, Venkat Govindaraju, and Jai Menon. I have also learnt one thing or another from Arka Basu, Nilay Vaish, Hongil Yoon, Cong Wang, Emily Blem, Derek Hower, Dan Gibson, Tony Nowatzki, Lena Olson, and Jason Power. Eric Harris, Peter Ney, Luke Pillar, Raghav Mohan, Dinesh Thangavel, Bhuvana Kukanoori, Raj Setaluri, and Aditya Venkataraman tested the runtime systems, and helped develop several programs used in this work.

Finally, my daughters, Sanika and Megan, have endured a mentally absent father for the last several years. My wife, Ragini, has sacrificed more than me as I pursued this endeavor. If this research amounts to anything, much credit should go to her. Ragini also editorializes all that I write.

Contents

List of Figures, Tables, and Listings

Abstract

Conventional wisdom says that to scale a program's performance on multiprocessors, designers must explicitly exploit parallelism by discarding the sequential order between computations. Unfortunately, the resulting nondeterministic execution makes systems difficult to program and operate reliably, securely, and efficiently. Unless made easier to use, multiprocessor computing systems may remain underutilized.

Proposals to improve multiprocessor usability largely address the individual aspects of system use affected by the nondeterministic execution, without addressing the nondeterminism itself. Hence these proposals may have limited utility. Moreover, they can be complex and limit performance.

To improve system usability, this dissertation proposes parallel, but semantically ordered execution of ordered programs on multiprocessors. Ordered programs are sequentially expressed parallel algorithms composed for multiprocessors. Parallelism from these programs is reaped by performing dataflow execution of computations. To maximize the parallelism, the computations may also execute speculatively when the dataflow between them is temporarily unknown. Despite the concurrency and speculation, the execution is managed to make the computations appear to execute in the program order, eliminating nondeterminism. This approach improves usability by eliminating nondeterminism, and yet exploits the available parallelism.

But, can an ordered approach, ostensibly antithetical to parallelism, effectively exploit parallelism and simplify system use in practice? To answer this question, the approach is applied to two of the more complex aspects of system use: programmability and reliability mechanisms.

In this dissertation we design runtime systems to implement, apply, and test the ordered

approach. The runtimes provide a C++ programming interface to develop ordered programs, and libraries to parallelize their exception-tolerant execution. The runtimes were used to express a range of parallel algorithms as ordered programs. The ordered approach matched the explicitly parallel programs in expressing algorithms, with the added benefit of simplified programming. In almost all cases it also yielded comparable performance. However, on certain "highly nondeterministic" algorithms, it may underperform, presenting a choice to trade off performance for simplicity in such cases. The ordered approach also simplified recovery from exceptions, such as transient hardware faults. Its ability to handle exceptions scaled with the system size, where the conventional method failed altogether.

1 Introduction

I suppose that we are all asking ourselves whether the computer as we now know it is here to stay, or whether there will be radical innovations. In considering this question, it is well to be clear exactly what we have achieved. Acceptance of the idea that a processor does one thing at a time–at any rate as the programmer sees it–made programming conceptually very simple, and paved the way for the layer upon layer of sophistication that we have seen develop. Having watched people try to program early computers in which multiplications and other operations went on in parallel, I believe that the importance of this principle can hardly be exaggerated.

— Maurice Wilkes (1967)

By discarding order between computations, nondeterministic parallel programs may yield performance, but complicate system use. Programs with implicitly ordered computations can do better. Respecting the implicit order during their execution can simplify system use. Despite the implicit order, the execution can be orchestrated to be parallel without compromising the performance in all but a few cases.

1.1 Improving Multiprocessor Usability is Important

Computers have proliferated almost all walks of life today, making information economy the backbone of modern society. The proliferation has been fueled by the phenomenal advances in hardware and software. Designers developed progressively more powerful computers to solve increasingly more complex problems. Successful solutions, in turn, drove further proliferation, creating a demand for even more powerful computers to solve new problems. Sustaining this cycle of innovation is important for the continued growth of the information economy and society.

In our view, a key to this virtuous cycle was the relative ease of using computers. We believe that the intuitive, sequential view of a program's execution on computers is central to their usability.

Computers present a consistent sequential view to users, abstracting away the complex hardware-software mechanisms used to speed up the execution, or to enable new system capabilities (e.g., resource management). The sequential view put the increasing capabilities of computers within the reach of more and more programmers. It simplifies programming, simplifies the analysis and design of efficient systems, and promotes overall productivity.

A confluence of recent and emerging trends, however, threatens the above cycle of innovation. Multiprocessors have replaced uniprocessors in almost all devices, ranging from mobile phones to cloud servers. Although multiprocessors provide more computational power, unlike before, the computational power is no longer within the easy reach of programmers. Programmers now have to explicitly tap it, by developing parallel algorithms, expressing the algorithms as parallel programs, and managing the program's execution for the desired efficiency. Programmers also have to account for the target platform's artifacts, e.g., the memory consistency model. Unfortunately, for various well-known reasons, parallel programming has proven to be challenging to both humans and tools alike [180].

Looking ahead, future systems may further complicate matters by failing to execute programs as intended. Designers are striving to utilize compute, energy, and power resources more and more efficiently. To improve efficiency, they are proposing dynamic techniques to scale voltage [81, 82], operate hardware close to margins [188], compute approximately [17, 65, 66, 160], share resources[24], and provide only best-effort resources [3, 60], among others. Further, ultra-deep submicron semiconductor technology is rendering computer components increasingly unreliable[28, 99]. As a result, programs will be frequently exposed to dynamically changing, unreliable, potentially insecure, and imperfect resources. Consequently, programs may not execute without interruptions, or complete, or execute efficiently, or produce correct results. Moreover, growing system sizes will make them more vulnerable to such vagaries.

In summary, the evolving technology trends will have the following implications for future computers. Multiprocessors will be ubiquitous. Programmers will be required to develop parallel algorithms. But developing and deploying parallel programs will be complex. Multiprocessors

may not automatically scale their performance. Worse, due to techniques used in these systems, programs may frequently produce erroneous results or not run to completion. Yet, users will desire systems that are easy to program, on which programs complete, efficiently, with the desired accuracy, and with minimum intervention. To do so, making the program's execution easy to manage and analyze will become important. Unless their usability is improved, multiprocessors may remain underutilized, slowing the cycle of innovation.

1.2 The State of the Art is Unsatisfactory

Although the challenges posed by the above multiprocessor trends impact different aspects of system use, ranging from programming to reliability, as we shall see in Chapter 3, the execution characteristic of the program is key to them all. The execution characteristic of parallel programs, in turn, is influenced by the programming abstraction.

Since Gill first anticipated and coined the term *Parallel Programming* in 1958 [75][1], researchers have proposed more than three hundred parallel programming abstractions[2]. They have primarily focused on simplifying programming.

The prevailing and predominant approach is *Multithreading*. The popular APIs, Pthreads [1], OpenMP [48], MPI [70], and TBB [151] are some examples of this approach. Multithreading approaches rely on **multithreaded** programs. Multithreaded programs do not specify a static total order on their computations, i.e., any order of execution of parallel computations may be acceptable. The lack of order leads to nondeterminism.

Multithreading gives programmers the utmost freedom to express and exploit parallelism, setting the theoretical benchmark for parallel processing. But Multithreading puts the entire burden to exploit the parallelism on the programmer. Programmers have to explicitly coordinate the execution, while accounting for the underlying system architecture. Despite the general lack of

[1] Interestingly, in the addendum to the same paper, J.A.Gosden immediately predicted the problems parallel programming might create in operating computers–issues that we address in this dissertation!

[2] We do not include work on auto-parallelization of sequential programs in this discussion, due to its limited success in exploiting reasonable degrees of parallelism in algorithms for non-scientific problems [128].

order, on occasion programmers have to enforce order, e.g., between dependent computations. Nondeterminism and the need for explicit coordination complicate reasoning about the program's dynamic execution, managing it, and developing correct programs. Multithreaded programming can be onerous [109, 180]. Although some proposals simplify some aspects of multithreaded programming (to be seen in Chapter 9), overall, it makes multiprocessor systems difficult to use.

To overcome the challenges nondeterminism poses, researchers have tried to eliminate it in recent proposals. *Deterministic* approaches eliminate nondeterminism by introducing determinism during the execution. Like Multithreading, some Deterministic approaches start with multithreaded programs. Although the multithreaded program defines no static order, Deterministic approaches introduce a partial dynamic order between the computations by ordering accesses to a given datum. This order is repeatable, and hence partially eliminates the nondeterminism. Determinism can potentially make the execution easier to reason, possibly simplifying system use. However, Deterministic approaches are not entirely satisfactory solutions. Some Deterministic approaches can penalize the performance [18, 22, 135, 137], sometimes severely, and may require complex hardware support [51, 52, 97]. Moreover, the introduced order can be arbitrary and system-specific, and hence it is unclear how effective they are in practice.

Other deterministic approaches enforce a user-provided order, by using **ordered** programs [2, 9, 34, 141, 154, 179, 183, 190]. Contrary to multithreaded programs, ordered programs specify an order on computations, either implicitly [9, 141, 154, 183] or explicitly [2, 34, 179, 190]. These proposals also use the order to automate the parallelization, relieving the programmer from explicitly coordinating the execution. Since the order is derived from the program, it is portable and more intuitive. However, whereas some proposals perform well on some programs [9, 141], others have shown otherwise on other programs [88]. Further, our analysis shows that these approaches may not match multithreaded programs in expressing and exploiting parallelism. Moreover, the order, although not arbitrary, is deterministic only for accesses to a given program variable, but not between variables. As we show in Chapter 3, in practice deterministically ordering accesses to individual program variables alone is insufficient to simplify system use. For all of these reasons,

deterministic approaches have not found a broad acceptance.

In short, although multiprocessor programming has received much attention, we believe that no proposal is satisfactory. Issues other than programming, e.g., efficiency and reliability, that arise in operating parallel systems, have received only nominal attention. Solutions for them are largely based on the prevalent multithreaded programs. Nondeterminism of multithreaded programs complicates their design, but alternatives are limited as long as multithreaded programs are used.

1.3 Semantically Ordered Parallel Execution

We observe that the state-of-the-art multiprocessor programming approaches have moved from nondeterminism to determinism, making the program's execution less intractable to reason. Although a step in the right direction, we believe that ultimately they do not go far enough.

Drawing lessons from the past, especially from the factors that have contributed to computer proliferation, we argue for the elimination of all nondeterminism from the program's execution for a given input. We note that modern microprocessors execute instructions in parallel and yet provide a precise-interruptible, sequentially ordered view of the execution. The execution itself may be parallel, but it *appears* to be totally ordered.

Sequential execution has several desirable properties, which arise because computations execute, or at least appear to, one at a time in a specified order. For a given input (which may arrive asynchronously during the execution), the flow of execution and the results it produces are naturally repeatable, intuitive to reason, and amenable to intermediate interruptions. These properties make it easy to analyze and manage the program's execution. Almost all aspects of system use rely on managing the program's execution in one form or another. For example, debugging a program requires stopping and resuming the execution from desired points, as does fault tolerance. By discarding order, multithreaded programs programs lose repeatability, intuitiveness, and interruptibility. Worse yet, they may admit conditions such as data races and deadlocks, which further compound the complexity.

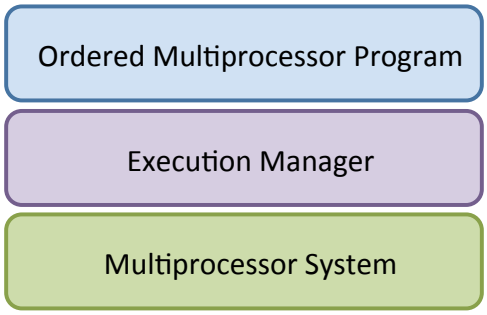


Figure 1.1: Logical view of the proposed approach, Parakram. Parakram relies on sequentially expressed parallel algorithms, i.e., **ordered** multiprocessor programs, and introduces an Execution Manager between the program and the system. The Execution Manager orchestrates the program's parallel execution, while providing an ordered view.

Therefore, analogous to sequential execution, we propose ordered execution of multiprocessor programs to obtain the concomitant properties. The challenge is to then achieve ordered execution, but without compromising the parallelism.

1.3.1 Parakram: Ordered Programs, Ordered Execution

In this dissertation we take a broader view of how programs might be composed, executed, and managed on multiprocessor systems.

We start from **ordered programs** and then obtain a parallel, yet ordered execution. We envision that programmers will develop parallel algorithms, but express them sequentially, as ordered programs. Ordered programs will express computations, but not explicitly coordinate their parallel execution. Although developed for parallel execution on multiprocessors, they will *imply* a total order on computations. The order will be implied through the program's text, analogous to the canonical sequential programs.

To orchestrate the program's envisioned execution, we interpose an **Execution Manager** between the program and the system, as depicted in Figure 1.1. The Execution Manager, oblivious to the programmer, will dynamically parallelize the execution, while respecting the implicit order. Importantly, the Execution Manager will abstract away the system from programmers. It may

be implemented in hardware, or software, or in some combination of the two, although we have pursued software designs in this dissertation. No hardware modifications are needed. We call this approach, **Parakram** , and also interchangeably refer to it as the "ordered approach".

The ordered approach at once addresses multiple issues in multiprocessor programming. Its execution is deterministic, amenable to interruptions, and is intuitive since the order is derived from the program itself. Further, it minimizes the programmer's burden of explicitly coordinating the parallel execution. It leaves the programmer to deal with only the parallel algorithm, and not its dynamic execution, simplifying programming.

Although appealing, the ordered approach will be viable only if it matches the established multithreaded programs in expressing and exploiting parallelism. Introducing order where one is presumably not needed, raises the following questions: Would order obscure the parallelism and hinder performance of parallel algorithms? Can all types of parallel algorithms be expressed as ordered programs? Can the artificial ordering constraint in such programs be overcome to discover the parallelism automatically?

To answer these questions we analyzed and distilled the key parallelism patterns that arise when solving problems using multithreaded programs. We ensured that Parakram permits all commonly used parallelism patterns to be expressed programmatically in ordered programs, and realized during the execution. Parakram comprises two logical components. The first is a task-based programming abstraction to express annotated, arbitrary parallel algorithms. The second is an execution model to effect their parallel execution.

Parakram's execution model, implemented in the Execution Manager, borrows the principles from the widely successful out-of-order (OOO) superscalar processors [172] to exploit parallelism on multiprocessors. Parakram's operations are logically depicted in Figure 1.2. It retains the ordered view of programs, but treats a processor (or an execution context) as a functional unit. Instead of instructions, coarser-grained tasks ❶ form units of computations. Inter-task dependences are identified on the basis of memory locations (that hold data) accessed by the tasks, instead of registers ❷. Dataflow schedule is followed to execute the tasks ❺.

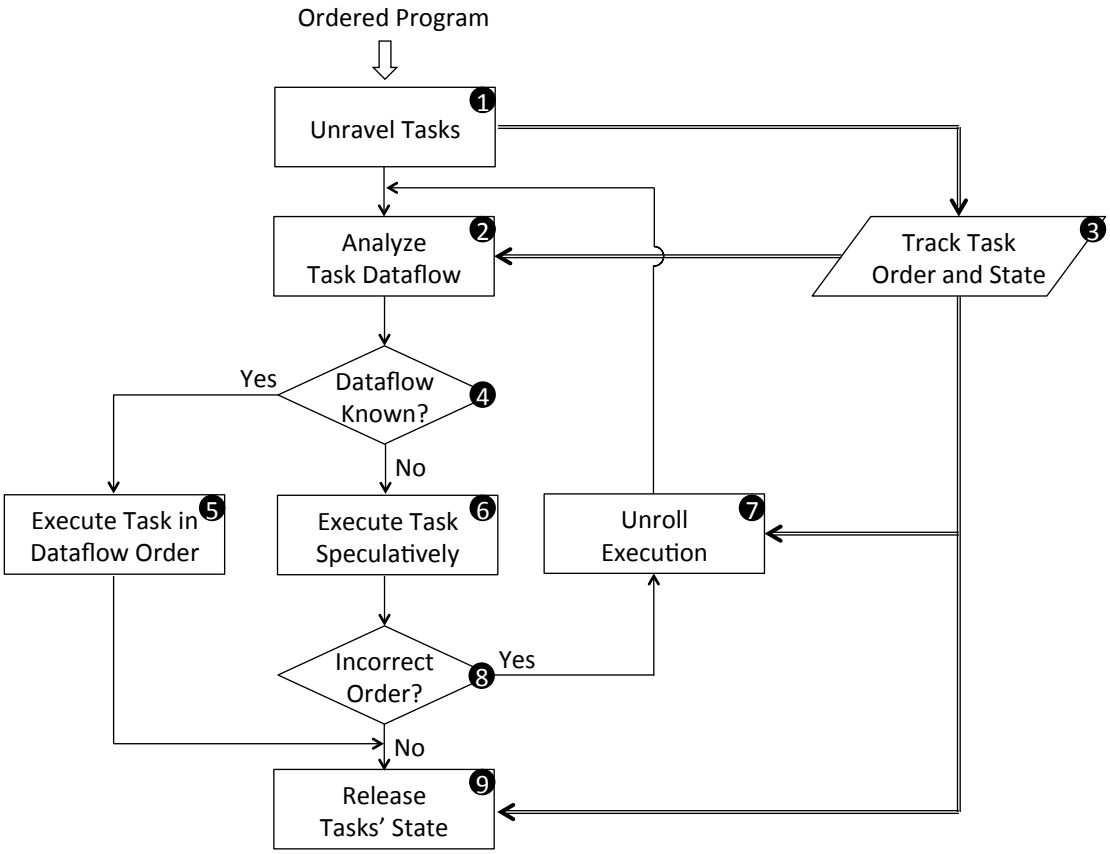


Figure 1.2: A logical flowchart of Parakram's operations.

Merely scheduling tasks in dataflow order results in deterministic, but not ordered, execution. To obtain an appearance of a total order, we use a mechanism analogous to precise-interrupts in microprocessors. We permit tasks to execute in parallel, but construct a precise architectural state, after the fact, when needed, to achieve **globally precise-interruptible** execution. In such an execution, tasks only up to the desired point in the program, and none beyond, appear to have completed. Tasks and the state that they modify are tracked using structures analogous to the Reorder (ROB) and History Buffers in microprocessors ❸ [171]. The architectural state reflective of the program's execution up to a given task, or even up to an instruction, can be constructed using this information. Globally precise-interruptible execution can be paused and resumed from any point in the program.

When Parakram cannot accurately analyze the inter-task dataflow ❹, for a variety of reasons that we shall see in Chapter 6, it resorts to **dependence speculation** to maximize the parallelism opportunities ❻. When precise dataflow between tasks in unknown, Parakram speculates that there is none, and executes the tasks without waiting to resolve the dataflow. However, speculation can lead to execution in the wrong order. If found to be so subsequently ❽, Parakram rolls back and rectifies the execution ❼, similar to misspeculation handling in OOO processors. To support speculation, Parakram conveniently relies on global precise-interruptibility.

Although concurrent, possibly even speculative, Parakram execution will be ordered[3]. Whether inspected at the end or intermediately, the execution will appear to have followed the defined program order.

Parakram's execution model has some similarities with Thread-level Speculation (TLS) [150, 174]. TLS proposals typically exploit parallelism from sequential programs, as they encounter loop iterations and functions during the execution. By contrast, Parakram proactively seeks and exploits higher degrees parallelism from a much larger window of tasks in programs composed for multiprocessors.

I/O in a parallel system can pose challenges since it is often required to be performed in a strict order amidst the otherwise concurrent execution. Multithreading approaches often require programmers to explicitly enforce the order. By contrast, ordered execution naturally simplifies I/O operations.

Any method applied to parallel systems must also tackle scalability. At relatively larger scales, different system sizes can warrant different designs and implementations. This dissertation's primary focus is to test the viability of what is essentially a new approach. Hence in this work we apply Parakram to shared memory multiprocessor systems of ubiquitous sizes of tens of processors or threads. Lessons learnt from this exercise can then be useful in applying the approach to larger systems, an aspect we touch upon later in Chapter 10.

To evaluate Parakram we developed a software runtime prototype in the form of a C++ library.

[3]Formally defined in Chapter 2.

The library provides a small set of APIs to develop imperative ordered programs in C++, although the Parakram principles are equally applicable to functional programming languages. The library also implements the Execution Manager. We describe the library's design and implementation in this dissertation.

The library applies the proposed approach in terms of the prevalent object-oriented programming. The library raises the level of abstraction in the program, viewing functions as tasks and objects as data. Programmers annotate object-oriented programs with hints identifying potentially parallel methods, i.e., functions, and the objects they compute. The Execution Manager leverages these hints to apply its execution mechanisms. Not all programs are suitably-written. The library provisions to handle such cases gracefully.

Applying dataflow, dependence speculation, and globally-precise interrupts at the scale of multiprocessors presented unique implementation challenges. In response, the Execution Manager implements a range of mechanisms. The Manager is a parallel runtime system, modeled after runtimes like Cilk [71] and PROMETHEUS [9], but enhanced to meet its aggressive objectives. Its design is distributed and asynchronous. A distributed design avoids potential centralized bottlenecks, and the asynchronous design enables the OOO dataflow execution. The Manager manages the program's tasks and their state to enable speculation and globally-precise interrupts. It employs high performance non-blocking data structures and mechanisms to enable the concurrency. The non-blocking data structures were drawn from a mix of proposals [68, 91, 167] and adapted to suit our design.

We used the Parakram library to develop ordered programs spanning the range of common parallelism patterns. We evaluated how well Parakram programs can express and exploit parallelism. Developing ordered programs was easier than the multithreaded programs. Experiments on stock multiprocessor systems showed that ordered programs can express the parallelism idioms that are commonly found in multithreaded programs. Experiments showed that further, the artificial constraint of order can be overcome to exploit the parallelism. Except in the cases of certain algorithms, Parakram matched the performance of multithreaded programs, written using

Pthreads [1], OpenMP [48], Cilk [71], or TL2 software TM [54].

In two cases Parakram programs may underperform in comparison to the multithreaded programs. The first is when the algorithm uses very small task sizes. Parakram needs tasks to be of a minimum size to achieve speedups, which can be slightly higher than the size needed in primitive APIs like Pthreads. Parakram's specific implementation influences the minimum needed size. The second case arises when the algorithm is "highly nondeterministic", i.e., in which tasks are predominantly independent of each other, and traverse the data structures that they compute, e.g., trees. Highly nondeterministic algorithms pose a more basic limitation to the ordered approach. They can constrain the parallelism that an ordered approach can exploit in comparison to Multithreading. In such cases, the ordered approach presents a choice to trade off performance for productivity. However, at large, order need not constrain parallelism. Interestingly, the ordered approach can exploit parallelism in some cases more naturally than the more explicit Multithreading approach.

1.3.2 Applying Ordered Execution to Simplify Other Uses

Several issues besides programming, such as security breaches, hardware faults, exception conditions, and inefficiencies, arise when operating computing systems. Solutions to these issues become complex in multiprocessor systems due to the parallel program's execution characteristics. Ordered execution can help simplify the design of these solutions. For example, elsewhere we have applied ordered execution to improve system efficiency [177, 178].

In this dissertation we explore how ordered execution simplifies yet another use-case, recovering from frequent exceptions. We treat events that alter the prescribed flow of a program's execution as an exception. For example, such events could arise due to hardware faults or OS scheduling interrupts to manage resources. Exceptions may also be program-generated, e.g., due to divide-by-zero errors, or be programmer-induced, e.g., due to debugging breakpoints. In future systems, as designers introduce techniques to compute approximately, scale voltage aggressively, schedule tasks in heterogeneous system, etc., we believe that exceptions will become the norm.

The conventional checkpoint-and-recovery (CPR) method is commonly used to recover from

exceptions in the prevalent multithreaded programs [62]. CPR periodically checkpoints the program's state. To recover from an exception, it restarts the program from a prior error-free checkpoint. A plethora of hardware [5, 134, 145, 176] and software [32, 33, 56, 114, 120, 153, 189] approaches, striking trade-offs between complexity and overheads, have been proposed in the literature. Our qualitative analysis shows that their overheads will be too high to handle frequent exceptions. In fact, they lack the scalability needed for future, increasingly larger, exception-prone systems.

We apply the proposed globally precise-interruptible execution to simplify recovery from exceptions. We introduce a notion of **selective restart**, in which not all computations, but only those affected are restarted. Selective restart exploits the dataflow tasks in Parakram programs to localize the impact of exceptions to a minimum number of program's computations. Minimizing the impact of exceptions on the program results in a scalable design. Ordered execution also helps to combine checkpointing and log-based approaches to minimize the recovery overheads.

We incorporated the exception recovery capability in the Parakram runtime prototype. It can uniquely handle exceptions in the user code, third-party libraries, system calls, as well as its own operations.

We evaluated the Parakram prototype by applying it to recover from non-fatal exceptions in standard parallel benchmarks. Experiments showed that selective restart outperformed CPR. Importantly, Parakram withstood frequent exceptions, and scaled with the system size, whereas the conventional method did not, validating our qualitative analysis.

1.4 Dissertation Outline

In Chapter 2 we define the common terms used in the rest of the dissertation. Chapter 3 presents the properties of a multiprocessor program's execution and their impact on the system's usability. It describes properties of ordered execution and how they can simplify system design. Chapters 4, 5, 6, and 7 present the Parakram approach. In Chapter 4 we present our approach to multiprocessor programming, and compare it with the prevailing Multithreading approach. This chapter expands

on our earlier work [78]. The chapter summarizes the overall Parakram approach to programs and their execution. Chapter 5 analyzes the common patterns that arise in parallel algorithms. It presents the Parakram APIs, which admit similar patterns in ordered programs. A few examples of Parakram programs are also presented and examined. Parts of this chapter have been submitted to PACT'15 [77]. In Chapter 6 we describe Parakram's execution model and the various techniques it employs, dataflow execution, speculation, and globally-precise interrupts, to exploit the parallelism. The details of the dataflow execution [76] and the principles of globally-precise interrupts [80] have been presented at other venues, whereas the details on speculation are currently in the same submission to PACT'15 [77]. Chapter 7 evaluates the expressiveness and performance implications of the ordered approach. We explore its benefits and limitations. In Chapter 8 we apply ordered execution to simplify exception recovery. We analyze the performance implications of designs based on nondeterministic and ordered execution. Related work on exception handling is also summarized. Some aspects of exception recovery, although applied in the context of multithreaded programs, but germane to the discussion in this dissertation, were presented at PLDI'14 [80]. Next, we compare and contrast our work with the large body of work on multiprocessor programs, in Chapter 9. Chapter 10 concludes with our final thoughts and future directions.

2 Terms and Definitions

Some common terms related to multiprocessor algorithms, programs, and execution have been used inconsistently in the literature, and on occasion with inconsistent definitions. In this chapter we state and define the main terms used in this dissertation. We have tried to be consistent with as much existing literature as possible in our use.

Solving a problem on a multiprocessor system requires developing a parallel algorithm, expressing the algorithm as a suitable multiprocessor program, and effecting the program's parallel execution. The characteristics of algorithms, programs, and the execution are the focus of this dissertation. Although the concepts implied here are applicable to programs written in any programming language in general, assume imperative languages for the discussion.

2.1 Algorithms

To solve a problem at hand, the programmer first develops an algorithm. Constructing algorithms is not our focus, but their characteristics do impact our work. To take advantage of multiprocessors the algorithm must be parallel, i.e., computations in the algorithm must be amenable to parallel execution. Even if an algorithm is parallel, typically some computations are performed in a specific order, e.g., I/O, or the initialization before the algorithm's main parallel kernel, or the final steps after the kernel.

An algorithm is **nondeterministic** if it solves a problem by performing computations in its main kernel in no specific order. For example, the algorithm that finds the shortest path between a given source node and all other nodes in a graph by traversing the graph no specific order is nondeterministic [88]. The same problem can be solved by the well-known Dijkstra's algorithm by

performing at least some traversals in a specific order (to be more efficient) [88][1].

2.2 Programs

Definition 2.1 (**Instruction**). *An **instruction** is a primitive operation performed by a machine.*

Definition 2.2 (**Statement**). *A **statement** is the smallest standalone element that expresses some action, as is commonly understood in a programming language.*

Assume that a statement translates into an instruction, or an ordered sequence of instructions.

Definition 2.3 (**Ordered Program** and **Program Order**). *A program is **ordered** if it statically specifies a total order on its statements. The total order is also called the **program order**.*

Ordered programs may statically imply the order through the program's text, as sequential programs do. We shall discuss ordered multiprocessor programs in more details in the following chapters.

In practice, programs that are not ordered are seldom totally unordered, but are often partially ordered, i.e., at least some statements, e.g., within a task, or a function, or parts of the algorithm as stated above, are performed in order. The conventional multithreaded programs and programs in Deterministic proposals that may explicitly specify partial order between statements (e.g., Intel CnC programs [34]), are examples of such "unordered" programs.

2.3 Execution

Definition 2.4 (**Strictly Sequential Execution**). *Given an ordered program, its **strictly sequential execution** is the sequence in which the dynamic instances of its instruction are performed as per the static order.*

[1] Others have termed what we call nondeterministic algorithms as **unordered** [88, 142].

Strictly sequential execution results in a totally ordered sequence of instructions.

Even if a program does not specify a total static order, some (legal) order can be enforced on its execution to obtain a totally ordered sequence of instructions, as can be done for multithreaded programs [80].

Definition 2.5 (**Computation**). *A **computation** is an instruction or any contiguous sequence of instructions performed by a program on a single execution context, where a dynamic instance of an instruction can belong to one and only one computation.*

No two computations can overlap. Once a computation is defined, its definition cannot change in a particular discussion. Although the definition admits computations comprising arbitrarily long sequences of instructions, computations corresponding to logical structures in the program, e.g., statements, basic blocks, logical group of statements that operate on a program variable, functions, or tasks, are more purposeful.

A totally ordered sequence of instructions may also be viewed as a totally ordered sequence of computations for some definition of computations.

Definition 2.6 (**Point**). *A **point** refers to a computation in a totally ordered sequence of computations.*

Definition 2.7 (**Precede**). *A computation performed before a particular computation in a totally ordered sequence of computations is said to **precede** the particular computation.*

Definition 2.8 (**Succeed**). *A computation performed after a particular computation in a totally ordered sequence of computations is said to **succeed** the particular computation.*

A program's execution may be **nondeterministic**, **deterministic**, or **ordered**.

Nondeterministic execution is as is generally understood, one in which a program variable may not be assigned the same sequence of values in different executions with the same input. Nondeterministic execution is pertinent to multithreaded programs, and is perhaps not meaningful in the context of ordered programs.

Definition 2.9 (**Deterministic Execution**). *A program's execution is* ***deterministic*** *if each program variable is assigned the same sequence of values in any execution with a particular input*[2].

The input to the program may arrive during its execution. Deterministic execution orders the computations that access a variable, thus partially ordering the program's computations. Kendo [137], CoreDet [18], Grace [22], Calvin [97], and others perform deterministic execution of multithreaded programs. They impose an implementation-specific sequence on the execution. Jade [154], SMPSs [141], and PROMETHEUS [9] perform deterministic execution of ordered programs. They derive the sequence from the program's text.

Definition 2.10 (**Ordered Execution**). *Given a program of totally ordered computations, the program's execution is* ***ordered*** *if at any point in the execution, the architectural state reflects that all computations that precede the point have completed and none others have started.*

Ordered execution is the main objective of our work. Note that in the ordered execution, computations need not be performed in the given order, but the architectural state must provide an appearance as if they did. Ordered execution appears to produce the same totally-ordered sequence of assignments to the program variables in every run for an input. Ordered execution is also deterministic.

Note that the execution type is distinct from the algorithm type despite the use of the common term *nondeterministic* to describe them. A nondeterministic algorithm may be expressed as a multithreaded program, whose execution may be nondeterministic. A nondeterministic algorithm may also be expressed as an ordered program, but with the added artificial, ordering constraint. Its execution may be ordered. When the term is used, its meaning will be evident from the context, otherwise it will be explicitly qualified.

[2]Karp first defined this property and called it **determinate** [101], but the term *deterministic* has gained currency in the literature.

3 A Case for Ordered Execution of Ordered Programs

Everything that's worth understanding about a complex system, can be understood in terms of how it processes information.

— Seth Lloyd

The properties of a program's execution on a system influence how easy or difficult the system is to use. In this chapter we identify key desirable properties and study their impact on the system's programmability and usability. Nondeterministic execution lacks these properties whereas ordered execution naturally exhibits them.

Programmability. Complexities in composing the multithreaded program, arising chiefly due to the difficulties in analyzing the program's nondeterministic execution, are well understood and documented [118, 181]. We will briefly touch upon the related issues and present how ordered programs and their ordered execution can address the issues.

Usability. When programs are being developed and are eventually deployed, various artifacts can prevent the programs from functioning as intended. These artifacts can arise from within the program, e.g., software bugs, or from the system, e.g., hardware faults. Unless addressed adequately, they can adversely affect the system's usability. For example, bugs in parallel programs are known to have been fatal in the extreme [112]. Frequent faults can prevent programs from completing execution [35, 36].

To tackle these artifacts and ensure that programs function as desired, designers often use **design-assist** solutions. Design-assist solutions are tools and procedures used to identify the

source of the artifacts, or recover from them, or circumvent them. Design-assist solutions often become an integral part of the program development process and the system design. Debugging procedures and fault tolerance mechanisms are examples of design-assist solutions. As we describe in Section 3.1, some solutions may rely on managing the execution, e.g., to recover from faults, while others may require reasoning the execution, e.g., to find bugs. The program's execution properties influence the processes of managing and reasoning, and hence also the implementations of the design-assist solutions, as we shall see in this chapter.

We examine the properties of the conventional sequential execution, and examine how they simplify the management and analysis, in Section 3.2. By contrast, nondeterministic execution loses these properties, which makes managing and reasoning the execution intractable. Hence multithreaded programs pose considerable challenges to implement the design-assist solutions (Section 3.3). Deterministic execution can help matters, but is still inadequate (Section 3.4). Based on this study of execution properties, we make a case for ordered execution of ordered programs. Subsequently in the dissertation, we explore the impact of nondeterministic and ordered execution on two of the more complex aspects of system use, program development and exception handling.

3.1 Design-assist Solutions

Design-assist solutions, which help ensure that programs execute as intended, although varied in design and execution scenarios, rely on three basic common operations. They rely on the ability to *pause* the execution at a precisely desired point, possibly *analyze* the execution, and *resume* it thereafter. These operations may be periodically repeated at intermediate points during a single run of the program, or across multiple runs. We briefly describe how these operations are used in practice, and then turn our attention to how the properties of the program's execution affect them.

Debugging and testing. Almost all non-trivial software development undergoes debugging and testing.

Programs are typically debugged by examining and analyzing the state modified individually or collectively by computations [10]. The control-flow followed during the execution is also often analyzed. The observed execution is compared with "expected" results, which are often precomputed, e.g., by analyzing the program's text. Reasoning about the execution also involves similar steps.

In the debugging process it is customary to examine the state at intermediate points during the execution, as well as after the program completes. Pausing the execution, often by setting breakpoints, inspecting the state, and resuming the execution are critical to this process. Multiple iterations of this process, on repeated executions, to hone in on a bug, is common.

Similar to debugging, procedures to test software compare intermediate and final results created by the program against known "good" results [61]. When comparing intermediate results, as the program executes, it is repeatedly paused (and resumed) to record and compare the state.

Resource Management. Shared systems, which are becoming prevalent with the growing popularity of cloud systems, need to ensure that all users receive their promised share of computational resources. In this process systems may periodically schedule and deschedule programs for execution, which essentially requires pausing and resuming the execution.

In some instances, like Amazon's EC2 [60] and modern mobile platforms [3], the system may simply terminate the program without rescheduling it. Even in such cases the user may want to resume the execution at another time, possibly on another system, without discarding the already completed work. This process is also akin to pausing and resuming the execution.

Fault Tolerance. System hardware, which is becoming increasingly unreliable, can suffer from transient and permanent faults, potentially corrupting or crashing programs [28, 35, 36, 99]. Yet it will be desirable that programs resume and complete as if they were not affected.

Tolerating faults requires detecting faults and then recovering from their effects [175]. One approach to detecting faults is to run temporal or spatial replicas of a program and periodically

compare the intermediate results of the replicas. Different results of the same computations, or of the entire program, at identical intermediate points across the replicas indicates presence of faults. This process involves periodically pausing (and resuming) the execution.

To recover from faults, a common approach is to periodically checkpoint a program's results at intermediate points [175]. When a fault is detected, the system "rolls" back the execution to a past known error-free checkpoint and resumes the execution, discarding the affected work. The checkpointing process pauses and resumes the execution periodically. The recovery process also relies on resuming the program, although from a different (past) point.

Efficiency. An increasingly important aspect of operating computers is executing a program efficiently on the system. In the case of multiprocessor programs, merely exposing parallelism does not guarantee efficient execution. Efficient execution requires continuously regulating the parallelism [177, 178]. Regulating the program's parallelism requires pausing and resuming parts of its execution.

Security. Interconnected and shared systems are often vulnerable to malicious manipulations. Clients who use third-party systems to conduct transactions may desire audit logs, to hold the system accountable in case of improper operations.

To analyze security breaches [15, 57] or untoward manipulations [83], forensic analysis of a program's execution periodically pauses and resumes the execution to analyze it for unexpected events. The execution may be compared with precomputed expected results, or against a spatial or temporal replica.

Other Cases. Pausing, analyzing, and resuming a program's execution is a common idiom that also arises in other aspects of system use. For example, emerging proposals like approximate computing [17], near-margin operation of hardware[81, 82, 113, 188], etc., can affect a program's execution by producing erroneous results or crashing programs. These proposals hold promise, but if they hinder the program's functioning, their utility will be limited. If ways can be found to

recover from their effect and produce acceptable results, they may meet better success. Design-assist solutions, similar to those needed to tolerate hardware fault, can help these proposals.

An execution that is *interruptible*, *deterministic*, and *intuitive*, can prove beneficial in all of the above scenarios. A program's execution is interruptible if it can be suspended at a desired point to create a consistent architectural state from which the program can be resumed to complete correctly. The execution is deterministic if it generates the same results in every run for an input (Definition 2.9). The execution is intuitive if the unique sequence of assignments to each program variable can be predicted only from the program and the input. If the execution is interruptible, deterministic, and intuitive at any given point during the execution, then pausing, analyzing, and resuming the execution can be greatly simplified.

The ability to interrupt the execution, and knowing precisely what results to expect simplifies program analysis, which is needed for debugging, testing, forensics, and fault detection. Deterministic execution and intuitiveness help establish unique expected results based only on the program and the input. Interruptibility, in addition to pausing the program, also ensures that once the program resumes, its integrity will be preserved. All of these aspects simplify the implementation of the design-assist solutions. As we expound with the help of the conventional sequential program, order naturally yields these properties.

3.2 Sequential Execution

Consider the sequential execution of programs on a microprocessor. We present the properties of sequential execution to highlight the difficulties designers face when these properties are lost.

The dynamic sequence of computations of a sequential program reflects the order specified in the program[1], where a computation may be a function, or a basic operation, or any arbitrarily long contiguous sequence of operations. We introduce the notion of four properties, **S**equentiality,

[1]Dijkstra introduced a notion of nondeterminism in sequential programs to analyze algorithms [55]. Although not commonly adopted, if so realized, the resulting program will be considered multithreaded for our purposes.

Atomicity, **F**low-control independence, and **E**xclusion, acronymed **SAFE**, which the sequential execution exhibits even when the execution is parallel. We define them as follows:

1. **S**equentiality: Sequentiality is ordered execution of ordered programs (Definition 2.10), i.e., each computation appears to be performed after the preceding computations have completed.

2. **A**tomicity: All updates to the architectural state of a computation appear to take effect instantaneously, i.e., all updates are available to a succeeding computation.

3. **F**low-control independence: The flow of control at any point in the program is independent of any computation that *happens-after* [106] in the program, and is dependent only on the current architectural state of the system.

4. **E**xclusion: Each computation is performed in isolation from all other computations, i.e., during its execution, no other computation in the program can alter the architectural state.

Note that defining an order on the program's computations naturally leads to Sequentiality. Sequentiality, by definition, ensures that the execution is deterministic and intuitive for a given input (as intuitive as a sequential program can be). If interruptibility is desired, a point in the program, i.e., the position of a computation in the program, can be clearly defined. Interruptibility can then be implemented using appropriate mechanisms, as modern processors do. Consequently, Sequentiality simplifies the implementation of the design-assist solutions.

Sequentiality also simplifies programming. A program may be developed from a collection of modules, each developed independently. Once developed, any part of the program may be analyzed using the ground facts as established *before* its start. This makes the program naturally composable, and the execution easier to reason.

Atomicity, Exclusion, and Flow-control independence follow from Sequentiality. The appearance of executing operations in an order implies that each completes before the next is executed, each executes in isolation from others, and no "future" computations affect a computation's results. But

we list these properties separately since they become relevant when the execution loses Sequentiality, as we see next.

3.3 Nondeterministic Execution

Nondeterministic execution poses challenges to both programming and design-assist solutions since it lacks the desired SAFE properties.

The semantics of multithreaded programming abstractions permit the program's state and the order in which the state is modified to differ from run to run. The execution schedule, and hence the sequence in which variables are accessed and computed, is influenced by the program, the input, as well as system-specific parameters, such as OS scheduling and inter-processor communication latencies. Hence, across runs program variables can receive different values, possibly in a different sequence, and in different order relative to each other.

Moreover, nondeterministic algorithms permit the program's state to diverge across runs. For example, in the single-source-shortest-path algorithm [88], the intermediate program state can be different across runs even if the final results are identical. In yet other algorithms, e.g., the Delaunay mesh refinement [42], even different final results are acceptable as a solution to the problem. Multithreaded programs permit programmers to exploit these algorithmic properties, which they often do in search of performance. The resulting property of the program's execution, nondeterminism, complicates pausation, analysis, and resumption of the program.

Since multithreaded programs are devoid of a total order, a point in the program may only indicate a position locally within a computation, but its global position in the entire program is moot. Moreover, parallel execution spatially disperses the program's state and the execution contexts across multiple processors. Contexts may communicate with each other, and the communication is not instantaneous. Due to these factors, interruptibility, i.e., pausing the execution at a desired point, is neither clearly definable, nor straightforward.

Since the execution schedule and the sequence in which variables receive values is nondetermin-

istic, the execution may not be ordered, i.e., lacks Sequentiality, especially at intermediate points across runs. Understanding nondeterministic execution requires reasoning about the possible parallel execution schedules of the computations and their dynamic interleavings, which can be inordinately very large in number [109]. Hence the program's results, whether final or intermediate, are not predictable or intuitive, or at least not as predictable and intuitive as of a sequential program.

3.3.1 Design-assist Solutions

Given that the execution is not easily interruptible, deterministic, and intuitive, design-assist solutions take the following approach to pause, analyze, and resume multithreaded programs. To pause an execution they typically enforce system-wide barriers to first quiesce the execution, either from within or without the program, an aspect we explore further in Chapter 8. This approach ensures that the observed results are consistent with execution, and the execution can be easily resumed. However, it gives only a coarse-grain control to manage the execution, which can only be applied sparingly due to its performance impact. Moreover, it is intrusive and can alter the execution and its results, which may be unhelpful in some use-cases, like debugging. Furthermore, the intervening execution is still nondeterministic.

When it comes to analyzing and reasoning the execution, programmers are largely left to their own ingenuity and skills, despite the availability of debugging tools [41, 170].

3.3.2 Programming

In addition to developing the parallel algorithm to solve a problem, multithreaded programs may require the programmer to explicitly coordinate their parallel execution, which increases the potential sources of bugs [118]. Since multithreaded programs explicitly permit at least a subset of the computations (user-designated) to execute in any order, they may violate the SAFE properties. We exposit further with the help of examples from real-world applications, which were presented by Lu et al. in their analysis of concurrency bugs [118].

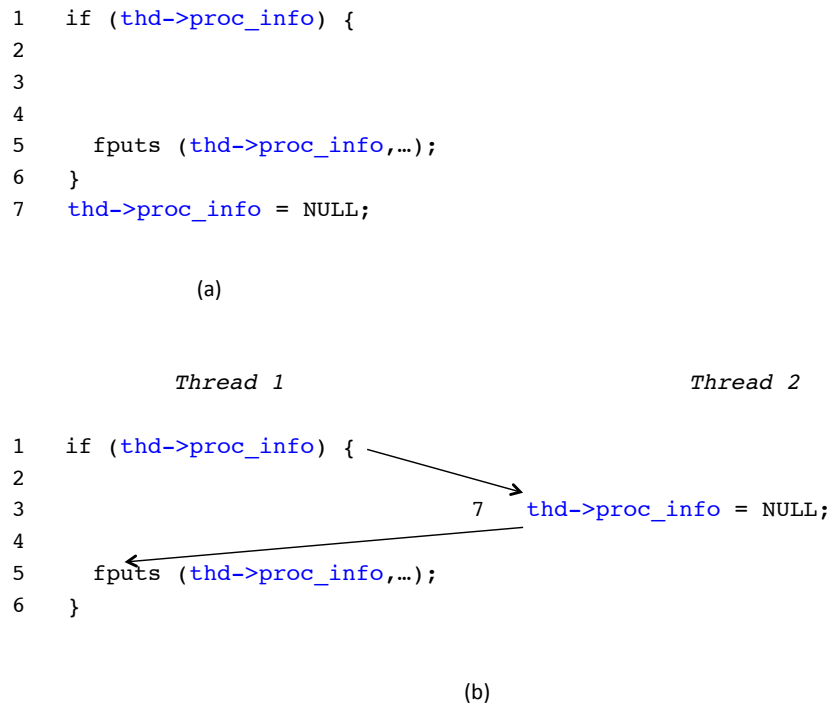


Figure 3.1: (a) A sequence of operations in MySQL. (b) Parallel implementation: lines 1-6 and line 7 are divided across two threads. Arrows depict the execution sequence that violates Exclusion of operations in thread 1. Operation on line 7 in thread 2 can influence the operations of thread 1.

In multithreaded programs, operations within a computation follow the sequential order, but not across computations. Hence Sequentiality is violated. Violation of Sequentiality can produce erroneous results if dependent computations are not executed in the correct order (producer $\rightarrow$ consumer).

Concurrent computations can access common data, leading to data races and potentially impacting each other's execution. If they do, Exclusion is violated. For example, consider the operations in MySQL, as shown in Figure 3.1a. Operations on lines 1-6 are expected to be performed in Exclusion, and complete before the operation on line 7. In the parallel implementation of these operations, shown in Figure 3.1b, Exclusion can be violated by the execution sequence depicted using arrows. Operation on line 7 can influence the operations on line 1-6 and produce incorrect results.

Further, due to loss of Sequentiality, a computation may access only partial results of another

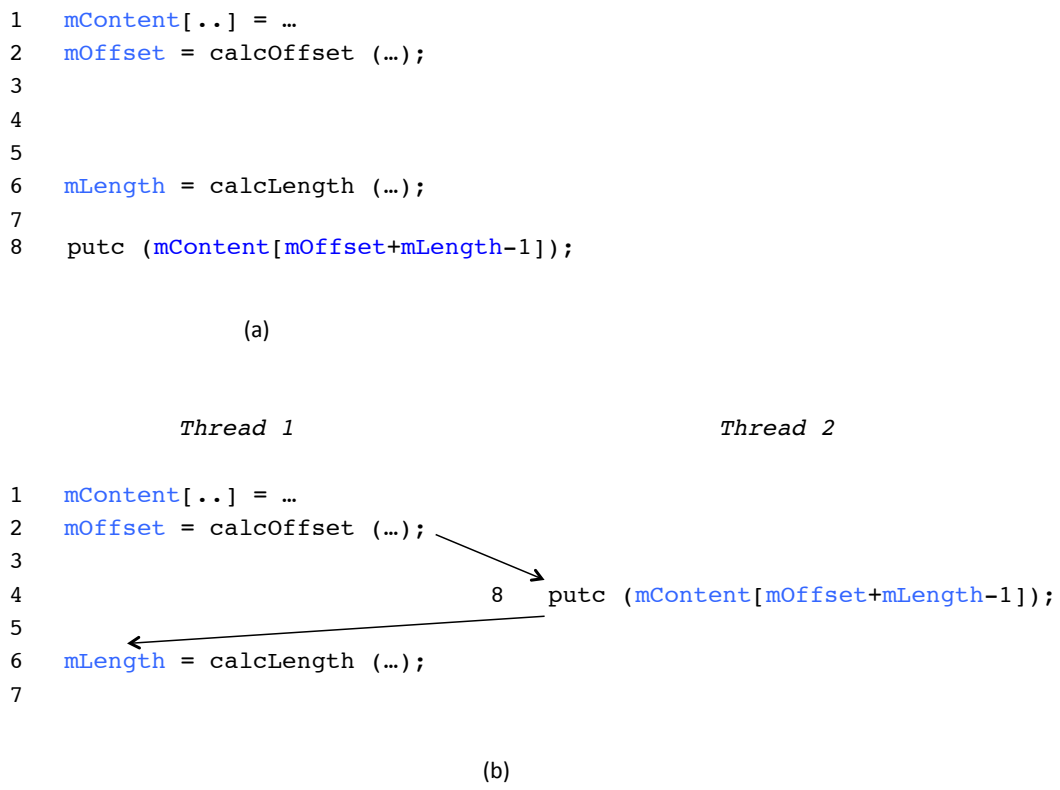


Figure 3.2: (a) A sequence of operations in Mozilla. (b) Parallel implementation: lines 1-7 and line 8 of the sequential operations are divided across two threads. Arrows show the execution sequence that violates Atomicity. `putc` on line 8 uses partial, inconsistent results from Thread 1.

concurrent, not yet completed computation, violating Atomicity. For example, consider the sequence of operations in Mozilla, shown in Figure 3.2a. Operations on lines 1-6 are expected to complete before line 8. In a parallel implementation of these operations, Atomicity can be violated by the execution sequence depicted using arrows. The operation on line 8 can consume partial results of the operation on lines 1-6. (Lu et al. have termed this as a "multi-variable concurrency bug".)

Loss of Sequentiality can also lead to loss of Flow-control independence. Consider another sequence of Mozilla operations in Figure 3.3a, and its parallel implementation in Figure 3.3b. Line 4 assumes that line 2 has completed. But in the parallel design the flow of control must reach line 4 after line 2 in Thread 1 happens, as shown by the dashed arrow. But if the actual sequence is as

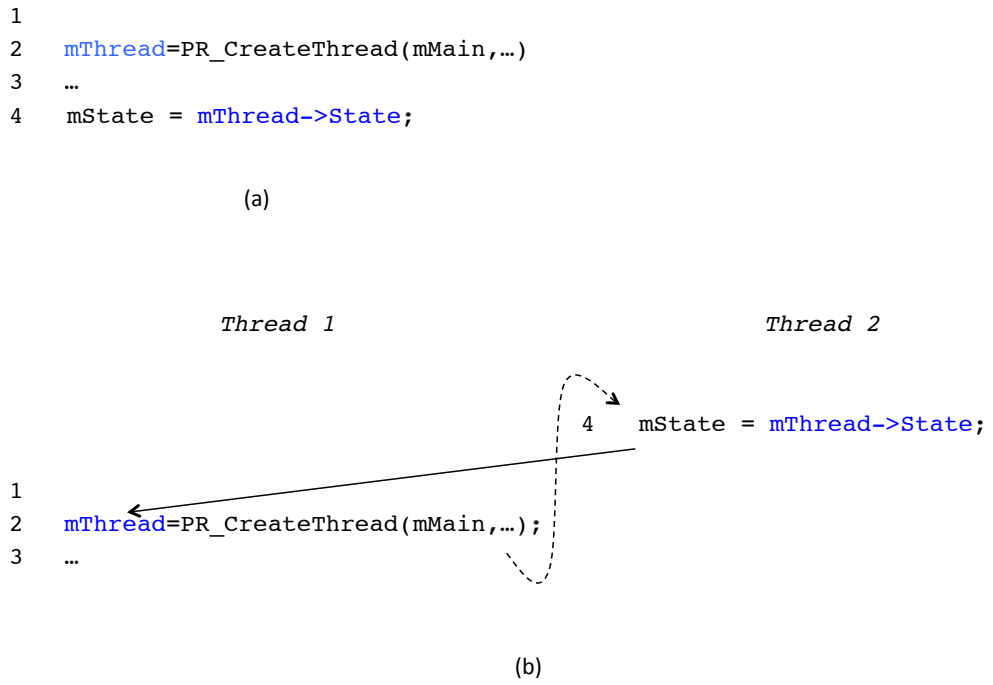


Figure 3.3: (a) Another sequence of operations in Mozilla. (b) Parallel implementation in which line 2 and line 4 are performed on different threads. Dashed arrow shows the desired flow of control; solid arrow shows the sequence that can result due to the loss of Flow-control independence.

shown by the solid arrow, the Flow-control independence is violated. (Lu et al. have termed this as "order violation".)

Needless to say, violation of Sequentiality, Exclusion, Atomicity, and Flow-control independence can produce erroneous results [118]. To restore these properties in the desired parts of the multi-threaded program, programmers often serialize accesses to the shared data, or create critical code regions, or control the execution sequence. This is achieved by imposing an order, possibly arbitrary, on the involved operations by means of regulating the control-flow in the respective computations, e.g., by using locks. If performed incorrectly, Sequentiality, Exclusion, and Atomicity may still be violated.

Locking can introduce other complications. When one computation waits for the lock to be released, Flow-control independence is violated since the computation's execution can proceed only

after the lock is released, which is expected to *happen after*. The loss of Flow-control independence can lead to livelocks or deadlocks if the locks are used carelessly. This affects the composability of multithreaded programs.

Locking can also have performance implications. Serializing needlessly large portions of computations can defeat the very purpose of paralleling the program. Hence careful use and placement of locks becomes necessary to obtain the desired execution property without compromise performance.

In summary, the lack of SAFE properties makes multithreaded programs difficult to develop and analyze. Multithreaded programs abandon the properties for the sake of performance, but then need to explicitly restore them in parts, for correctness. Explicitly doing so is non-trivial. Unfortunately, tools can only be of limited assistance since automated analysis of parallel programs is provably undecidable [147].

3.4 Deterministic Execution

Deterministic execution yields some of the desired properties, and can be helpful depending on the programming abstraction used. Since deterministic execution orders accesses to a program variable, the execution exhibits Atomicity and Exclusion. However, if the program is written using Multithreading, the programmer still has to develop the program keeping nondeterminism in mind, which is still non-trivial. Moreover, the execution, although repeatable, is not intuitive. Hence it is unclear how useful this "after-the-fact" determinism is. On the other hand, if ordered programs are used, the deterministic execution can be more intuitive.

Nonetheless, irrespective of the programming abstraction, deterministic execution lacks Sequentiality. It is not interruptible.

3.5 Ordered Execution of Ordered Programs

Given the above, wide-ranging impact of the properties of a program's execution on programming and using multiprocessor systems, we argue for an approach that first simplifies these aspects and then seeks methods to exploit parallelism. We aim to obtain the desired SAFE properties, but without compromising performance or complicating programmability and usability in other ways. Inspired by the success of the sequential programs and their sequential execution in the realm of uniprocessors[2] , we argue for ordered execution of ordered programs on multiprocessors.

We distinguish ordered programs from sequential programs. Whereas sequential programs are composed for execution on a single processor, ordered programs are composed for multiprocessors. In addition to solving the problem at hand, ordered programs need to account for parallelism, as will be seen in the next two chapters. Although composed for parallel execution, the order in ordered programs may come from different sources. One convenient source is the program's text, analogous to sequential programs. Another source can be the sequence of asynchronous events processed by concurrent programs. We envision that an ordered execution will respect the order as the system defines it, and hence is distinct from sequential execution which, as is generally understood, follows only the program's sequential order.

By performing an ordered execution, and using an user-friendly, well-defined order, we can obtain Sequentiality, Atomicity, Flow-control independence, and Exclusion.

[2]The notion of sequential operations to manipulate numbers was first introduced by Charles Babbage in 1837 [117]. Sequential programming has survived almost eight decades of remarkable advances in computer science since Alan Turing formulated the notion of sequential manipulation of symbols (on an infinite tape) in his *automatic machine*, now popularly known as the "Turing" machine, in 1935 [182].

4 Parakram

Parakram *(English)*
Pronunciation*: para (as in para-legal) - crum (as in ful-crum)*
Etymology*:*

2010, Parakram is a portmanteau, derived from the English Word "parallel" and the Sanskrit word "anukrama": para+kram

Parallel: A line that runs side by side with and equidistant from another, things running side by side

Anukrama (Sanskrit, अनुक्रम): Succession, due order, sequence
Noun

1. A system to concurrently perform tasks drawn from a sequence
Adjective

2. Of or pertaining to parakram

parakrama *(Sanskrit, पराक्रम)*
Pronunciation*: per (as in per-haps) -a (as in o-ut) - cru (as in cru-x) - ma (as in dog-ma)*
Verb

1. To march forward, advance, excel, distinguish one's self

—

In Chapter 3 we hypothesized that ordered execution of ordered programs can simplify programming and the use of multiprocessor systems. Over the next five chapters we test this hypothesis.

Conventional parallel programming has a long legacy. We believe that any new proposal must measure up to it to be considered viable. Specifically, any new approach must match the conventional approach in expressing and exploiting parallelism. The new approach must meet three criteria: (i) it must match the conventional approach in expressing parallel algorithms, (ii) it

must obtain matching performance, and (iii) it must not complicate programming and usability in other ways.

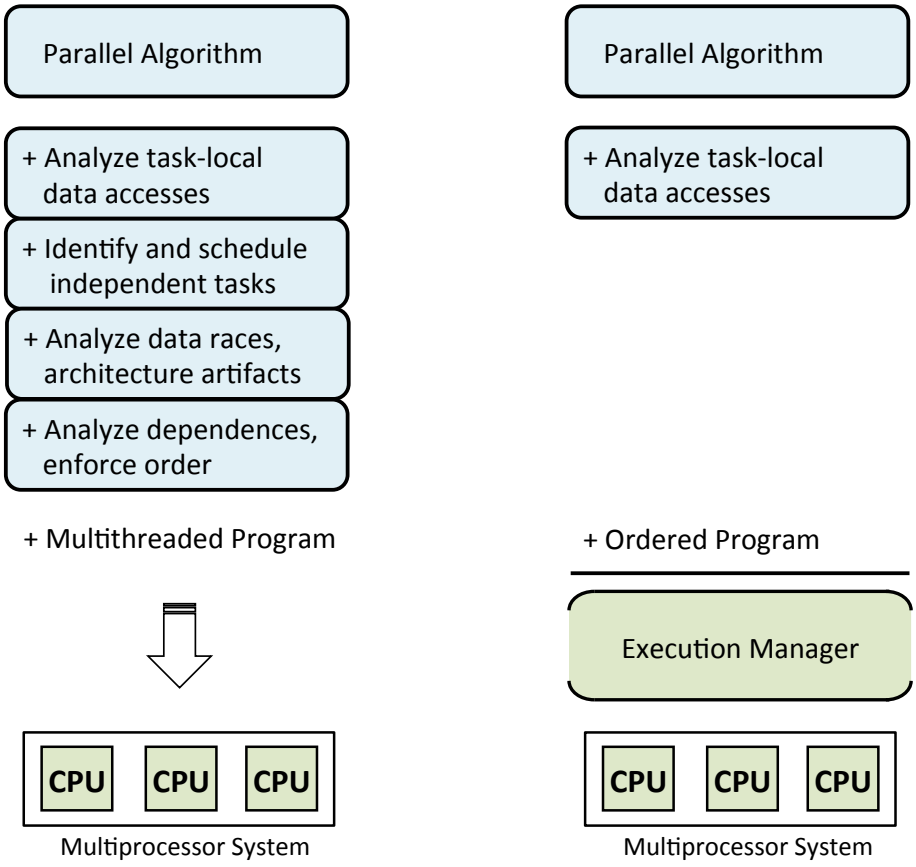
We draw several lessons from the modern out-of-order superscalar processors in our work. Based on these lessons we summarize the proposed general approach to multiprocessor programming and how it compares with conventional parallel programming in Section 4.1. We applied the lessons to formulate Parakram, a model to express and execute ordered programs on multiprocessors. This chapter overviews Parakram. Parakram's model to perform parallel, but ordered execution, and a companion programming model are presented in Section 4.2. We developed a practical Parakram prototype to evaluate its efficacy against the three criteria. The prototype implements the programming and execution models. Its overview is presented in Section 4.3.

4.1 The Parakram Approach

Over a period of many years modern out-of-order (OOO) superscalar processors have progressively scaled performance of sequential programs. Programs remained sequential, but processors executed the program's instructions in parallel. Architects introduced mechanisms, such as dataflow execution and speculation, to exploit the parallelism, but always provided an ordered view of the execution to the programmer. Such an abstraction permitted the architects to introduce many performance-enhancing innovations transparently to the programmer. Further, system software developers used architectural features, like precise interrupts, to introduce usability-enhancing innovations, like handling page faults, also transparently to the programmer. We apply a similar paradigm to multiprocessors.

4.1.1 Programming Multiprocessors

In conventional parallel programming, the programmer develops a parallel algorithm and also explicitly manages its concurrent execution, as depicted in Figure 4.1a. The algorithm considers and exposes parallel computations, in the form of tasks, to operate on data concurrently. Programmers



(a) The conventional approach.

(b) The Parakram approach.

Figure 4.1: Overview of multiprocessor programming approaches.

reason about the dynamic task-local data accesses. Using this analysis they identify and schedule independent tasks for parallel execution. They reason about the dynamic interactions of tasks with each other and the underlying system's artifacts, such as the memory consistency model. They coordinate the task execution to avoid data races, enforce order to respect dependences, account for the system artifacts, and maximize the parallelism. In the process, they may use a range of concurrency control mechanisms. This recipe is then statically baked into a multithreaded program. Explicitly managing the program's parallel execution makes parallel programming onerous. Relatively newer proposals simplify one aspect or another of the above process, but fail to address the problem in entirety. They are summarized in Chapter 9.

We envision a simpler approach to multiprocessor programs (Figure 4.1b). Programmers will still develop parallel algorithms, often similar to the ones they already do in the conventional approach. They will reason about the task-local data accesses, but the actual management of the concurrent execution will be abstracted away from them. They will not need to reason about dynamic independent tasks, data races, system artifacts, or the inter-task dependences. Neither will they need to coordinate the execution. Instead, they will express the computations as a list of tasks in an object-oriented, annotated ordered program. Then under the hood, an **execution manager** will intercept the program, and dynamically manage its ordered, parallel execution with the help of the program annotations, transparently to the programmer. Thus we decouple the expression of a parallel algorithm from its execution, easing much of the burden on the programmer. The execution manager will also provide a convenient facility to introduce new techniques to meet new objectives, also transparently to the programmer, one example of which we will present later in the dissertation. This approach then forms the basis for the Parakram model.

4.2 The Parakram Model

Parakram incorporates two main components: an execution model and a programming model. The execution model facilitates the execution while providing a simple interface to the programming model. The programming model facilitates the execution model while providing a simple interface to the programmer.

4.2.1 The Execution Model

Parakram uses the time-tested principles commonly employed by the microprocessor to execute programs. Consider the execution of a program in a modern OOO superscalar microprocessor, as depicted in Figure 4.2. The processor fetches a sequence of instructions of the sequential programs and creates an *Instruction Pool*. A *Speculative Dataflow Unit* creates a dynamic dataflow graph of the instructions using their order (sequential) and operands, e.g., the register names embedded

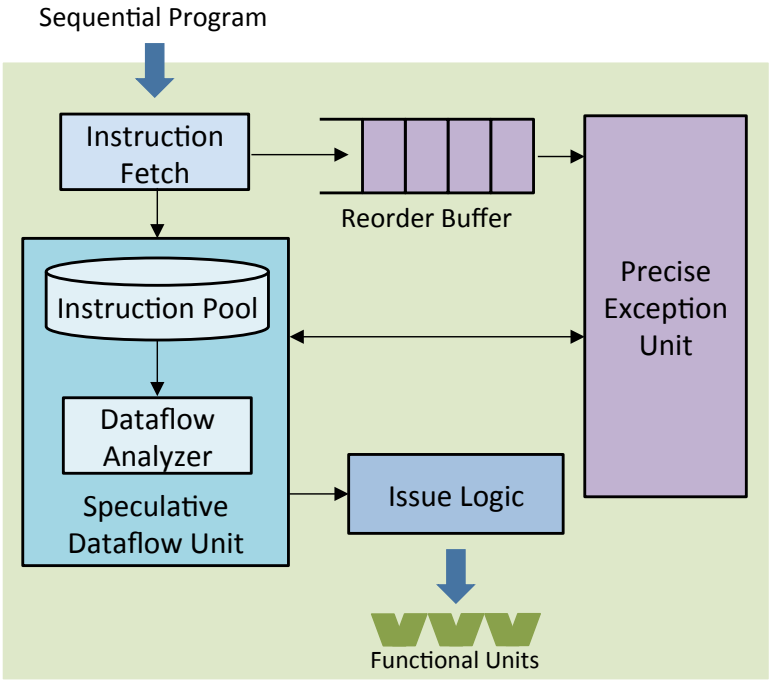


Figure 4.2: Block diagram of an out-of-order superscalar processor.

in the instructions or the memory locations accessed by the instructions. From this pool it *issues* multiple ready instructions for execution to functional units, possibly out of order, thus exploiting parallelism.

To maintain an appearance of ordered execution, the processor tracks the order of in-flight instructions and the state they modify. It uses two features to do so. It logs the incoming instructions into the *Reorder Buffer* (ROB) in the program order. Results from an instruction are temporarily held separately from the architectural state, in the ROB, or a history buffer, or a future file [171], or their variants [172], until the instruction is ready to *retire*. A *Precise Exception Unit* waits for the oldest instruction to complete, which is conveniently always at the head of the ROB, and retires it by committing its results to the architectural state. Thus the architectural state always appears to be updated in the original program order. If a breakpoint or an exception were to be encountered, the Precise Exception Unit marshals the execution, and uses the instruction order and the separately

held state to ensure that the architectural state reflects the execution precisely up to that point.

As instructions complete and retire, additional instructions from the program are added to the Instruction Pool. The Speculative Dataflow Unit reconstructs the dataflow graph, accounting for the completed and the new instructions, and advances the execution as above.

The Speculative Dataflow Unit (SDU) does not always have the complete or accurate information to immediately build a complete or accurate dataflow graph, e.g., due to temporarily unknown memory locations accessed by instructions. To prevent the loss of parallelism, the SDU will often issue subsequent instructions *speculatively*, assuming that they are independent of the previous instructions. Speculative instructions are tracked in the ROB. When subsequently the necessary information becomes available, the SDU completes or corrects the dataflow graph and checks whether it had *misspeculated*. If it had, the misspeculation is treated as an exception. The Precise Exception Unit rectifies the misspeculation by undoing the operations performed by the misspeculated instructions, and resumes the correct execution.

Thus the microprocessor exploits instruction-level parallelism in a sequential execution.

Parakram maps an analog of this execution model to multiprocessors. First, it replaces the functional units with hardware execution contexts (processors or threads). Next, it replaces the sequential program with an ordered program, composed as described before. To match the scale of multiprocessors, it executes coarser-grained tasks, instead of instructions, as a unit of execution on a context. It analyzes the dataflow between the tasks and schedules them for execution in the out-of-order dataflow fashion.

To effect the dataflow execution, Parakram fetches tasks, ascertains their operands, builds a task dataflow graph, and issues ready tasks for execution. Parakram introduces **globally-precise interrupts**, analogous to precise interrupts, to give an appearance of ordered execution. As we shall see, like in the microprocessor, on occasion the dataflow between tasks may be temporarily unknown. In such instances Parakram may execute tasks speculatively to maximize the parallelism. To perform speculative execution correctly, Parakram incorporates features to detect and rectify misspeculations, analogous to the microprocessor.

In summary, the execution model effects speculative, dataflow execution by relying on the knowledge of tasks, their order, and their operands. The programming model provides it this information.

4.2.2 The Programming Model

To enable its execution model, Parakram employs a programming model with matching capabilities. It leverages the programming practices developers use, raises the level of abstraction to match the abstraction at which the execution model operates, and takes assistance from programmers without overly burdening them.

Programmers today follow modern software engineering and object oriented (OO) design principles [27]. These principles encompass modularity, OO designs, data encapsulation, and information hiding. Programs are composed from reusable modules in the form of functions (or methods). Consequently functions act as self-contained computations that manipulate "hidden" data, and communicate with each other using well-defined interfaces. Manipulating global data not communicated through the interface is often avoided and hence most such computations are free from side-effects. We exploit these common practices in "well-composed" programs, and also provision for the rare "poorly-composed" program in which these practices may not have been followed (as will be seen).

To enable the execution model, we first raise the level of abstraction of a computation from an instruction to a function. We observe that the programs already comprise functions. Functions make an appropriate fit, logically as well as in granularity, for a unit of computation at the scale of multiprocessors. User-designated functions are treated as tasks by the execution model.

Second, the order of tasks, needed by the execution model to establish the dataflow, is obtained from the program's text. The dynamic invocation order of a function at run-time, together with its invocation context provide a function's order in the program, as is detailed in Chapter 6.

Third, Parakram raises the level of abstraction of a datum from a register or a memory location to an object. An object, possibly comprising multiple fields, typically forms the atomic unit of data

in modern object oriented programs.

Finally, the programming model helps the execution model to ascertain a task's operands, which are also needed to determine the dataflow. A task's operands are the data it reads, i.e., its **read set**, and the data it writes, i.e., its **write set**, also collectively called its **dataset**. In well-engineered reusable functions, the operands are often readily available from the function's interface. Parakram obtains the operands with the help of user annotations. We view a task's dataset as comprising objects. The execution model treats the dataset objects as the task's operands. Often objects in the dataset are statically unknown. The execution model enumerates the dataset dynamically, at run-time.

4.3 Parakram Prototype

To evaluate the ordered approach we developed a fully functional Parakram prototype. Figure 4.3 shows the logical view of the prototype as it relates to the model. The prototype mimics the model and also comprises two components: a programming interface and an execution engine.

The programming interface provides APIs for the programmer to express parallel algorithms. The APIs permit programmers to annotate tasks for parallel execution, identify their datasets, and help guide the execution. Traditional dataflow machines have relied on functional programming languages to express programs. However, imperative languages have gained more popularity. Hence we adopt the more familiar and established C++ as the programming language for our prototype. Other languages, imperative or functional, may be used just the same.

A key aspect of the programming model and its actual implementation is to ensure that they match multithreaded programs in expressing parallel algorithms. This forms an important part of our study and Parakram design, described in Chapter 5.

The second component, the execution manager, is implemented as a software runtime library. In addition to parallelizing the execution, it enables the parallelism idioms expressed in the program. The library can operate on stock multiprocessor systems. No additional hardware support is needed

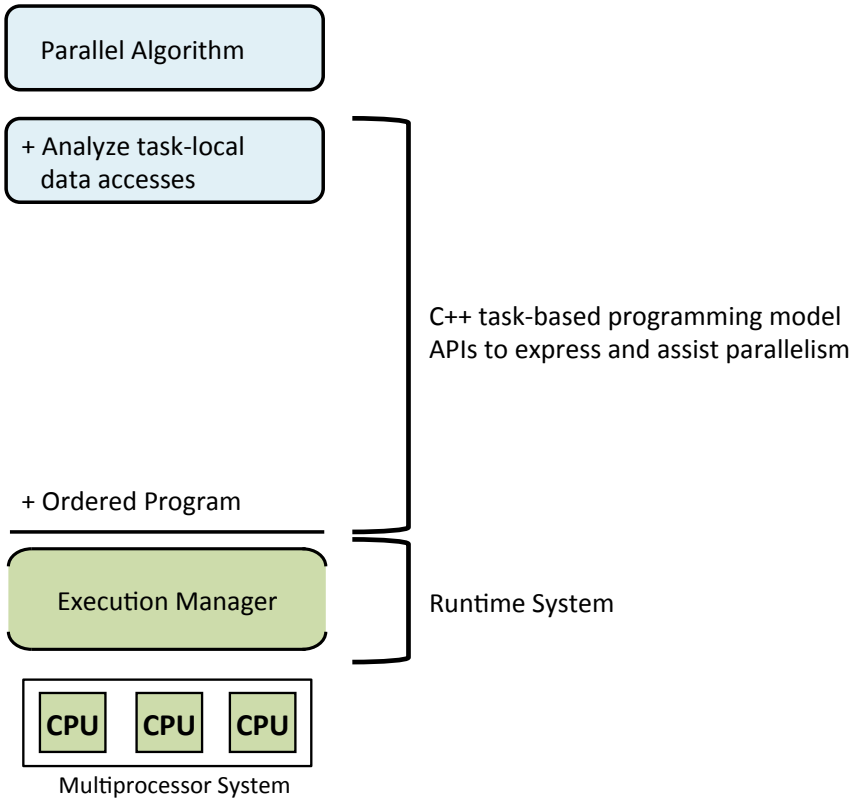


Figure 4.3: Parakram prototype. It comprises a C++ programming interface and a runtime system. The programming interface provides APIs to develop ordered programs. The runtime parallelizes their execution on a multiprocessor system.

for its functioning.

Implementing the execution model presented several challenges. In its design, we addressed the following key questions in the context of a multiprocessor program (Chapter 6):

1. How will a sequence of tasks be fetched from the program?
2. What are a task's operands, and how will they be identified?
3. How will the dataflow graph be constructed?
4. How will a task be "issued" for execution?
5. What is the meaning of precise interrupts in a parallel execution?

6. How will an ordered view be provided?

7. Is speculation needed?

8. If needed, how will speculation be supported, especially when a task's state can be very large?

4.4 Summary

In summary, we have proposed that: programmers express parallel algorithms as ordered programs, they annotate the program with task-local information, and an execution manager will parallelize the program's execution with no further help from the programmer.

Can this approach, based on order, express and exploit parallelism to the same degree as multithreaded programs? Can the seemingly artificial ordering constraint be overcome? Does it simplify programming and system use in practice? We answer these questions in the following chapters.

5 Ordered Programs

I decided long ago to stick to what I know best. Other people understand parallel machines much better than I do; programmers should listen to them, not me, for guidance on how to deal with simultaneity.

— Donald Knuth

In Chapter 4 we presented the overall Parakram approach to multiprocessor programs. One of the two aspects of the Parakram approach is its programming model. In this chapter we expound the programming model more concretely by describing the details of the programming interface provided by the Parakram prototype.

Our primary objective with ordered programs is to simplify multiprocessor programming and improve usability, but without compromising the ability to express parallel algorithms. Using ordered programs raises the question whether imposing order fundamentally prevents them from expressing parallel algorithms that can otherwise be expressed in multithreaded programs. In theory, a parallel Turing machine, as a multithreaded program might represent, is only as powerful as a sequential Turing machine (ordered program) in expressing computations [186]. Expressiveness then boils down to the capabilities of specific programming abstractions. Hence we need only ask, in practice what facilities does the multithreaded programming abstraction provide, and can an ordered approach match them?

To answer the above question, we study the common parallelism patterns that arise in multithreaded programs and apply the lessons to ordered programs. With an appropriate programming interface, supported by a matching execution manager, we ensure that ordered programs can express similar patterns.

Although parallelism patterns have been studied in the past, past studies were inadequate for our analysis since their perspectives do not reveal the details germane to our work. Hence, based on our study of the popular multithreaded programming APIs, past studies on patterns, and a range of parallel programs, we formulate **MAPLE**, a multiprocessor programming pattern language[1]. MAPLE and the pattern analysis are presented in Section 5.1. Based on the observed patterns, we then formulate the Parakram APIs that admit similar patterns in ordered programs. The APIs are described in Section 5.2. In Section 5.3 we present example programs authored using Parakram APIs and compare them with equivalent multithreaded programs. Ordered programs, despite implementing parallel algorithms, closely resemble the sequential counterparts, unlike the multithreaded programs, are easier to develop, and provide similar capabilities.

5.1 Parallelism Patterns

Design patterns are an established way to organize and understand recurring design structures in engineering fields [7, 72]. We use a similar approach to study multithreaded programs.

To solve a problem on a multiprocessor system, a programmer first uses domain expertise to develop a solution. The solution is transformed into a parallel algorithm, which is then expressed as a multiprocessor program. Multithreaded programs also explicitly manage the execution by using concurrency mechanisms.

We observe that problem domains, and algorithms needed to solve them, are orthogonal to the programming interface. As long as multithreaded and ordered programs use similarly capable languages they should be equally capable of expressing the basic algorithm needed to solve the problem. The differences between the two might arise in the concurrency mechanisms. Multithreaded programming abstractions, e.g., Pthreads, provide a well-defined repertoire, and it is unclear how ordered programs, with the added constraint of order, can provide similar capabilities.

Others have classified design patterns that arise in multithreaded programs [103, 122, 131, 142].

[1] "Language" is a standard term others have used to organize patterns; we use the same term in this dissertation.

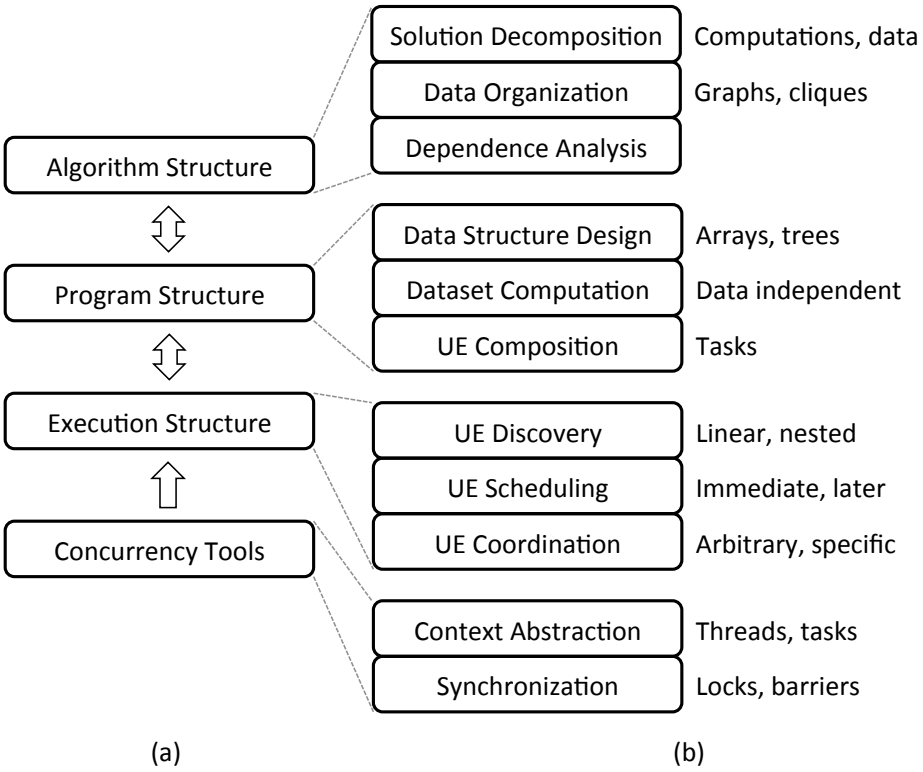


Figure 5.1: MAPLE: pattern language for multiprocessor programs. (a) Design spaces. (b) Sub-space within each design space, with examples.

They take a literal view of the program, in which the program intertwines the algorithm needed to solve the problem with the coordination needed to parallelize its execution. Consequently, the patterns they formulate present a composite view of the problem domain, algorithm choices, and concurrency mechanisms. These patterns do not clearly articulate the concurrency mechanisms, which are of primary interest to us. Hence we develop a pattern language that isolates them.

To better understand the concurrency management capabilities of multithreaded programs, we separate the algorithm from its coordination. MAPLE draws inspiration and some terminology from the prior work, but organizes the parallelism patterns by taking a more elemental view of how concurrency is achieved and managed, and how concurrency interacts with the algorithm itself. It distills the core capabilities from common design patterns and logically organizes them into four *design spaces*, as shown in Figure 5.1a, which represent the key aspects of developing

```
1  ...
2  while blocks still in file do
       /* Create a pipeline of read, compress, and write tasks */
3      task read(block, input file)
4          block ← readFile(input file)
5      end
6      task compress(block)
7          block ← compressBlock(block)
8      end
9      task write(block, output file)
10         output file ← writeFile(block)
11     end
12 end
13 ...
```

Algorithm 5.2: Parallel bzip2 algorithm.

parallel programs. To solve a problem, the programmer develops a parallel *Algorithm Structure*, encodes the algorithm into a *Program Structure*, and develops an *Execution Structure* to coordinate the algorithm's dynamic execution, using *Concurrency Tools*. Typically, programmers iterate over this process to reach a satisfactory design. Multiple patterns arise in each design space. Depending on their purpose, we group the patterns into *sub-spaces* within each design space, as shown in Figure 5.1b along with examples in each sub-space.

We highlight the patterns using three algorithms, parallel bzip2 (Pbzip2) [74], dense matrix Cholesky decomposition [144] and Delaunay mesh refinement [42]. We examine the parallelism opportunities in the three algorithms and the related patterns.

Parallel bzip2 is the parallel implementation of the popular bzip2 compression utility. Algorithm 5.2 shows the pseudocode of the main kernel of the algorithm. It iteratively reads blocks from an input file (line 3) to be compressed, compresses the blocks (line 6), and writes the results to an output file (line 9). Parallelism arises when the three operations are performed concurrently on different blocks.

Cholesky decomposition (CD), a popular scientific kernel, is mainly used to solve linear equations of the form $Ax = b$. A is a a Hermitian, positive-definite matrix, and b is a vector. Cholesky

```
1  Data: matrix M[blocks]
2  ...
3  for i ← 1 to #blocks do
4  |  ...
   |  /* Operate on sub-matrices concurrently */
5  |  task B()
6  |  |  M[i] ← cd(..)
7  |  end
8  end
9  barrier
10 ...
11 for j ← i + 1 to #blocks do
12 |  ...
13 |  task A()
14 |  |  ...
   |  |  /* Operate on sub-matrices concurrently */
15 |  |  M[j] ← mmult(..)
16 |  end
17 end
18 barrier
19 ...
```

Algorithm 5.3: Iterative Cholesky decomposition algorithm

decomposition factorizes the matrix into a product of lower triangular matrix and its conjugate transpose, $A = LL^T$, then forward substitutes to solve $Ly = b$, and finally solves $Lx = y$ by back substitution. Parallelism in the algorithm arises from computing sub-matrices concurrently. The matrix may be decomposed into sub-matrices geometrically and processed iteratively [141], or decomposed and processed recursively [123]. Algorithms 5.3 and 5.4 show the respective algorithms. In both cases multiple instance of tasks, as invoked in a loop in Algorithm 5.3 (lines 5, 13), and recursively in Algorithm 5.4 (lines 6, 7). Inter-task dependences arise in Cholesky decomposition which need to be respected for correct results.

Delaunay mesh refinement (DeMR) pertains to mesh generation, a popular engineering application [42]. Mesh generation tesselates a surface or a volume with shapes such as triangles, tetrahedra, etc. DeMR ensures that the triangles in a given tesselated mesh meet a specified quality criteria. DeMR refines "bad" triangles in the mesh (Algorithm 5.5). The mesh is maintained as a graph,

```
1  Data: matrix M[blocks]
2  ...
3  task B(M, blocks)
4  |  ...
5  |  if blocks > 1 then
   |  |  /* Recursive call to task B */
6  |  |  task B(M, blocks/2);
7  |  |  task B(M, blocks/2);
8  |  |  ...
9  |  else
10 |  |  ...
11 |  |  task A()
12 |  |  |  ...
13 |  |  |  M[j] ← mmult(..)
14 |  |  end
15 |  |  ...
16 |  end
17 end
18 ...
```

Algorithm 5.4: Recursive Cholesky decomposition pseudocode.

```
1  Data: Graph G
2  workList W ← bad △ (G)
3  while W ≠ empty do
4  |  ...
5  |  task A(item)
6  |  |  △ ← W[item]
7  |  |  W = W − △
8  |  |  C ← cavity(G, △)
9  |  |  new N ← refine(C)
10 |  |  G = G − C
11 |  |  delete C
12 |  |  G = G + N
13 |  |  W = W + bad △ (N)
14 |  end
15 |  ...
16 end
17 ...
```

Algorithm 5.5: Delaunay mesh refinement algorithm.

whose nodes represent triangles. Bad triangles are queued in a work list (line 2). Refining a triangle causes the triangle and any adjacent triangles within a region around the bad triangle, called its *cavity*, to be modified (line 8). This process may delete the cavity from the graph (line 10), and add new triangles to the mesh (lines 9, 12). Some of the new triangles may also be bad, which are added back to the work list (line 13). The algorithm iterates until no bad triangle is left. Parallelism arises from refining (non-conflicting) triangles concurrently, as depicted by task A in the algorithm (lines 5-14). DeMR is a part of the Lonestar benchmark suite [104].

We now describe the MAPLE patterns with the help of these examples. Wherever applicable, we put the MAPLE patterns in context with patterns from the related literature.

Algorithm Structure. The Algorithm Structure captures the process of developing a parallel algorithm from a domain-specific solution. There can be multitudinous domain-specific patterns [103, 142], and they can be grouped in this design space. However, the Algorithm Structure is not our primary focus since algorithm construction and related aspects are independent of a given programming abstraction.

Briefly, the solution is first decomposed into potentially concurrent computations (sub-space *Solution Decomposition*). There are two primary sources of parallelism. The first comes from the computational steps involved in solving the problem; different steps may operate concurrently, e.g., the three operations in Pbzip2 (Algorithm 5.2) can be performed concurrently on different blocks. The second comes from decomposing data such that the sub-units may be computed concurrently, e.g., the matrix M in CD (Algorithms 5.3 and 5.4) can be divided into sub-matrices and computed concurrently.

Next, abstract data structures, e.g., a graph in the case of DeMR (Algorithm 5.5), best suited to permit and expose the concurrency are identified (sub-space *Data Organization*). Also, dependences are analyzed by examining data-sharing and data-flow between the computations (sub-space *Dependence Analysis*). For example, in Pbzip2, for a given block, the `write` computation is dependent on `compress`, which is in turn dependent on `read`.

Program Structure. The Program Structure captures the process of transforming the algorithm structure into a program. This includes designing data structures and composing computations. A key part of this process is computing the identity of data that will be processed.

Appropriate concrete data structures, such as trees, queues, etc., to match the algorithm are chosen (sub-space *Data Structure Design*). CD in our examples uses a two-dimensional array to hold the matrix. Further, data may be allocated *statically*, or *dynamically*. Declaring an array to hold the matrix in CD (Algorithm 5.3, line 1), whose size remains static is an example of the former. The graph that grows dynamically in DeMR (Algorithm 5.5, lines 9, 12) is an example of the latter.

Before a computation can process data, it must first identify them, i.e., compute its dataset (sub-space *Dataset Computation*). The dataset may be *data-independent* or *data-dependent*. Data-independent datasets can be identified without examining the data itself, e.g., by using index-based referencing, as is done in CD (Algorithm 5.3, line 6, 15). By contrast, data-dependent datasets require examining the data first, e.g., DeMR first traverses the graph, i.e., reads the nodes, to locate the nodes that will be operated on (Algorithm 5.5, line 8).

Finally, the computations identified in the Algorithm Structure are organized into *units of execution* (sub-space *UE Composition*). An unit of execution (UE) forms a logical computation, e.g., the **task**s in the three algorithms, that is submitted for execution to a hardware context.

Execution Structure. Once the algorithm is encoded, the Execution Structure captures the process of exposing and coordinating its parallel execution. "Task parallelism", "geometric decomposition", "loop parallelism", "fork-join", "divide-and-conquer", "recursive data", "pipeline parallelism" , "master-worker", "event-driven coordination", and "work queue" are examples of patterns that arise in this process [102, 122, 123]. We distill these patterns into three basic elements: (i) exposing UEs at run-time, (ii) scheduling them for execution, and (iii) ordering them or their operations, when required.

When a parallel program executes, UEs may be discovered or invoked from the program (sub-space *UE Discovery*) in two ways for potentially concurrent execution with other UEs. They may

be discovered *linearly*, i.e., from a single context, or in a *nested* fashion, i.e., invoked from other UEs. Task invocation in iterative CD (Algorithm 5.3, line 5-7) is the example of the former. Task parallelism, geometric decomposition, loop parallelism, and fork-join parallelism are examples of such discovery. Nested task invocation in recursive CD (Algorithm 5.4, task B on line 6 is nested in task B on line 3) is an example of the latter. This pattern can arise in recursive data decomposition and divide-and-conquer algorithms.

Once discovered, executing a UE presents two choices (sub-space *UE Execution*). The UE may be scheduled for execution *eagerly*, or *lazily*. Pbzip2 and both implementations of CD schedule tasks eagerly, i.e., immediately upon invocation. Task parallelism, loop parallelism, geometric decomposition, fork-join, divide-and-conquer, and recursive data patterns typically schedule UEs eagerly. Discovered tasks may also be scheduled lazily for execution, i.e., at a later time, to honor dataflow dependences or to utilize resources efficiently. Patterns such as master-worker (as in DeMR, line 13), event-driven coordination, and work queues employ such a design.

When UEs execute concurrently, on occasion the programmer may coordinate parts of their execution (sub-space *UE Coordination*). The programmer may order UEs to: (i) synchronize accesses to shared resources, (ii) create critical regions, or (iii) respect data dependences. We saw some examples in Section 3.3.2 where such coordination was needed for correctness. There are myriad ways to coordinate, each with associated tradeoffs [124]. However, ultimately, they all impart an order to the involved UEs by serializing them at the desired points. The order may be arbitrary if it does not affect the correctness. For example, in DeMR, a user may choose to synchronize concurrent accesses to *W* on line 7 (Algorithm 5.5), in some arbitrary order since the order does not affect the correctness. However, often a specific order is enforced between the entire or specific portions of UEs, e.g., to respect data dependences. The `barrier` in CD is used to ensure that subsequent UEs are invoked only after all preceding UEs have completed (Algorithm 5.3, line 9), to ensure correctness. It is an example of specific ordering. Producer-consumer designs, pipeline parallelism, and event-driven coordination are also examples of the specific-ordering pattern.

Further, the ordering may be applied at a desired granularity, ranging from individual data

accesses to coarse-grained computations. For example, fine-grain locks may be used in DeMR to synchronize the concurrent accesses to *W* on line 7. Locking larger parts of the DeMR code, say encompassing lines 10-13 to synchronize accesses to both G and *W* in one critical region would be an example of coarse-grain ordering. The ordering granularity has performance implications, and the choice of granularity permits the programmer to tune the execution as desired.

Concurrency Tools. Finally, programmers use Concurrency Tools to manage the execution. They map UEs to programmatic entities, e.g., threads and tasks , which are scheduled for execution as a unit (sub-space *Context Abstraction*). Synchronization primitives, e.g., locks, barriers, etc., (sub-space *Synchronization*) may be used to implement the UE Coordination patterns. Constructs to coordinate the control-flow across threads (e.g., `signals` in Pthreads [1]), support for inter-task communication (e.g., `send` and `receive` directives in MPI [70]), etc., may also be employed.

Typically, Concurrency Tools are provided by the system; the programmer rarely needs to implement or alter them.

Summary. In general, Multithreading gives a great degree of latitude to express parallel programs. Parallel computations may operate on static and dynamic data structures. They may first inspect data to determine what data to operate on. Computations may be exposed linearly or through nesting. Their execution schedule and their relative execution order may be arbitrarily coordinated.

5.2 Parakram APIs

Parakram APIs serve the goal of matching the capabilities of multithreaded programs by enabling the parallelism patterns identified above, but without complicating programming. Parakram provides a small set of APIs to express task-based ordered programs in C++.Whereas the APIs permit the programmer to express the intent, the Parakram runtime library realizes the patterns at run-time. We will refer to various operations performed by the runtime in this section, but the details will be described in Chapter 6.

The APIs permit programmers to express: (i) arbitrary, including nested, invocation of tasks, (ii) tasks with data-independent and data-dependent datasets, which may operate concurrently on static and dynamic data structures, and (iii) fine- and coarse-grain inter-task ordering.

The APIs abstract away most of the concurrency management details, e.g., threads, locks, task scheduling, etc., from the programmers. Through the APIs, programmers identify the following basic information: (i) potentially parallel tasks, (ii) objects shared between the tasks, (iii) the task datasets, and (iv) provide deep copy constructors of the objects in the dataset. Note that the programmer already reasons about parallel tasks and the data they access when constructing the parallel algorithm. Copy constructors are often routinely coded in the regular course of developing programs. None of this information requires an analysis of the program's dynamic execution, as was our original objective with Parakram.

Parakram's execution model hinges on identifying data accessed by tasks and executing them in the dataflow fashion. Identifying tasks, their order, and the data they access are sufficient to enable this model. The copy constructors are needed to enable speculation when Parakram strives to explore parallelism that may be otherwise obscured when programmers use patterns such as nested tasks and data-dependent datasets. All patterns described above can be realized by Parakram using this information alone. However, Parakram yields additional control to the programmer to guide the execution, for performance fine-tuning, or simply for programming convenience, as will be seen.

Parakram APIs are grouped into four types, related to objects (Table 5.6), tasks (Table 5.7), datasets (Table 5.8), and miscellaneous functions (Table 5.9).

Objects. Refer to Table 5.6. Parakram requires that programmers identify global objects accessed in parallel tasks, particularly those shared between the tasks, by inheriting from a provided base class, `token_t`. This allows the Parakram runtime to track objects and manage their state during the execution. The base class defines two virtual methods, `clone()` and `restore()`, which the programmer must define for the derived class. `clone()` is similar to a deep copy constructor, and

APIs	Description and Usage
`token_t`	Parakram base class that global objects accessed in parallel tasks must inherit from.
`clone ()`	Virtual copy constructor method in the `token_t` class. Derived class must provide a definition. `token_t* clone ();`
`restore ()`	Virtual assignment operator method in the `token_t` class. Derived class must provide a definition. `void restore (token_t* object);`
`pk_obj_set_t`	Parakram STL set container of `token_t` type objects. Used to communicate a task's dataset.

Table 5.6: Parakram APIs related to program objects, their brief descriptions, and declarations in pseudocode.

`restore()` is equivalent to an assignment operator. The runtime uses them to manage the program's execution.

Tasks. Refer to Table 5.7. Parakram provides two APIs, `pk_task()`, `pk_task_uc()`, to invoke potentially parallel tasks. Both support linear and nested invocations. `pk_task()` expects the task to modify the program's state *eagerly*, i.e., at any time during the execution. `pk_task_uc()` is a performance-enhancing variation of `pk_task()` for use by tasks that use data-dependent datasets. It expects the task to modify the state *lazily*, i.e., Parakram permits the task to execute speculatively and modify local copies of the state, and updates the globally visible state once it has ensured that the speculation was not incorrect.

Datasets. Refer to Table 5.8. Parakram requires that each task *declare* its dataset, as a separate read set, a write set, and a *mod set*, by providing pointers to the objects in them. It is necessary to include only objects that are shared with other parallel tasks in the read and write sets. Whereas the write

APIs	Description and Usage

`pk_task ()`	Invokes parallel task. The task may declare its dataset eagerly through the same interface, or just-in-time, or lazily. The task updates state eagerly.
	
	`void pk_task (pk_obj_set_t* write set, pk_obj_set_t* mod set, pk_obj_set_t* read set, void* task pointer, void* task arguments);`
	`void pk_task (token_t* write object, token_t* mod object, token_t* read object, void* task pointer, void* task arguments);`
	`void pk_task (void* task pointer, void* task arguments);`

`pk_task_uc()`	Invokes parallel task. Task declares dataset lazily. Updates state on commit. `void pk_task_uc (void* task pointer, void* task arguments);`
`pk_ic ()`	Invokes function on commit; used to perform non-idempotent I/O. `void pk_ic (void* task pointer);`

Table 5.7: Parakram APIs to invoke tasks, their brief descriptions, and declarations in pseudocode. Multiple interfaces are provided for programming convenience, e.g., `pk_task()` can take individual objects or set of objects as arguments.

set includes shared objects that the task may modify, the mod set includes *all* global objects that the task may modify; thus the mod set is a superset of the write set. Objects allocated in the task need not be included in the mod set. The mod set serves a purpose similar to the undo log used in Transactional Memory [86]. (The mod set can be computed by the compiler, but in this work it is provided by the programmer.) Each set is formed using the provided `pk_obj_set_t` set container. Objects specified in the write set need not be included in the mod set (the execution manager uses their union in its internal operations).

Importantly, a task need declare only its own dataset and not of any nested children tasks. This features simplifies use of nested tasks and enables nested parallelism patterns.

The dataset declaration is *eager* if performed at the time the task is invoked, through `pk_task()`.

APIs	Description and Usage
`pk_read ()`	Declares object(s) to read (just-in-time). `void pk_read (token_t* object);`
`pk_write ()`	Declares object(s) to write (just-in-time). `void pk_write (token_t* object);`
`pk_mod ()`	Declares object(s) to modify (just-in-time). `void pk_mod (token_t* object);`

`pk_declare ()`	Declares read set, write set, and mod set (lazily).
	
	`void pk_declare (pk_obj_set_t* write set, pk_obj_set_t* mod set, pk_obj_set_t* read set);`
	`void pk_declare (token_t* write object, token_t* mod object, token_t* read object);`

`pk_declare_ia ()`	Similar to `pk_declare ()`. Declares read set, write set, and mod set (lazily).
	
	`void pk_declare_ia (pk_obj_set_t* write set, pk_obj_set_t* mod set, pk_obj_set_t* read set);`
	`void pk_declare_ia (token_t* write object, token_t* mod object, token_t* read object);`

`pk_release ()`	Releases object(s) from the dataset (eagerly).
	
	`void pk_release (pk_obj_set_t* object);`
	`void pk_release (token_t* object);`

Table 5.8: Parakram APIs related to dataset declaration, their brief descriptions, and declaration in pseudocode.

To support datasets dependent on dynamic data structures, which a task may need to first traverse, Parakram permits *just-in-time* (JIT) declaration. The task can declare objects in the read, write, and mod sets as it advances (APIs `pk_read()`, `pk_write()`, and `pk_mod()`). In just-in-time

declaration, the task declares an object before accessing it.

Parakram provides more flexible APIs, `pk_declare()` and `pk_declare_ia()`, to declare the dataset *lazily*. Lazy declaration permits the task to read objects before declaring them, but requires the task to declare the read set, the write set, and the mod set before it modifies data. The differences between the two APIs will be described in the next chapter.

A task can also *release* objects in its dataset *eagerly*, indicating that it will no longer access them (`pk_release()`). If not released eagerly, they are implicitly released when the task completes. Eager release allows other dependent tasks to proceed sooner, instead of waiting for the task to finish.

With linear and nested tasks, and eager, JIT and lazy dataset declaration, Parakram admits arbitrary algorithms. Combined with eager release, it also permits finer ordering granularity. Chapter 6 describes these aspects in detail.

Concurrency Tools. Refer to Table 5.9. Parakram precludes the need for the programmer to coordinate the execution. However, in certain cases when the programmer cannot specify a task's dataset, e.g., of a third-party function, Parakram permits the user to revert to safe, sequential execution by first quiescing the parallel execution. The `pk_serial()` API may be used for this purpose. `pk_serial()` acts an **ordered barrier**. It quiesces the parallel execution that logically precedes the `pk_serial()` call. The next `pk_task()` or `pk_task_uc()` call resumes the parallel execution.

`pk_barrier()` is another concurrency control API, but is provided for programming convenience. It helps perform coarse-grain parallelism control which might otherwise require the programmer to create large datasets. `pk_barrier()` acts as barrier for a given nested scope, analogous to the `sync` directive in Cilk [71]. It permits the execution to advance only after all children tasks invoked by the parent have completed.

No other concurrency tools are provided. There is no implicit barrier at the end of the task calls in Parakram, unlike the implicit `sync` in Cilk's `spawn`ed tasks, or the implicit join in OpenMP's `parallel for` pragma.

APIs	Description and Usage
`pk_except ()`	Reports exception. Exceptions are processed in the program order. `void pk_except (int error code);`
`pk_barrier ()`	Barrier synchronization. `void pk_barrier ();`
`pk_serial ()`	Delineates serial region of execution. `void pk_serial ();`
`pk_allocate ()`	Allocates memory for an object. `void* pk_allocate ();`
`pk_delete ()`	Deallocates memory for an allocated object. `void pk_delete (object);`

Table 5.9: Miscellaneous Parakram APIs, their brief descriptions, and usage in pseudocode.

Other APIs. `pk_ic()` (Table 5.7) is used to perform I/O when a program uses speculation. `pk_except()` (Table 5.9) allows programmers to throw exceptions for the Parakram runtime to process. Their utility is described in the next chapter.

5.3 Ordered Programs

Given the above APIs, we evaluated Parakram's expressiveness. Expressiveness is function of the APIs and Parakram's execution model. We study the APIs in this chapter and revisit expressiveness as a whole in Chapter 7 after the execution model has been described in Chapter 6.

To study the expressiveness, we developed ordered programs for a wide range of algorithms commonly used in parallelism studies, as listed in Table 5.10. We chose programs from the PARSEC [23],

Programs (1)	**Source** (2)	**Multithreaded** (3)	**Parallelism** (4)	**UE** (5)	**Dataset** (6)	**Declare** (7)	**Release** (8)
Barnes-Hut	Lonestar	Pthreads	Regular	Linear	Data-ind.	Eager	Lazy
Black-Scholes	PARSEC	Pthreads	Regular	Linear	Data-ind.	Eager	Lazy
Iterative CD	[141]	OpenMP	Irregular	Linear	Data-ind.	Eager	Lazy
CGM	[93]	OpenMP	Regular	Linear	Data-ind.	Eager	Lazy
Histogram	Phoenix	Pthreads	Map-reduce	Linear	Data-ind.	Eager	Lazy
Dedup	PARSEC	Pthreads	Irregular	Linear	Data-ind.	Eager	Lazy
Pbzip2	[74]	Pthreads	Irregular	Linear	Data-ind.	Eager	Lazy
I. Sparse LU	Barcelona	OpenMP	Irregular	Linear	Data-ind.	Eager	Lazy
Reverse Index	Phoenix	Pthreads	Map-reduce	Linear	Data-ind.	Eager	Lazy
Swaptions	PARSEC	Pthreads	Regular	Linear	Data-ind.	Eager	Lazy
Word Count	Phoenix	Pthreads	Map-reduce	Linear	Data-ind.	Eager	Lazy
I. Mergesort		Cilk	Regular	Linear	Data-ind.	Eager	Lazy
Recursive CD	[123]	Cilk	Irregular	Nested	Data-ind.	Eager	Lazy
DeMR	Lonestar	Pthreads	Irregular	Linear	Data-dep.	JIT	Lazy
Genome	STAMP	TL2 TM	Irregular	Nested	Data-dep.	JIT	Eager
Labyrinth	STAMP	TL2 TM	Irregular	Linear	Data-dep.	Lazy	Lazy
Mergesort	[123]	Cilk	Regular	Nested	Data-ind.	Eager	Lazy
BFS	Lonestar	Pthreads	Irregular	Linear	Data-dep.	Lazy	Lazy
RE	[11]	Pthreads	Regular	Linear	Data-dep.	JIT	Lazy
R. Sparse LU	Barcelona	OpenMP	Irregular	Nested	Data-ind.	Eager	Lazy
Vacation	STAMP	TL2 TM	Irregular	Linear	Data-dep.	JIT	Eager

Table 5.10: Programs and their relative characteristics. CD = Cholesky decomposition, CGM = conjugate gradient method, Data-ind. = data independent, Data-dep. = data dependent, LU = LU decomposition, I. = iterative, DeMR = Delaunay mesh refinement, JIT = just-in-time, BFS = breadth-first search, R. = recursive.

Phoenix [149], STAMP [129], Lonestar [104], and Barcelona [59] suites, scientific kernels, common utilities Pbzip2 [74] and Mergesort, and a networking program RE [11]. We chose algorithms with different characteristics: type of parallelism (column 4), static and dynamic dependences, task types (column 5), dataset types (column 6), dataset declaration (column 7), and dataset release (column 8).

To develop Parakram programs we started with the original C/C++ sequential codes and first transformed them into C++ object-oriented programs. Next, we studied the parallelization strategies used in the multithreaded variants, and applied similar strategies using Parakram APIs, e.g., recursion, to develop the ordered versions.

In this section we compare the Parakram programs with their sequential and multithreaded counterparts. An extensive user-study comparing the programming effort required for multithreaded and ordered programs, similar to one performed for Transactional Memory [157], was outside the scope of our work. Nonetheless, we briefly comment on the relative programming effort needed to develop ordered and multithreaded programs.

Column 3 in Table 5.10 lists the multithreaded variants used for comparison. We coded published parallel algorithms for iterative CD [141] and the Conjugate gradient method [93] in OpenMP. We implemented nondeterministic Galois-like designs [142] for DeMR and breadth first search (BFS) in Pthreads. RE was parallelized by us using Pthreads. Cilk codes for recursive CD and Mergesort were obtained from McCool et al. [123]. Iterative Mergesort was coded by us in Cilk. All other multithreaded parallel codes were obtained from their sources (column 2).

In all cases the Parakram code is similar to the sequential code, except for the additional code needed to formulate the read and write sets, and the use of Parakram APIs. Unlike multithreaded programs, no threads are created and no synchronization primitives are needed to facilitate the concurrent execution. No explicit work distribution or explicit ordering of the execution is required. Further, no pattern-specific structure is needed in the code to exploit the parallelism.

We highlight the key aspects of developing ordered programs using the three examples from before, Pbzip2, Cholesky decomposition, and Delaunay mesh refinement. We show (incomplete) code segments of the main kernels of the three programs to highlight the differences[2]. To ease the comparison, the sequential and Parakram codes are listed beside each other and the sequential code is appropriately formatted. The multithreaded code, which is usually longer, follows thereafter.

We first show how a generic object is defined in a Parakram program, and then present how the parallelism is expressed using Parakram in the three examples.

[2]Details not relevant to parallelism are omitted.

```
1  // User defined class; note that it inherits token_t
2  class user_object_t : token_t {
3    public:
4    ...
5      // Allocate memory for the clone and define the copy constructor
6      token_t* clone ( void ) {
7        user_object_t* obj_clone;
8        obj_clone = pk_alloc_t <user_object_t>::pk_allocate();
9        // Copy fields of this object into obj_clone's fields
10       ...
11       return obj_clone;
12     }
13     // Define the assignment operator
14     void restore ( token_t* obj_clone ) {
15       // Copy fields of obj_clone to this object's fields
16       ...
17     }
18     ...
19   private:
20     ...
21 };
```

Listing 5.1: Defining a user class in a Parakram program.

5.3.1 Defining Objects

Listing 5.1 shows an example of a user-defined class inheriting from the `token_t` base class (line 2), and defining the `clone()` (line 6) and `restore()` methods (line 14). `clone()` and `restore()` are essentially functions that make copies of an object. The former also allocates memory for the copy. These functions are needed to make "deep" copies, i.e., to ensure that the data in dynamically allocated object fields are also copied instead of only pointers. Similar functions are routinely used in regular sequential programming. `pk_alloc_t` is a Parakram provided templated class that supports the `pk_alloc()` API (line 8).

5.3.2 Pbzip2

Listing 5.2 shows the sequential code for Pbzip2 and Listing 5.3 shows the Parakram code. The Parakram code is similar to the sequential code in structure. The `compress` and `output` functions on lines 12 and 13 in Listing 5.2 are invoked using `pk_task()` (lines 12, 13 in Listing 5.3). Parakram code forms the write set for the `compress` function on lines 10 and 11. Note that the write set of `compress`, i.e., the data it computes, becomes the read set for the `output` function (line 13). Explicit mod set is not needed in Pbzip2. Pbzip2 invokes tasks linearly, the tasks declare datasets eagerly, and release the dataset lazily. Note that formulating the read and write sets required only reasoning about the task-local accesses and not about the dynamic inter-task interactions.

```
1  ...
2  while (blockBegin < fileSize - 1) {
3    OFF_T bytes_left = fileSize - blockBegin
         ;
4    OFF_T block_length = blockSize;
5    if (bytes_left < block_length)
6      block_length = bytes_left;
7
8    blockBegin += block_length;
9    block_t* block = new block_t(hInfile,
         block_length);
10
11
12   compress (block);
13   output (op_file, block);
14 }
15 ...
```

Listing 5.2: Sequential bzip2 pseudocode segment.

```
1  ...
2  while (blockBegin < fileSize - 1) {
3    OFF_T bytes_left = fileSize - blockBegin
         ;
4    OFF_T block_length = blockSize;
5    if (bytes_left < block_length)
6      block_length = bytes_left;
7
8    blockBegin += block_length;
9    block_t* block = new block_t(hInfile,
         block_length);
10   pk_obj_set_t* wrSet = new pk_obj_set_t;
11   wrSet->insert (block);
12   pk_task (wrSet, NULL, NULL, &compress);
13   pk_task (opSet, NULL, wrSet, &output);
14 }
15 ...
```

Listing 5.3: Parakram parallel bzip2 code segment.

Listing 5.4 shows the main parts of the multithreaded Pthreads implementation of Pbzip2. The Pthreads code, optimized for performance, is far more complex, as is also noted by others [73]. The `producer` function on line 1 reads from the input file, the `consumer` function on line 25 compresses the data and the `fileWriter` function on line 51 writes the compressed data to the output file. The code is complex because of several reasons. It explicitly implements pipeline parallelism, it

ensures that the writes to the output file are performed in the correct order, and it orchestrates the execution for optimum performance. The code implements the pipeline between the three functions executing on different threads, using lock-protected queues (lines 13, 28, 54). Embedded in the code on lines 16, 19, 36, and 44 is the logic to ensure that the output is written in the correct order despite the nondeterministic execution. To optimize resource utilization, the code uses wait-signalling mechanisms (lines 18, 34, 38, 58).

Note that the Pthreads design requires reasoning about the task-local data accesses as well as the dynamic inter-task interactions. By contrast, the Parakram code achieves the same goals of exploiting pipeline parallelism, performing I/O in the correct order, and optimizing performance, with only task-local reasoning and with no explicit coordination, as the Listing 5.3 shows. We shall see how these goals are realized once the execution model is presented in detail.

The Pthreads code highlights the complexity of developing optimized multithreaded designs using a generic parallel programming API.

```
1  int producer(int hInfile, int blockSize, queue *fifo) {
2    ...
3    while (1) {
4      inSize = blockSize;
5      ...
6      // read file data
7      ret = bufread(hInfile, (char *) FileData, inSize);
8      if (ret == 0) {
9        break;
10     }
11
12     // add data to the compression queue
13     pthread_mutex_lock(fifo->mut);
14     while (fifo->full) {
15       pret = pthread_cond_wait(fifo->notFull, fifo->mut);
16       queueAdd(fifo, FileData, inSize, blockNum);
17       pthread_mutex_unlock(fifo->mut);
18       pthread_cond_signal(fifo->notEmpty);
19       blockNum++;
20     }
21   }
22   ...
23 }
```

```
24
25 void *consumer (void *q) {
26   ...
27   for (;;) {
28     pthread_mutex_lock(fifo->mut);
29     while (fifo->empty) {
30       if (allDone == 1) {
31         pthread_mutex_unlock(fifo->mut);
32         return (NULL);
33       }
34       pret = pthread_cond_timedwait(fifo->notEmpty, fifo->mut, &waitTimer);
35     }
36     FileData = queueDel(fifo, &inSize, &blockNum);
37     pthread_mutex_unlock(fifo->mut);
38     pret = pthread_cond_signal(fifo->notFull);
39     outSize = (int) ((inSize*1.01)+600);
40     ret = BZ2_bzBuffToBuffCompress(CompressedData, &outSize, FileData, inSize,
           BWTblockSize, Verbosity, 30);
41
42     // store data to be written in output bin
43     pthread_mutex_lock(OutMutex);
44     OutputBuffer[blockNum].buf = CompressedData;
45     OutputBuffer[blockNum].bufSize = outSize;
46     pthread_mutex_unlock(OutMutex);
47     ...
48   }
49 }
50
51 void *fileWriter(void *outname) {
52   ...
53   while ((currBlock < NumBlocks)  (allDone == 0)) {
54     pthread_mutex_lock(OutMutex);
55     if ((OutputBuffer.size() == 0)  (OutputBuffer[currBlock].bufSize < 1)  (OutputBuffer[
           currBlock].buf == NULL)) {
56       pthread_mutex_unlock(OutMutex);
57       // sleep a little so we don't go into a tight loop using up all the CPU
58       usleep(50000);
59       continue;
60     } else
61       pthread_mutex_unlock(OutMutex);
62
63     // write data to the output file
64     ret = write(hOutfile, OutputBuffer[currBlock].buf, OutputBuffer[currBlock].bufSize);
65
66     CompressedSize += ret;
67     pthread_mutex_lock(OutMutex);
68     pthread_mutex_lock(MemMutex);
69     if (OutputBuffer[currBlock].buf != NULL) {
70       delete [] OutputBuffer[currBlock].buf;
```

```
71        NumBufferedBlocks--;
72      }
73      pthread_mutex_unlock(MemMutex);
74      pthread_mutex_unlock(OutMutex);
75      currBlock++;
76    }
77    ...
78  }
```

Listing 5.4: Pthreads parallel bzip2 code.

Modern programming models like TBB [151] and Cilk [110] do provide some respite from the complexity of multithreaded programming by directly supporting some of the parallelism patterns. For example, the programmer can instruct them to create a pipeline between identified tasks and the associated runtimes provide the necessary underlying concurrency mechanisms. Listing 5.5 shows the code snippet for Pbzip2 implemented using TBB (Cilk also provides a similar capability). It uses the in-built `pipeline` construct [123]. Line 2 informs TBB about the pipeline between the three tasks specified on lines 3, 14, and 25. This code is far simpler than Pthreads. It is similar to the sequential and the Parakram versions, but still requires the programmer to reason about the parallelism pattern in the algorithm. If the pattern does not fit a predefined template, the programmer must pursue a regular design.

```
1  ...
2  tbb::parallel_pipeline(simultaneous,
3    tbb::make_filter<void, blockBuf*>(tbb::filter::serial_in_order, [&](tbb::flow_control &
         fc)-> blockBuf* {
4      blockBuf *b = new blockBuf(BLOCKSIZE);
5      int ret = read(hInFile, b->buf, BLOCKSIZE);
6      if (ret == 0) {
7        fc.stop();
8        return NULL;
9      }
10     b->originalSize = ret;
11     return b;
12   } ) &
13
14   tbb::make_filter<blockBuf*,blockBuf*>(tbb::filter::parallel, [] (blockBuf *b)->
         blockBuf* {
15     unsigned int outSize = b->originalSize * 1.01 + 600;
```

```
16      b->bufSize = outSize;
17      char *temp = new char[outSize];
18      int ret = BZ2_bzBuffToBuffCompress(temp, &outSize, b->buf, b->originalSize, BLOCKSIZE
            / 100000, 0, 30);
19      b->bufSize = outSize;
20      delete b->buf;
21      b->buf = temp;
22      return b;
23    } ) &
24
25    tbb::make_filter<blockBuf*,void>(tbb::filter::serial_in_order, [&](blockBuf *b)->
          blockBuf* {
26      int ret = write(hOutFile, b->buf, b->bufSize);
27      delete b;
28      return NULL;
29    } )
30  );
31  ...
```

Listing 5.5: Using the pipeline template in TBB for parallel bzip2

5.3.3 Cholesky Decomposition

Listing 5.6 shows the sequential code for iterative design of Cholesky decomposition and Listing 5.7 shows the Parakram code.

```
1   matrix M[blocks];
2   ...
3   for ( i = 0; i < blocks; i++ ) {
4     for (j = i; j < blocks; j++) {
5       in = j*(j+1)/2 + i;
6       for ( k = 0; k < i; k++ ) {
7         l1 = j*(j+1)/2 + k;
8         l2 = i*(i+1)/2 + k;
9
10
11
12
13
14
15
16          mmult (M[in], M[l1], M[l2], size,
                -1);
17        }
18        if (i == j) {
19
20
21
22          cd_diag ( M[in], size );
23
24        } else {
25          l2 = i*(i+1)/2 + k;
26
27
28
29
30
31
32          cd ( M[in], M[l2], size );
33
34        }
35      }
36    }
```

Listing 5.6: Sequential Cholesky decomposition code snippet.

```
1   matrix M[blocks];
2   ...
3   for ( i = 0; i < blocks; i++ ) {
4     for (j = i; j < blocks; j++) {
5       in = j*(j+1)/2 + i;
6       for ( k = 0; k < i; k++ ) {
7         l1 = j*(j+1)/2 + k;
8         l2 = i*(i+1)/2 + k;
9         pk_obj_set_t* wrSet = new
              pk_obj_set_t;
10        pk_obj_set_t* rdSet = new
              pk_obj_set_t;
11        wrSet->insert (&M[in]);
12        rdSet->insert (&M[l1]);
13        rdSet->insert (&M[l2]);
14        pk_task (wrSet, NULL, rdSet, &
              mmult, args);
15      }
16      if (i == j) {
17        pk_obj_set_t* wrSet2 = new
              pk_obj_set_t;
18        wrSet2->insert (&M[in]);
19        pk_task (wrSet2, NULL, NULL, &
              cd_diag, args);
20      } else {
21        l2 = i*(i+1)/2 + k;
22        pk_obj_set_t* wrSet3 = new
              pk_obj_set_t;
23        pk_obj_set_t* rdSet3 = new
              pk_obj_set_t;
24        wrSet3->insert (&M[in]);
25        rdSet3->insert (&M[l2]);
26        pk_task (wrSet3, NULL, rdSet3, &cd,
               args);
27      }
28    }
29  }
```

Listing 5.7: Parakram iterative Cholesky decomposition code snippet.

```
3   for ( i = 1; i < blocks; i++ ) {
4     #pragma omp parallel for schedule ...
5     for (j = i; j < blocks; j++) {
6       in = j*(j+1)/2 + i-1;
7       l2 = (i-1)*i/2 + i-1;
8       ch (M[in], M[l2]);
9     }
10    #pragma omp parallel shared (M) num_threads ... {
11      #pragma omp single {
12        in = i*(i+1)/2 + i;
13        l1 = i*(i+1)/2 + i-1;
14        #pragma omp task shared (M, index, l1)
15        {
16          mmult_diag(M[in], M[l1], M[l1]);
17        }
18      }
19      for (j = i+1; j < blocks; j++ ) {
20        #pragma omp parallel for schedule ...
21        for (k = i; k <= j; k++ ) {
22          in = j*(j+1)/2 + k;
23          l1 = j*(j+1)/2 + i-1;
24          l2 = k*(k+1)/2 + i-1;
25          if (k == j) {
26            mmult_diag (M[in], M[l1], M[l2]);
27          } else {
28            mmult (M[in], M[l1], M[l2]);
29          }
30        }
31      }
32    }
33  }
```

Listing 5.8: OpenMP Cholesky decomposition code snippet.

Once again, the Parakram code is similar to the sequential code but for the creation of the read and write sets (lines 9-13, 17-18, 22-25) and the `pk_task()` calls (lines 14, 19, 26). Iterative CD also uses linear tasks, eager dataset declaration, and lazy dataset release. CD has complex, unstructured inter-task dependences. Parakram precludes the need to analyze and account for them in the code. Only task-local analysis is required, to formulate the read and write sets (there is no need for a separate mod set). By contrast, the multithreaded code, written in OpenMP (Listing 5.8) needs to explicitly expose and coordinate the parallelism, which results in a different code structure. It uses the fork-join parallelism, that OpenMP facilitates readily, and relies on the implicit barriers at the

join points to respect inter-task dependence. This comes at a performance penalty, as we will see in Chapter 7.

```
1  matrix M[blocks];
2  ...
3  cd_rec (int i, int j, ...) {
4    if ( blocks > 1 ) {
5      int n2 = blocks/2;
6
7
8
9      cd_rec (i, j, M, n2, size);
10
11
12
13     cd_rec (i, j+n2, M, n2, size);
14
15
16
17
18     cd_rec (i+n2, j, M, n2, size);
19
20
21
22
23     cd_rec (i+n2, j+n2, M, n2, size);
24
25   } else {
26     int in = i*(i+1)/2 + j;‘
27     int k;
28
29
30
31
32     for ( k = 0; k < j; k++ ) {
33       int l1 = i*(i+1)/2 + k;
34       int l2 = j*(j+1)/2 + k;
35
36
37
38
39        mmult (M[in], M[l1], M[l2], size,
              -1);
40     }
```

```
1  matrix M[blocks];
2  ...
3  cd_rec (int i, int j, ...) {
4    if ( blocks > 1 ) {
5      int n2 = blocks/2;
6
7      cd_args_t* arg1 = new cd_args_t;
8      arg1->i = i; arg1->j = j;
9      pk_task (NULL, NULL, NULL, &cd_rec,
           arg1);
10
11     cd_args_t* arg2 = new cd_args_t;
12     arg2->i = i; arg2->j = j+n2;
13     pk_task (NULL, NULL, NULL, &cd_rec,
           arg2);
14
15     cd_args_t* arg3 = new cd_args_t;
16     arg3->i = i+n2; arg3->j = j;
17     pk_task (NULL, NULL, NULL, &cd_rec,
           arg3);
18
19     cd_args_t* arg4 = new cd_args_t;
20     arg4->i = i+n2; arg4->j = j+n2;
21     pk_task (NULL, NULL, NULL, &cd_rec,
           arg4);
22   } else {
23     int in = i*(i+1)/2 + j;
24     int k;
25     pk_obj_set_t* wrSet = new pk_obj_set_t
           ;
26     wrSet->insert (&M[in]);
27     for ( k = 0; k < j; k++ ) {
28       int l1 = i*(i+1)/2 + k;
29       int l2 = j*(j+1)/2 + k;
30       pk_obj_set_t* rdSet = new
             pk_obj_set_t;
31       rdSet->insert (&M[l1]);
32       rdSet->insert (&M[l2]);
33       pk_task (wrSet, wrSet, rdSet, &mmult,
             args);
34     }
```

```
40      int l2 = j*(j+1)/2 + k;
41      if ( i == j ) {
42        cd_diag (&M[in], size);
43
44      } else {
45
46
47        cd (&M[in], &M[l2], size);
48
49      }
50    }
51  }
```

Listing 5.9: Sequential recursive Cholesky decomposition code.

```
36      int l2 = j*(j+1)/2 + k;
37      if ( i == j ) {
38        pk_task (wrSet, wrSet, NULL, &
              cd_diag, args);
39      } else {
40        pk_obj_set_t rdSet;
41        rdSet.insert (&M[l2]);
42        pk_task (wrSet, wrSet, rdSet, &cd,
              fn_args);
43      }
44    }
45  }
```

Listing 5.10: Parakram recursive Cholesky decomposition code.

CD can also be implemented using recursive decomposition. Listings 5.9 and 5.10 show the sequential and Parakram codes. Once again both are similar. Listing 5.11 shows a recursive design in Cilk, which is particularly well-suited for recursive algorithms. Yet the Cilk code is dramatically more complex to design and develop [123]. The complexity arises because Cilk requires that concurrently spawned tasks be independent. The algorithm implemented in the Cilk code is structured to avoid the complex inter-task dependences (implicit barriers at the end of spawned functions in Cilk help in the process). As we will see, despite this complex design, it fails to exploit the algorithm's inherent parallelism.

Note that designing the Cilk program required reasoning about the dynamic execution and devising an algorithm that fit the Cilk paradigm. By contrast the Parakram code is simply a listing of tasks annotated with read and write sets, which require only task-local reasoning. Also note that although Cilk (and likewise TBB) could simplify Pbzip2, it is unable to simplify the CD code because the dependence patterns in CD are unstructured and do not neatly follow a pattern template. By contrast, Parakram can handle arbitrary, as well as structured dependences equally well.

```
1  void parallel_potrf (int n, T a[], int lda) {
2    if (double(n)*n*n<=CUT) {
```

```
3     leaf_potf2( n, a, lda );
4   } else {
5     int n2=n/2;
6     parallel_potrf( n2, a, lda );
7     parallel_trsm( n-n2, n2, a, lda, a+n2, lda );
8     parallel_syrk( n-n2, n2, a+n2, lda, a+n2+n2*lda, lda );
9     parallel_potrf( n-n2, a+n2+n2*lda, lda );
10  }
11 }
12
13 void parallel_trsm (int m, int n, const T b[], int ldb, T a[], int lda) {
14   if (double(m)*m*n<=CUT) {
15     leaf_trsm(m, n, b, ldb, a, lda, 'R', 'T', 'N' );
16   } else {
17     if (m>=n) {
18       int m2=m/2;
19       cilk_spawn parallel_trsm( m2, n, b, ldb, a, lda );
20       parallel_trsm( m-m2, n, b, ldb, a+m2, lda );
21     } else {
22       int n2=n/2;
23       parallel_trsm( m, n2, b, ldb, a, lda );
24       parallel_gemm( m, n-n2, n2, a, lda, b+n2, ldb, a+n2*lda, lda,'N', 'T', T(-1), T(1) )
             ;
25       parallel_trsm( m, n-n2, b+n2+n2*ldb, ldb, a+n2*lda, lda );
26     }
27   }
28 }
29
30 void parallel_syrk (int n, int k, const T c[], int ldc, T a[], int lda) {
31   if (double(n)*n*k<=CUT) {
32     leaf_syrk( n, k, c, ldc, a, lda );
33   } else if( n>=k ) {
34     int n2=n/2;
35     cilk_spawn parallel_syrk( n2, k, c, ldc, a, lda );
36     cilk_spawn parallel_gemm( n-n2, n2, k,c+n2, ldc, c, ldc, a+n2, lda,'N', 'T', T(-1), T
         (1) );
37     parallel_syrk( n-n2, k, c+n2, ldc, a+n2+n2*lda, lda );
38   } else {
39     int k2=k/2;
40     parallel_syrk( n, k2, c, ldc, a, lda );
41     parallel_syrk( n, k-k2, c+k2*ldc, ldc, a, lda );
42   }
43 }
44
45 void parallel_gemm (int m, int n, int k, const T a[], int lda, const T b[], int ldb, T c
     [], int ldc, char transa='N', char transb='N', T alpha=1, T beta=1) {
46   if (double(m)*n*k<=CUT) {
47     leaf_gemm( m, n, k, a, lda, b, ldb, c, ldc, transa, transb, alpha, beta );
48   } else if( n>=m && n>=k ) {
```

```
49      int n2=n/2;
50      cilk_spawn parallel_gemm( m, n2, k, a, lda, b, ldb, c, ldc, transa, transb, alpha,
            beta );
51      parallel_gemm( m, n-n2, k, a, lda, at(transb,0,n2,b,ldb), ldb, c+n2*ldc, ldc, transa,
             transb, alpha, beta );
52    } else if( m>=k ) {
53      int m2=m/2;
54      cilk_spawn parallel_gemm( m2, n, k, a, lda, b, ldb, c, ldc, transa, transb, alpha,
            beta );
55      parallel_gemm( m-m2, n, k, at(transa,m2,0,a,lda), lda, b, ldb, c+m2, ldc, transa,
            transb, alpha, beta );
56    } else {
57      int k2=k/2;
58      parallel_gemm( m, n, k2, a, lda, b , ldb, c, ldc, transa, transb, alpha, beta );
59      parallel_gemm( m, n, k-k2, at(transa,0,k2,a,lda), lda, at(transb,k2,0,b,ldb), ldb, c,
             ldc, transa, transb, alpha );
60    }
61  }
62
63  int main () {
64    ...
65    parallel_potrf (size, &M, rows);
66    ...
67  }
```

Listing 5.11: Cilk recursive Cholesky decomposition code.

5.3.4 Delaunay Mesh Refinement

Listings 5.12 and 5.13 show the sequential and Parakram codes for DeMR. DeMR uses linear (line 25) as well as nested tasks (line 17), just-in-time dataset declaration (lines 3, 8), and lazy dataset release. DeMR also dynamically allocates and deallocates objects (lines 13, 14). Once again, the Parakram code resembles the sequential code.

For the multithreaded DeMR, we implemented a speculative design analogous to the Galois system [142]. Some aspects of the speculation are visible in the Listing 5.14 (lines 20-23). The design is more complex than Parakram because of the explicit coordination. The Pthreads DeMR example shows the complexity of coding algorithms that use data-dependent datasets, when high performance is desired. The code would be simpler if the Galois runtime and APIs were used. Galois uses TM-like speculation, and provides Atomicity and Isolation. Such algorithms can take

advantage of Galois' automated conflict detection and rollback, which can eliminate the need for the explicit coordination, simplifying the code. Galois is also particularly suitable for graph algorithms. Parakram code is be similar to the Galois code in the Lonestar suite.

Nonetheless, Parakram matches the Pthreads DeMR code in expressiveness. The `pk_write()` on lines 3 and 8 in Parakram (Listing 5.13) parallel the locking on line 5 and 13 in the Listing 5.14. The logic to enable speculation in Pthreads is not needed in Parakram.

```
1  void refine (graph, node) {
2    ...
3
4    while (.. // cavity is incomplete
5      node_t neighbor;
6      // Scan neighbors
7      neighbor = findNeighbor (node, ...);
8
9      cavity->insert (neighbor);
10     ...
11   }
12   ...
13   new_nodes = retriangulate (cavity);
14   deleteGraph (cavity);
15   addGraph (new_nodes);
16   new_bad_nodes = checkNodes (new_nodes);
17   updateWorkList (new_nad_nodes);
18
19 }
20 ...
21 int main (..) {
22   ...
23   while (!workList->empty()) {
24     ...
25     node = workList->front ()
26     refine (graph, node)
27     ...
28   }
29   ...
30 }
```

Listing 5.12: Sequential Delaunay mesh refinement code.

```
1  void refine (graph, node) {
2    ...
3    pk_write (node)
4    while (.. // cavity is incomplete
5      node_t neighbor;
6      // Scan neighbors
7      neighbor = findNeighbor (node, ...);
8      pk_write (neighbor);
9      cavity->insert (neighbor);
10     ...
11   }
12   ...
13   new_nodes = retriangulate (cavity);
14   deleteGraph (cavity);
15   addGraph (new_nodes);
16   new_bad_nodes = checkNodes (new_nodes);
17   pk_task (updateWorkList, new_bad_nodes)
         ;
18 }
19 ...
20 int main (..) {
21   ...
22   while (!workList->empty()) {
23     ...
24     node = workList->front ()
25     pk_task (&refine, graph, node)
26     ...
27   }
28   ...
29 }
```

Listing 5.13: Parakram Delaunay mesh refinement code.

```
1  void* refine (void* args) {
2    ...
3    while ( !localWorkList.empty() ) {
4      node = localWorkList[next];
5      if ( !get_lock (node) ) {
6        continue;
7      }
8      bool got_lock = true;
9      while (.. // cavity is incomplete
10       node_t neighbor;
11       // Scan neighbors
12       neighbor = findNeighbor (node, ...);
13       got_lock = get_lock (neighbor);
14       if ( !got_lock ) {
15         break;
16       }
17       cavity->insert (neighbor);
18       ...
19     }
20     if ( !got_lock ) {
21       release_locks ();
22       continue;
23     }
24     ...
25     new_nodes = retriangulate (cavity);
26     deleteGraph (cavity);
27     addGraph (new_nodes);
28     new_bad_nodes = checkNodes (new_nodes);
29     updateLocalWorkList (localWorkList, new_nad_nodes)
30     localWorkList.remove ();
31   }
32 }
33 ...
34 int main (..) {
35   ...
36   workList_t workList[num_threads];
37
38   for (int i = 0; i < num_threads; i++) {
39     // Populate workList for each thread
40     createWorkList (workList[i], Graph);
41   }
42   for (int i = 0; i < num_threads; i++) {
43     pthread_create (&refine, args);
44   }
45   for (int i = 0; i < num_threads; i++) {
46     pthread_join (...);
47   }
48   ...
```

```
49 }
```

Listing 5.14: Pthreads Delaunay mesh refinement code.

5.4 Multithreaded Programming

We make a brief note on our experience with multithreaded programs. We studied and used several APIs to develop them. Many more APIs are available. We found that Multithreading APIs do not provide uniform capabilities, nor is any one API suitable for a broad range of algorithms. For example, a TBB design for pipeline parallelism is simpler than the Pthreads implementation, but the DeMR design in TBB will be no simpler than the Pthreads design. Similarly, DeMR implemented in Galois [142] or TM [129] can be simpler than Pthreads (Listing 5.14), but Galois or TM cannot simplify the pipeline parallelism in Pbzip2 to the same extent as TBB. Neither can they tackle CD the way Parakram does. In our experience, the ordered interface of Parakram proved to be the easiest to express the range of programs in Table 5.10. No one Multithreading API proved easy to express all of them.

5.5 Summary

Multithreading abstractions permit programmers to implement a wide range of concurrency design patterns. It is possible for an ordered programming abstraction to provide features that admit analogous design patterns. Order need not constrain the expression of parallelism. We presented Parakram's programming APIs and a few program examples authored using them. Programming in Parakram requires programmers to reason about only task-local operations, and precludes the need for explicit coordination and orchestration of the execution. We found that all Parakram programs that we developed resembled the sequential counterparts and were simpler to code than the multithreaded counterparts.

6 Executing Ordered Programs

It turns out that an eerie type of chaos can lurk just behind a facade of order - and yet, deep inside the chaos lurks an even eerier type of order.

— Douglas Hofstadter

In Chapter 4 we presented an overview of the Parakram ordered approach. Recall that we posed the fundamental question, does the artificial constraint of order in a multiprocessor program, where one is presumably not needed, hamper the program's expressiveness and performance? Of the two-part answer to this question, in the last chapter we presented the first, Parakram's programming model. In this chapter we present the second part, Parakram's execution model.

Parakram's execution model serves two main goals. The first is to combine with the programming model and realize the encoded parallelism patterns at run-time. The second is to overcome the ordering constraint and exploit the program's inherent parallelism. Parakram achieves both objectives by primarily relying on dataflow execution. But exploiting parallelism requires breaking the strict total order of the program's computations. Nonetheless, Parakram provides an appearance of order, on demand, by managing the program's execution and its state. Furthermore, to maximize the parallelism opportunities when the dataflow execution may not be immediately possible, for reasons to be seen, Parakram may execute computations speculatively. Speculative execution may violate the order and dependence between computations. Yet, Parakram ensures that the program executes correctly and gives the appearance that it executed in order. Order may appear to constrain parallelism, and as we shall see in some cases, it can. But counter to intuition, order can also assist in exploiting parallelism.

To aid the exposition, we present the execution model's details in two parts: the conceptual principles it employs and the implementation of its prototype. Section 6.1 provides definitions of some terms that are relevant to the discussion. Section 6.2 presents the overall architecture of the model and Section 6.3 describes its operating principles. In Section 6.4 we present the design details of the prototype and its mechanisms. In Chapter 7 we experimentally evaluate the Parakram design for expressiveness and performance.

6.1 Terms

Definition of terms commonly understood in the literature, but used by us in the context of ordered programs, are provided here for reference.

Definition 6.1. *Given a program and its ordered computations, an **older** computation precedes (Definition 2.7) a **younger** computation in the order.*

Definition 6.2. *A **conflicting** access to an object is said to be performed if two or more computations simultaneously access the object and at least one computation writes to it.*

Definition 6.3. *Given a program of ordered computations, if two computations access an object and at least one writes to it, then the younger computation is said to be **data-dependent** on the older, **precedent** computation.*

Whenever it is evident from the context, we will shorten "data-dependent" to "dependent".

Definition 6.4. *A computation is said to be **control-dependent** on the computation it is invoked from.*

When a function calls another, the child is control-dependent on the parent.

6.2 System Architecture

Establishing the dataflow between a program's tasks and scheduling them for execution in dataflow order is central to Parakram's execution model.

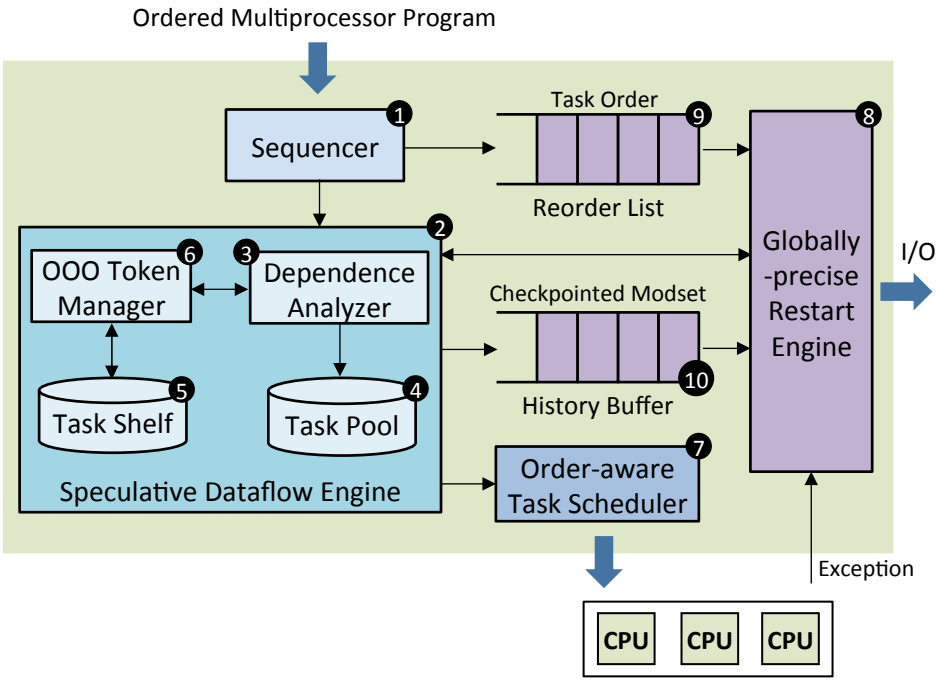


Figure 6.5: Architecture of Parakram's Execution Manager. The Sequencer interfaces with the program. The Speculative Dataflow Engine performs dataflow execution, and speculates to maximize parallelism. The Globally-precise Restart Engine provides an appearance of order.

Dataflow execution has proven highly successful when applied to instructions in modern microprocessors [12, 140, 173]. Dataflow scheduling permits instructions to execute as soon as their input operands are available [14]. This enables independent instructions to execute concurrently, breaking the ordering constraint where it is not needed. Furthermore, dependent instructions are automatically serialized, preserving the order where it is needed for correctness. Consequently, dataflow execution naturally exposes the program's innate parallelism, while ensuring deterministic execution, and remains an idealized standard for exploiting concurrency. We apply these principles to multiprocessor programs. In the process, we address some of the inherent and new challenges dataflow execution poses: applying the principles to tasks, managing inter-task dependences, managing resources, and applying the principles at the scale of multiprocessors.

Figure 6.5 shows the block diagram of Parakram's Execution Manager, which implements the

execution model. The Manager interposes between the program and the underlying system. It effects and manages the program's parallel execution, exploiting parallelism while providing an ordered view. The Execution Manager comprises four main components: (i) a *Sequencer* ❶, (ii) a *Speculative Dataflow Engine* ❷, (iii) an *Order-aware Task Scheduler* ❼, and (iv) a *Globally-precise Restart Engine* ❽.

The Sequencer ❶ intercepts the program to discover dynamic instances of potentially parallel tasks. It also identifies the dynamic identities of the objects in each task's user-provided dataset. The Speculative Dataflow Engine (SDX) ❷ uses the task order and the task datasets to dynamically construct a precise task dataflow graph. From the dataflow graph, independent tasks, i.e., tasks that do not need results from any other task, are considered to be *ready* for execution. Tasks dependent on results from other task(s) must wait until the results are available, i.e., until their dependences have *resolved*. Ready tasks are *submitted* to the *Task Pool* ❹. Dependent tasks are *shelved*, i.e., suspended from further execution and set aside on the *Task Shelf* ❺, freeing the resources to execute the program past them. Thus the execution can create a large task window to exploit the dynamic parallelism from. A shelved task is submitted to the Task Pool when all of its dependences have resolved.

The SDX exposes the Task Pool to the Order-aware Task Scheduler (OTS) ❼. The OTS monitors the system resources and the task order. It *schedules* tasks for execution, prioritizing older tasks over younger tasks, when resources are available, and balances the load within the system to use resources efficiently.

Dataflow task scheduling may execute tasks in parallel and out of program order. The Globally-precise Restart Engine (GRX) ❽ makes the program appear to execute in a strict order at any instance. It implements **globally-precise interrupts**, using which the execution may be paused at any point in the program. Once paused, the GRX can construct the precise architectural state of the program as if it had executed up to a given task, or even an instruction. A *Reorder List* ❾ and a *History Buffer* ❿, analogous to the Reorder Buffer in microprocessors [171], are used for this purpose.

The Reorder List (ROL) ❾ tracks in-flight tasks and their relative order. The History Buffer (HB) ❿ is used to checkpoint a task's user-provided mod set before it modifies objects. The order

from the ROL and the state from the HB are used to construct the precise architectural state when needed.

On occasion the SDX cannot immediately analyze the task dataflow, e.g., when tasks are nested or use data-dependent datasets, as will be seen. One approach to tackle such cases is to wait until the necessary information becomes available and the task dataflow can be established. But resources may idle in the meantime. Hence, to maximize parallelism, in such cases SDX speculates that the tasks are independent, and are ready for execution. We call this scheme **dependence speculation**. Subsequently, if misspeculation is detected, it is treated as an exception. The GRX rectifies the execution using the ROL and the HB, analogous to recovery of misspeculation in microprocessors.

Order also helps Parakram to gracefully handle I/O despite the out-of-order and speculative execution.

6.3 Execution Manager Operations

The Execution Manager works in coordination with the programming APIs. It employs a range of mechanisms to support the APIs. Its operations serve three goals: discover parallel tasks, schedule them for execution, and maintain order. We describe the operations in two steps. The first addresses the operations needed for the basic APIs, which we call the **basic dataflow** case. In the second step we address the remaining, more complex APIs.

6.3.1 Basic Dataflow Execution

Assume for now that the program uses linear tasks and data-independent datasets which are declared eagerly. The dataset may be released lazily or eagerly. Linear tasks that declare datasets eagerly form the basic dataflow case for Parakram, and are preferred since they permit Parakram to exploit the parallelism with least overheads, as will become evident from the following exposition.

Consider the example ordered code in Figure 6.6a. It invokes the function `F`, as a task through `pk_task()`, from a loop and designates it for potentially parallel execution. Figure 6.6b shows the

```
1  for (i=1; i < 7; i++) {
2    pk_task (wrSet, modSet, rdSet, &F);
3  }
4  pk_serial ();
5  foo (…);
```

(a)

		wrSet	rdSet
1	F1:	{B, C}	{A}
2	F2:	{D}	{A}
3	F3:	{A, E}	{I}
4	F4:	{B}	{D}
5	F5:	{B}	{D}
6	F6:	{G}	{H}

(b)

Figure 6.6: Example ordered program and dynamic task invocations. (a) Function (task) `F` is invoked in a loop. `foo()` is a "poorly" formed function whose dataset is unknown. (b) Dynamic instances of task `F`, and the dynamic instances of objects in their write and read sets.

dynamic instances of `F` as they would be invoked in a sequential execution. The objects in the write and read sets of each dynamic instance of the function are also listed (ignore the mod set for now).

At run-time, the Sequencer sequences through the program, a task at a time. When `pk_task()` is encountered, it first ascertains the invoked task's operands to build the dataflow graph. Often objects in the dataset are unknown statically. Therefore, Parakram enumerates the objects in the dataset dynamically, by dereferencing the pointers provided in the read and write sets.

The *Dependence Analyzer* ❸ uses the object identities to establish the data dependence between tasks. This is in contrast to multithreaded programs, which establish independence between tasks. In particular, the Dependence Analyzer determines whether the task currently being processed is dependent on any prior task that is still executing. If not, the task is submitted to the Task Pool. If the task is found to be dependent, it is shelved until its dependences are resolved. In either case,

the Sequencer then proceeds to process the next task in the program.

Handling Dependences.

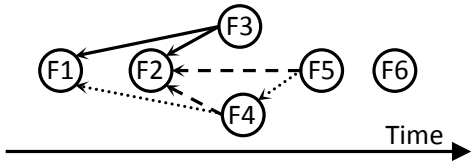
Classical dataflow machines handle dependences using tokens to signal production and availability of data [14]. We employ a similar technique, but make two enhancements. First we associate tokens with objects instead of individual memory locations. Second, we assign each object multiple read tokens and a single write token, to manage both production and consumption of data.

The *Token Manager* ❻ employs a *Token Protocol* to manage the data dependences. When the Dependence Analyzer encounters a task to be considered for dataflow execution, it requests read tokens for objects in the task's read set and write tokens for the objects in the write set objects. A task is ready for execution only after it has acquired all the requested tokens. Upon completion or when the task explicitly releases the objects (through `pk_release()`), the tokens are relinquished. Relinquished tokens are then passed to the shelved task(s) that may be waiting to access the associated objects. When a shelved task has acquired all of its requisite tokens, it can be unshelved and submitted to the Task Pool.

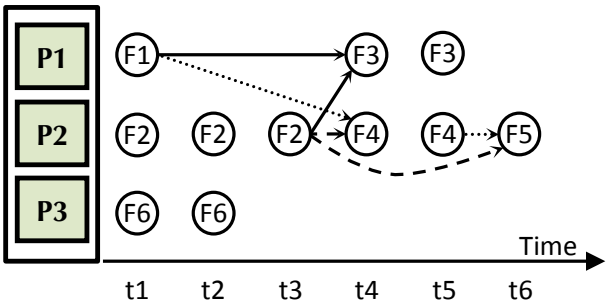
Imperative programs can impose additional hurdles. A "poorly" composed function (e.g., one which modifies global variables not communicated to it) or an opaque function (e.g., from a third party) may make it difficult to ascertain the function's read and write sets. In such a case the programmer may instruct Parakram to resort to sequential execution of the function, i.e., all prior computations in the program order complete before performing the function, and any subsequent computation begins only after its completion. This precludes the need to determine precise data dependences for the function, and hence its dataset.

Executing Tasks.

We illustrate the execution model using the program in Figure 6.6a. Figure 6.7a shows the data dependence between the dynamic task instances shown in Figure 6.6b. F3 writes objects to A and E, and thus it has a WAR dependence (solid arrows in the figure) on F1 and F2, which read from A.



(a)



(b)

Epoch	Reorder List	Completed	Retired
t1	F1 F2 F3 F4 F5 F6		
t2	F2 F3 F4 F5 F6	F1	F1
t3	F2 F3 F4 F5 F6	F1, F6	
t4	F3 F4 F5 F6	F1, F6, F2	F2
t5	F3 F4 F5 F6	F1, F6, F2, F4	
t6	F4 F5 F6	F1, F6, F2, F4, F3	F3
t6	F5 F6	F1, F6, F2, F4, F3	F4

(c)

Figure 6.7: Execution of the example ordered program. (a) Data dependence graph of the dynamic instances of task F in example 6.6b. (b) A possible execution schedule of the tasks on three processors. (c) Operations of the Reorder List. Shaded tasks in the List have completed but not yet retired.

F4 writes to object B, which F1 also writes to, hence F4 has WAW dependence (dotted arrows) on F1. F4 also reads from object D and hence has a RAW dependence (dashed arrows) on F2, which writes to D. Likewise F5 has WAW dependence on F4 (both write to object B), and has a RAW dependence on F2 (F5 reads from D which F2 writes to). Note that F1, F2, and F6 are independent. Any dependences must be preserved for correctness and to maintain the ordered appearance of the program.

Figure 6.7b simulates Parakram's execution of the code on a three-processor system. The execution is shown in time epochs. As the program begins execution in epoch t1, the Execution Manager first encounters invocation F1 and tries to acquire the necessary object tokens. Attempts to acquire a read token to object A, and write tokens to B and C are successful. Hence F1 is scheduled for execution on an available processor (P1). The execution proceeds (on another processor, not shown) and processes F2 while F1 is executing. Attempts to acquire a write token to D and a read token to A are successful, and thus F2 is scheduled to execute on processor P2. The execution advances to F3, which is shelved because a write token to object A cannot be acquired since F1 and F2, which are still executing, are reading from A. However, before being shelved, F3 acquires the read token to I and the write token to E. Next, tasks F4, F5, and F6 are processed in turn. F4 and F5 are shelved because they require the write token to B, which is being held by F1. On the other hand, F6 is able to acquire its requisite tokens (to G and H) and thus can be executed, possibly even in parallel (on processor P3) with F1 and F2.

As the execution continues, when F1 completes its execution in epoch t2, it releases the write tokens to B and C (signaling the availability of B and C). The token to B is passed to the shelved task F4, since it is the oldest task waiting for it; F5 continues to wait until F4 finishes and releases the token to B. The token for C is returned to the object since no task is waiting for it. At this point, none of the shelved tasks (F3, F4, or F5) are ready since none of them has successfully acquired all its tokens. F6 completes execution in t2.

Next, when F2 completes in t3, it releases the tokens to D and A (signaling the availability of D and completion of the read of A), thus waking up F3 (awaiting write token to A) and F4 (awaiting

read token to D). These tasks can be executed starting in epoch t4 (e.g., on processors P1 and P2), with F3 now being able to write to A in the same order as in the sequential program (after F2 is done reading it). When F4 completes, it passes the write token for B to F5, which can then be executed in epoch t6.

As tasks complete and resources become available, the Sequencer "fetches" new tasks and updates the dataflow graph by removing the completed tasks and adding the new tasks. The dataflow execution proceeds as above.

Note that dependences, even between dynamic instances of the same task, may or may not manifest at run-time, e.g., dynamic instances of the task F, F1 and F2 are independent whereas F2 and F3 are not. Handling the dependences statically, as is required in Multithreading approaches, often leads to overly conservative solutions. Thus by detecting and only serializing the dependent tasks dynamically, while permitting independent tasks to proceed in parallel, Parakram achieves the ideal dataflow schedule of execution (resources permitting).

In the event the programmer cannot formulate a function's dataset, e.g., for `foo` in the example code (Figure 6.6a, line 5), the programmer can guide Parakram to execute such functions serially, as the example does by using `pk_serial()` on line 4. `pk_serial()` ensures that `foo` executes only after the preceding parallel execution has quiesced. A subsequent `pk_task()` will prompt the Execution Manager to revert to parallel execution.

The dataflow execution as described above is naturally deterministic, i.e., any program variable receives the same sequence of values in any execution for a given input. Furthermore, because no two concurrently executing tasks can perform conflicting accesses, the execution is naturally race-free, unlike in multithreaded programs in which it is up to the programmer to ensure data-race freedom.

6.3.2 Globally-precise Interrupts

In the above example, although the program was statically ordered, the dynamic architectural state during the execution at any given instance may not be ordered due to the out-of-order execution.

This can lead to an imprecise architectural state when exceptions occur or when the programmer desires to inspect the state. For example, in Figure 6.7(b) F1, F2 and F6 execute in epoch t1. F1 completes in t1, F6 in t2 and F2 in t3. F4 can begin only in t4, once F1 and F2 complete. Thus, F6 completes out of the program order, and if the programmer were to inspect the architectural state or an exception were to occur in epoch t3, the state will be incongruous with ordered execution since F6 will have completed while F3, F4 and F5 have not even started.

Note that the program's execution need not be strictly ordered, merely an appearance is sufficient when desired. This model has been highly successful in the realm of sequential programs executing on modern superscalar processors. Despite executing instructions out of program order, they provide an ordered view, but only when desired, using precise interrupts.

Analogous to precise interrupts in microprocessors, we introduce the notion of **globally-precise interrupts** in the context of multiprocessor programs. Globally-precise interrupts (GPI) help Parakram manage the program's execution and state. In the case of the sequential program, its state is contained within the microprocessor (and the memory hierarchy) it is executing on, making it relatively easy to provide precise interrupts. The state of the multiprocessor program is dispersed among multiple processors. Hence, although a multiprocessor program's individual task running on a processor can be interrupted precisely (provided the processor supports precise interrupts), but not the entire multiprocessor program. The Globally-precise Restart Engine (GRX) extends precise interrupts globally across the multiprocessor program (and hence globally-precise interrupts). GRX permits users to pause an ordered program's execution and ensures that the collective system state reflects the program's ordered execution precisely up to the point whereby all computations preceding (Definition 2.7) it have completed and none succeeding (Definition 2.8) have started.

To implement GPI, GRX exploits the dataflow, race-free execution, leveraging the resulting properties. It achieves GPI in two steps: first, at the task boundary, and then within the task, at the instruction boundary. Since a task executes as a unit on a processor, it relies on precise-interruptible processors to provide GPI within a task. It is then left to achieve GPI at task boundaries.

The Execution Manager advances the program's execution by creating a sliding window of

consecutive tasks. GRX introduces the notion of *retiring* tasks. It manages the execution of the in-flight tasks until they retire. At run-time, as tasks are invoked, they are logged at the Reorder List's (ROL) tail, in the program order. The GRX retires the task when it reaches the ROL's head, i.e., becomes the oldest task.

Figure 6.7(c) shows the ROL's use when the example program executes as per the schedule in Figure 6.7(b). At t1, instances F1 to F6 have been processed and hence they are logged in the ROL in the program order. In t2, F1's is retired since it has completed, and its entry is removed from the ROL. At t3, F6 completes, but its entry is held in the ROL, and removed only after t6, once all preceding tasks have retired. Similarly, F4's entry is held in the ROL until F3 retires in t6. Note that since F3 and F4 complete in t5, their entries are retired in t6, one at a time.

Parakram's notion of retiring a computation is only logical. Unlike microprocessors, it permits computations to complete and update the architectural state before arriving at the ROL-head. To manage the program's state, it uses the History Buffer (HB).

Recall that the granularity of a datum is an object, and each task identifies its mod set. Once a task is scheduled, but before it starts, Parakram *clones*, i.e., checkpoints, the union of objects in its write and mod sets. The dynamic identities of the mod set objects are also established by dereferencing pointers (objects allocated in the task need not be included in the mod set). Since the dataflow execution is race-free, a clone can be created without interference from other concurrent tasks. Further, the Execution Manager clones the PC and the registers of the context just prior to invoking the task. These clones are logged in the HB for each task. When a task is retired from the ROL, the clones are discarded (and their memory is recycled). The task is no longer tracked.

The clones may be created using copy-on-write or other software-hardware schemes. Alternatively, we can use a software-hardware future file to hold the modified state until it can be committed to the memory when tasks retire (we do not explore these different cloning options in this dissertation).

Analogous to the microprocessor, when a program is to be paused for any reason such as an exception in a task, say, e.g., F4 excepts in epoch t4 (Figure 6.7(c)), the task is halted and the status

is recorded in its ROL entry. Other tasks (F3 in this case) continue to execute (epoch t5). After F3 completes and is retired, F4 reaches the ROL-head and attempts to retire in epoch t6. An exception status in the retiring task causes the program's execution to pause. The architectural state modified by the younger tasks (F6 in this case; F5 has not yet executed) is restored using the clones, in the reverse ROL order, effectively squashing their execution. The global architectural state now reflects the program's ordered execution up to the excepted instruction, i.e., F1, F2 and F3 as completed, F4 as executed up to the excepted instruction, and F5 and F6 as not started. The PC and registers saved in the precise-interruptible processor are sufficient to resume the execution from the excepting instruction. Thus we achieve globally-precise interrupts and ordered execution.

On occasion the processor may not be able to provide precise interrupts. For example, an exception caused by a hardware fault may not be precisely traceable to a specific instruction within a task. In such cases Parakram cannot be globally-precise interruptible (neither can the microprocessor be precise-interruptible). However, Parakram can provide coarse-grain preciseness. It can provide task-boundary GPI, by squashing the task itself (along with the younger tasks). It can roll back even deeper by squashing more tasks. The execution may resume from the oldest squashed task using its cloned PC and registers.

Order ensures that every dynamic task instance occupies a unique position in the the total program order. Using this fact we can apply GPI to achieve ordered execution. Achieving an analog in multithreaded programs would be difficult due to the lack of order.

6.3.3 Handling More than Dataflow Tasks

Above we have described only the execution of parallel tasks invoked from the program, but the program is usually composed from more than just parallel tasks. For example, the code in Figure 6.6a invokes task F, but from a loop which is not a part of any parallel task. Of course, the loop operations and any such non-task operations also need to be performed.

Parakram views the program as consisting of a *main program* from which parallel tasks are invoked. Consider the example program in Figure 6.8a consisting of the same loop as in Figure 6.6a.

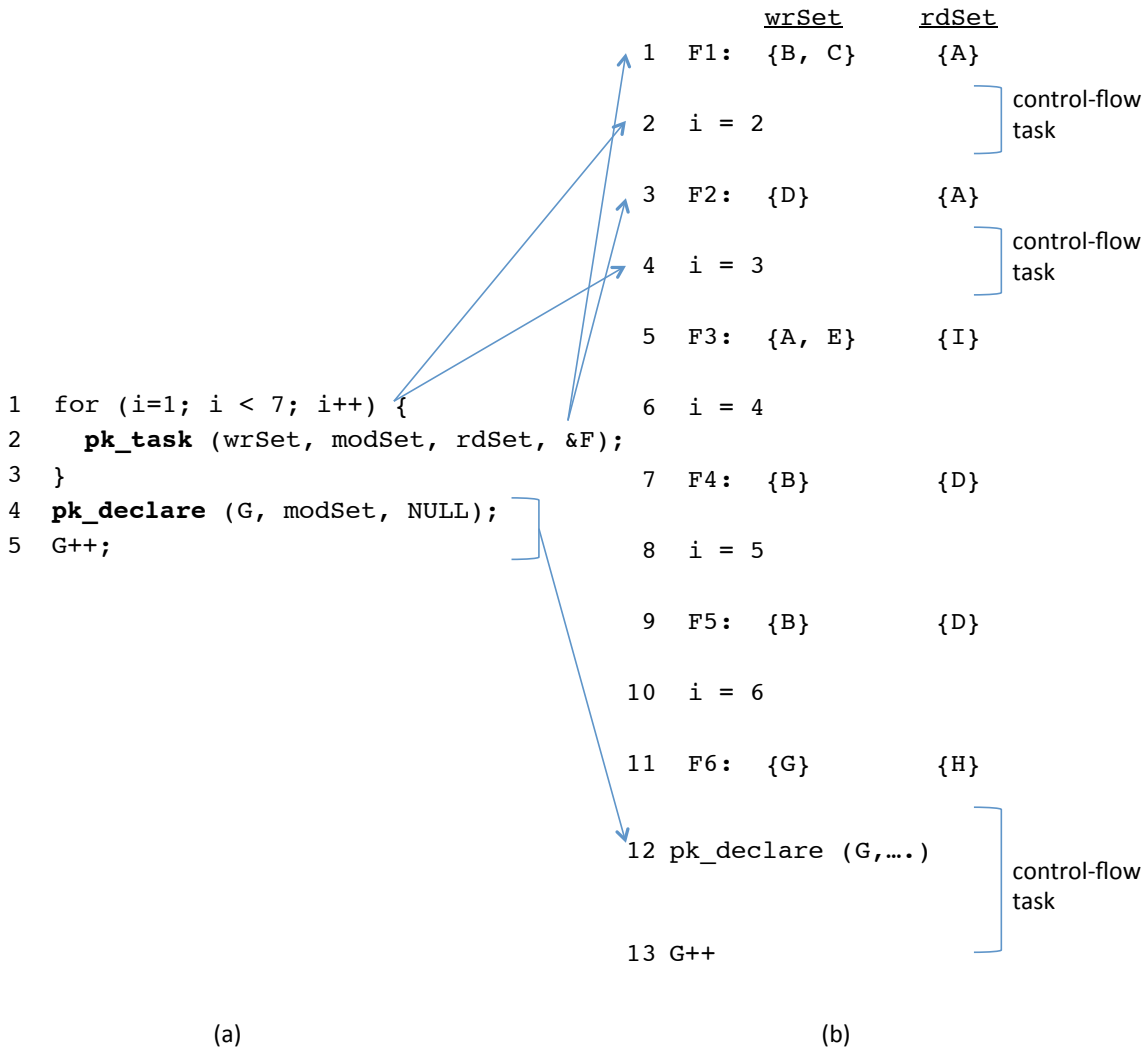


Figure 6.8: Ordered program and its dynamic logical decomposition. (a) Example program code. Function F is invoked from a loop, same as the one in Figure 6.6. (b) Dynamic execution in which tasks and intervening program segments, i.e., control-flow tasks, interleave.

The loop is part of the main program and task F is invoked from it. Figure 6.8b shows a more complete dynamic execution of the program. F1-F6 are the dynamic invocations of the parallel task F, as before. Call these, *dataflow tasks*. The dynamic execution also shows the operations performed on the induction variable `i` of the loop (other aspects of the loop operation are not shown). The execution past the last invocation, F6, is also shown (lines 12, 13). Tasks in turn may invoke nested tasks (but do not in this case).

Parakram logically treats the program segment between two successive dynamic task invocations as a *control-flow task*. Thus it views the program to be composed of interleaved dataflow tasks and control-flow tasks (CF tasks). F1, F2,..F6 are the dynamic invocations of F and the intervening segments are CF tasks, as marked.

When a dataflow task is invoked, Parakram attempts to execute the task in parallel with other dataflow tasks already invoked, but not yet completed, and the *program continuation*. The program continuation at a given point in the program is the remaining program past that point. For example, during the course of the execution, when the dynamic instance F3 on line 5 is invoked, it may execute concurrently with preceding instances F1 and F2 and the program continuation beginning from line 3 in Figure 6.8a, which corresponds to the dynamic sequence beginning from line 6 in Figure 6.8b.

Whereas the tasks may execute concurrently with respect to each other and the program continuation, CF tasks execute in the sequential order with respect to each other. Invocation F3 on line 5 may execute concurrently with the preceding and succeeding instances, F1, F2, F3-F6 (dependences permitting), but CF tasks on lines 2, 4, 6, etc., are assured to execute in that order.

Once a parallel task is processed, and whether it is submitted for execution or shelved, the control must flow through the following program segment (and hence called a control-flow task) to reach the next dataflow task invocation. This process repeats throughout the program.

A CF task may use results produced by preceding task(s). The programmer may specify the CF task's dataset, using `pk_declare()`. In our example, `pk_declare()` on line 4 states that the following CF task will write to the object G and hence declares G as a part of the write set (the read

set is empty). If no `pk_declare()` is encountered, Parakram assumes that the CF task's dataset is empty. Parakram treats CF tasks the same as tasks for dataflow and globally-precise interruptible execution, as described earlier. Entries for CF tasks are also logged in the ROL along with dataflow tasks, in the program order, and retired when they complete and become the oldest. When a `pk_declare()` is encountered in a CF task, tokens are processed and objects are cloned as described earlier. If a control-flow task cannot acquire a token, the entire program continuation gets suspended until the associated object is released.

6.3.4 Speculative Execution

So far we have described execution of programs that invoke linear tasks with eager datasets. We now turn our attention to nested tasks and lazy datasets, the more complex of the cases.

Parakram encourages use of linear tasks and eager datasets whenever possible. They help analyze dependences *before* executing the task, needed to facilitate the high performance dataflow execution. The order and the task datasets are used to construct the dataflow graph. Order of the dynamic instances of linear tasks is merely the same as their invocation order, as seen in examples Figures 6.6 and 6.8. Eager datasets allow the Execution Manager to establish the object identities before the task is executed. The immediate availability of the object identities and the task order permits the Execution Manager to construct the precise dataflow graph. However, nested tasks and lazy datasets pose challenges to this scheme.

Nested Tasks.

Unlike linear tasks, the invocation order of the dynamic instances of nested tasks may not be the same as the program order, preventing the Execution Manager from constructing the dataflow graph. Consider the pseudocode in Figure 6.9a. All tasks except A are nested. Figure 6.9b shows the sequential invocation order of the tasks (which is not the same as the program's execution order, as we shall see). Consider a possible parallel execution schedule of the tasks in Figure 6.9c along a time-line in epochs. Tasks A, B, and D are invoked as per the program order, but not C. Hence we

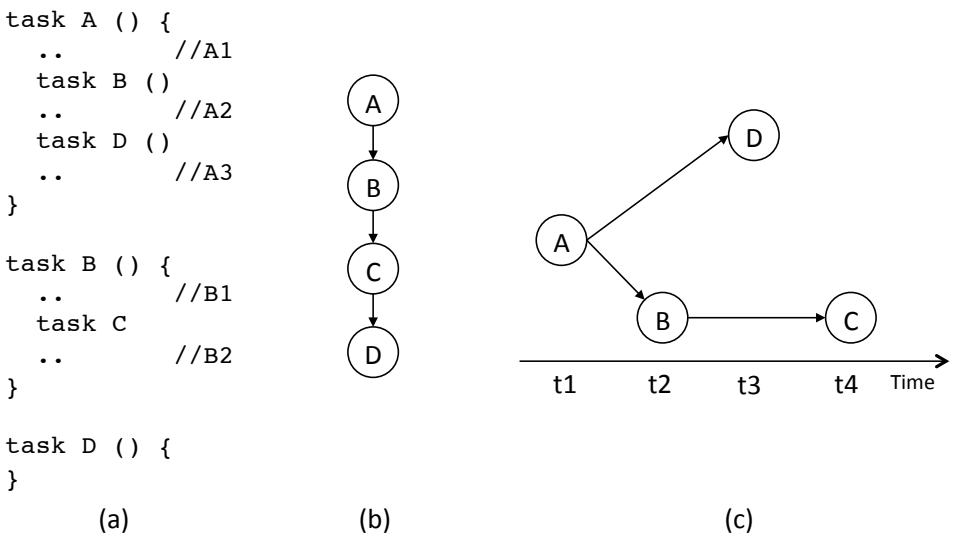


Figure 6.9: Parallel execution of nested tasks. (a) Example pseudocode. (b) Sequential order of task invocation. (c) Dynamic order.

cannot rely on the invocation order to determine their program order. If the order is unknown, the precise dataflow graph cannot be constructed.

Lazy Datasets.

The dataset can be declared eagerly if the dataset is data-independent. However, data-dependent datasets require that the task *first* execute to compute the dataset, which poses two challenges.

The first challenge arises when an in-flight task modifies data that affects another task's dataset. For example, in Delaunay mesh refinement (DeMR), refining a triangle may change the graph, which may affect another task's dataset (Algorithm 5.5, if lines 8 and 10 in different tasks execute concurrently). Further, computing the dataset forms bulk of a DeMR task's work. Hence requiring a task to wait could serialize the entire execution. One approach to handle such cases is to restructure the algorithm, e.g., by dividing the work into two. The first part would compute the dataset, without updating the data, and the second would perform the update. But the programmer may have to manually detect conflicts and coordinate the execution, complicating programming. Although this approach may be automated [89, 135], it can fail to maximize the parallelism.

The second challenge, arising from dynamic data structures, compounds the first. Algorithms that inspect and operate on dynamically changing data complicate concurrent execution. Again, in DeMR, as before, a task may insert nodes in the graph which can influence nodes traversed by another task as it computes the dataset, interfering with its operations (Algorithm 5.5, line 12). Moreover, traversing the graph while it is being modified can be unsafe. The conservative two-step approach, as above, may be employed here too, but is undesirable for similar reasons.

Most Deterministic proposals based on ordered programs do not support these two patterns, and either complicate programming or give up performance.

Recall that Parakram supports data-dependent datasets with simple, intuitive interfaces. It permits *just-in-time* (JIT) declaration via `pk_read()`, `pk_write()`, and `pk_mod()` APIs. A task can declare objects in the read, write, and mod sets as it advances, but before the objects are accessed the first time, as DeMR does in Listing 5.13, line 3. Parakram also provides a more flexible API, `pk_declare()`, to declare the dataset *lazily*. This permits the task to read objects before declaring them, but requires the task to declare the dataset before it modifies data.

JIT and lazy declaration enhance the expressiveness, but the task's dataset is unavailable before the task executes, and hence the precise dataflow graph cannot be built when the task is invoked.

In summary, a nested task may not be invoked in the correct program order, and JIT and lazy datasets are known only *after* the task starts executing. When such a *non-eager* task is encountered, the dataflow graph cannot be completely built, hindering the dataflow execution.

Handling Nested Tasks and Non-eager Datasets.

One option to handle non-eager tasks is to serialize the execution. Alternatively, they may be handled conservatively by waiting until the necessary information is known, still losing parallelism opportunities in the interim. However, Parakram resorts to **dependence speculation** to maximize the parallelism. It speculatively explores and exploits the parallelism when dependences are unknown, and still provides the ordered execution semantics.

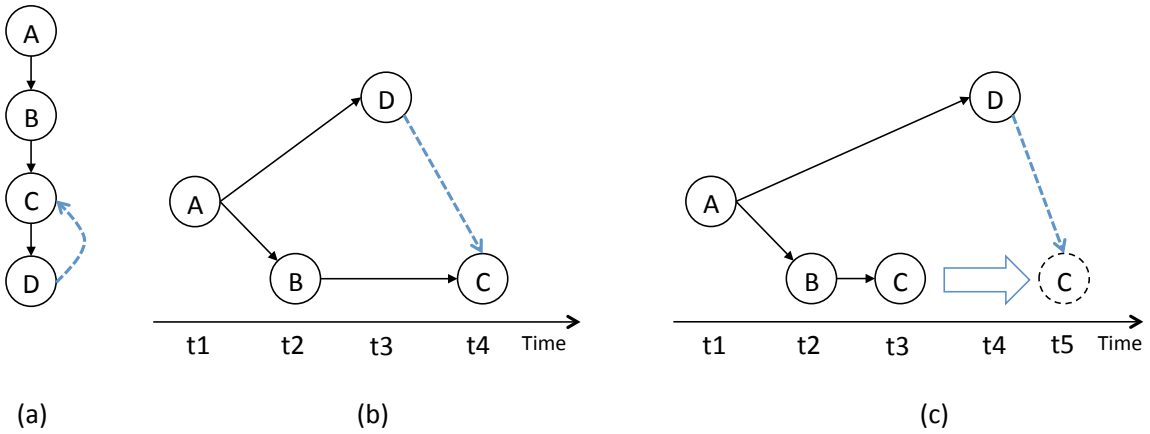


Figure 6.10: Misspeculation due to non-eager tasks (a) Sequential task invocation order. Task D has a RAW dependence on C. (b) Misspeculation from out-of-order invocation of nested tasks. (c) Misspeculation due to non-eager dataset declaration.

When a non-eager task is encountered, the Dependence Analyzer builds an incomplete dataflow graph. In the example of Figure 6.9, when task D is invoked in t3, Parakram builds the graph assuming only tasks A, B, and D, without being aware of C. If the task has no dependence on the current in-flight tasks, Parakram speculates that the task is indeed independent, and submits it to the Task Pool (from where it is scheduled for execution, resources permitting). In this case, if D is found to be independent of A and B, it will be submitted.

Eventually when the task's order and/or the dataset become known, the Dependence Analyzer completes the dependence graph, and checks whether it had misspeculated. In the example, once C is invoked in t4, the Execution Manager will complete the dataflow graph using C's dataset. Based on this new graph, if D is found to be independent of C, the speculation was correct and no further action is required.

However, the speculation can be incorrect under two scenarios. Figure 6.10 redraws the sequential task invocation order from the nesting example of Figure 6.9. Say task D reads object O which task C writes, as depicted by the RAW dependence arrow from D to C in Figure 6.10a. The first case of misspeculation arises from out-of-order invocation of nested tasks, as shown in Figure 6.10a. Task D is invoked in t3 and is executed speculatively. It reads the object O. When task C is invoked in t4

and declares its dataset, the Token Manager detects the violation of the RAW dependence between D and C. The second case of misspeculation arises from non-eager dataset declaration. Say, for the same nested code, tasks C and D are invoked in the correct order, as shown in Figure 6.10c. But in this case, say C declares the dataset either lazily or JIT. When D is invoked in t4, the dataflow graph between C and D cannot be constructed because C has not yet declared its dataset. Nonetheless, D is executed speculatively. Task D reads the object in t4. In t4, C finally declares its intent to write object O. The Dependence Analyzer completes the dataflow graph, and detects that the RAW dependence between D and C has been violated. In both cases, task D has misspeculated. Its execution and of any data- and control-dependent tasks (none in our example) was incorrect. Irrespective of the scenario, misspeculation has the same effect–incorrect execution.

To rectify the misspeculation, the GRX performs **dataflow recovery**, by undoing the operations of only the misspeculated and dependent tasks, and re-executing them. The ROL and the HB are repurposed, with some extensions, for this purpose. Before a task executes, its mod set is already checkpointed in the HB. To undo the task's operations, the checkpoint is restored from the HB. Once the correct state has been reinstated, the execution resumes in the correct dataflow order, as dictated by the more accurate dataflow graph.

Supporting speculative execution requires detecting and correcting misspeculation. This requires dynamically tracking the precise order of tasks and managing their dependences. Maintaining an ordered view of the execution helps with this process. We discuss these aspects next.

Tracking Task Order.

Tracking a linear task's order is relatively easy; it is the same as its invocation order. A strictly monotonic ID may be assigned to the task for tracking purposes. Nested tasks pose a challenge.

Parakram logically decomposes a nested task's parent into *subtasks*, at the task's call site, as shown in Figures 6.11a and 6.11b. In Figure 6.11a, task A is divided into subtask A1, comprising operations up to task B, and subtask A2, comprising computations between invocations of B and D, as marked. Parakram now views the program as comprising these ordered subtasks, shown in

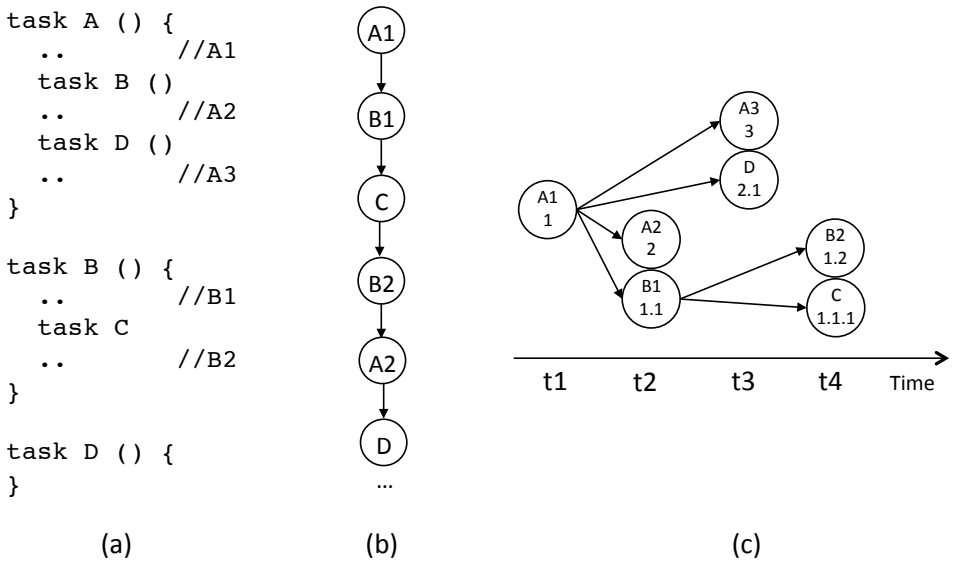


Figure 6.11: Handling parallel execution of nested tasks. (a) Example pseudocode. (b) Tasks divided into subtasks for order tracking. (c) IDs based on a hierarchical scheme.

Figure 6.11b. Note that this is consistent with the program's sequential order. (Similar schemes have been used by others for sequential programs [6, 152].)

Parakram permits each subtask to declare its dataset using `pk_declare()`. If undeclared, the parent task's dataset is *inherited* by the subtask. This is in contrast to the CF task, in which case an undeclared dataset implies an empty dataset. When nested tasks are used, the programmer may choose to declare the datasets of individual subtasks, which may yield higher parallelism.

Parakram uses a hierarchical numbering scheme, inspired by the Dewey Decimal Coding system [53], to assign unique, but ordered identifiers to the subtasks. The identifier is formed by concatenating "fractions", one per nesting level of the task. Each fraction is strictly monotonic. The nested task inherits its parent's identifier and appends a fraction in the least significant position. The succeeding subtask inherits the parent's identifier and the least significant fraction in it is incremented. Figure 6.9e labels the tasks with the IDs formed using one such scheme. How this scheme is used in practice is described later.

#	Issue	Dependence Management				
		Data-independent		Data-dependent		
	(1)	Eager (2)	JIT/Lazy (3)	Eager (4)	JIT (5)	Lazy (6)
1	Dependence	Known	Unknown	Known	Unknown	Unknown
		Dataflow	*Speculation*	*Dataflow*	*Speculation*	*Speculation*
2	State write	-	Speculative	-	Speculative	Speculative
		Eager	*Eager*	*Eager*	*Eager*	*At commit*
3	State read	Correct	Misspeculation	Correct	Misspeculation	Misspeculation
		-	*Rollback*	-	*Rollback*	*Rollback+Exception*

Table 6.12: Summary of dependence management issues. Each row lists the possible issue on the first line and Parakram's approach on the next.

Managing Dependences.

Managing dependences of eager tasks is relatively straightforward. But when tasks execute speculatively, out of order, we need to ensure that the states tasks read and update are correct. Table 6.12 lists the issues and how Parakram handles them (the Parakram approach is listed in italics below the related issue in the table).

When a data-independent dataset is declared eagerly (column 2), Parakram simply performs dataflow execution (row 1). Any state updates (row 2) and reads (row 3) are correct.

Although data-independent datasets can be declared eagerly, programmers may declare them JIT or lazily (column 3). In that event, the task starts executing speculatively since the dataset is unknown to the Execution Manager (row 1). Parakram allows the task to update the program's state *eagerly* (row 2), i.e., it becomes visible to all other tasks immediately. If Parakram subsequently discovers that it had misspeculated, the dataflow recovery repairs the state, and dependent tasks read the correct state upon re-execution (row 3).

If the dataset is data-dependent, the programmer may construct the program such that the dataset is computed outside of the task (e.g., in a CF task) and the task declares the dataset eagerly (column 4). This case is similar to tasks declaring the dataset eagerly, and hence handled similarly.

A data-dependent dataset may be declared JIT (column 5). Parakram executes the task speculatively (row 1) and permits eager updates (row 2). It tackles misspeculation using the dataflow recovery, and re-executing the affected and dependent tasks (row 3).

Lazily declared data-dependent datasets pose a unique challenge since such tasks are allowed to read data without first declaring them (column 6). We call such tasks, *lazy*, and require that they be invoked through the `pk_task_uc()` API. They are executed speculatively (row 1). Lazy tasks may read data to compute the dataset. Another executing task that speculatively updates data eagerly can influence the dataset. For programs that use lazy tasks, Parakram permits them to update the state upon **commit**, i.e., when the task's dependences are known for certain and the task is known to have not misspeculated (row 2). Conservatively, this is always the case when a task becomes **non-speculative**, i.e., becomes the oldest in the execution. Parakram presently relies on the programmer to perform updates locally, but updates the global state when the task commits (using the user-provided assignment operator). A task can still read incorrect data (if the producing task has not yet committed its updates). This case is detected as a misspeculation, and the task is rolled back and re-executed (row 3).

Further, a lazy task may read an object being modified by another task (even if it is being updated on commit). Since the object will be in the task's dataset (data read to compute the dataset is also part of the dataset), the conflict will be detected eventually and rectified. But the task may read inconsistent data in the meantime.

Programmers often use assertions to test invariant conditions. Such assertions can also potentially catch inconsistencies that may arise in lazy tasks. Parakram allows the program to report exceptions on such assertions to the Execution Manager, using `pk_except()`. These exceptions will cause the task to (eventually) re-execute. We successfully used this scheme in some programs, e.g., DeMR. Exceptions in non-speculative tasks are reported to the programmer.

Special Cases.

Certain algorithms may use datasets which are dependent on data structures that are not modi-

fied. In such cases, tasks may declare the datasets lazily, but may not require the full support for speculation. For example, in one phase of its processing, Genome (Table 5.10) maps its input strings into a hash table. Finding the slot in the hash table in which a string will be inserted requires computing the hash of the string (and hence data-dependent dataset), but the input strings themselves do not change. Hence, a task's dataset is not affected by other concurrently executing tasks. In such cases, although the datasets are declared lazily, order can be maintained and misspeculations can be entirely avoided if we simply wait to schedule tasks until the dataflow graph is constructed completely. This can be done by processing the datasets in the program order. Parakram permits programmers to identify such tasks and guide the dependence management. Programmers may declare the dataset using the `pk_declare_ia()` API, and request Parakram to perform in-order token acquisition. When a task declares its dataset in this fashion, Parakram assumes that the task need not be re-executed, even if the task started executing speculatively and declared its dataset out of the program order.

6.3.5 Scheduling Tasks for Execution

Once the Speculative Dataflow Engine (SDX) has gathered parallel tasks, speculative or not, in the Task Pool, the Order-aware Task Scheduler (OTS) schedules them for execution on hardware contexts. Cilk-5 has popularized load-balancing task schedulers over the recent years due to their provably optimal properties [71]. They create tasks lazily, use the work-first principle to unroll the program only when a resource is available, and use randomized task-stealing to balance the load in the system. OTS borrows some of these principles, but introduces new features to support the dataflow and speculative execution.

Schedulers for multithreaded programs, like Cilk-5, treat all tasks as "equal" and randomly schedule them when resources are available. In practice, programs are constructed such that tasks whose results will be needed by other tasks are performed sooner. This is usually reflected in the program encoding by the order in which the tasks are performed. OTS takes advantage of this order and prioritizes older tasks over younger tasks for scheduling. In our experiments, we observed that

in some algorithms like Cholesky decomposition, given a pool of ready tasks, executing an older task exposed more downstream parallelism. Hence OTS ensures that older tasks get scheduled before younger tasks from the Task Pool.

OTS employs two schemes to distribute work for execution. The first permits individual hardware contexts to run schedulers that actively seek work by querying the Task Pool, analogous to task stealing. The second assigns tasks to hardware contexts, eliminating the schedulers. Our experiments show that the first performed better for programs with relatively larger tasks whereas the second worked better for programs with very small tasks. The current prototype allows the programmer to choose between the two. An hybrid approach and an automatic, adaptive scheme to dynamically pick between multiple schemes can be an interesting future direction to pursue.

6.3.6 Other Aspects of Ordered Execution

Performing computations in parallel is but one aspect of executing multiprocessor programs. Other issues arise in this process, such as managing resources, executing computations efficiently, and performing I/O correctly. Parakram relieves the programmer from explicitly coordinating the execution, but introduces its own mechanisms. Parakram must ensure that these mechanisms do not alter the program semantics. Further dataflow and speculative execution can introduce new issues that must be tackled to create a viable solution. We discuss these aspects next.

Efficiency. Applying dataflow principles incurs overheads. The overheads can defeat the parallelism if the principles are applied to computations that are too fine-grained [163]. Parakram applies these principles to tasks, which are sufficiently coarse-grained, thus justifying the associated overheads. Nonetheless, task granularity plays an important role in how successfully the parallelism can be exploited, as will be seen in Chapter 7.

Resource Management. Another drawback of the classical dataflow execution was the possibility of resource deadlocks [47]. Dataflow machines can easily scout an entire program for parallelism

even when only a fraction of the program has actually executed. In the process they can exhaust resources, causing deadlocks. We prevent such deadlocks by unraveling the ordered program only as much as the resources permit, in an approach similar to superscalar processors which apply the dataflow principles to a relatively small window of instructions. As long as resources are available to execute at least one task, Parakram can ensure forward progress since it can always permit the oldest task in the system to proceed while suspending all other tasks. The Execution Manager's approach to unwinding the program ensures that it always has access to the oldest in-flight task. The Execution Manager can control the amount of program that is unwound in search of parallelism to match the available resources.

Tackling Deadlocks. The token mechanisms can potentially create deadlocks. We ensure that the mechanisms either do not create deadlock situations, or if they are about to, we avoid them with the help of order. We divide this discussion into two cases: basic dataflow execution and speculative execution.

The token mechanism in the basic dataflow execution could deadlock if two or more tasks create a cyclic dependency on tokens. For example, in the Figure 6.7b, invocations F4 and F5 may create a request sequence F4:B (acquired) $\rightarrow$ F5:B (waiting) $\rightarrow$ F5:D (acquired) $\rightarrow$ F4:D (waiting) and deadlock. This scenario cannot occur because: (i) token requests are processed one task at a time, and (ii) tokens for an object are granted in the order they are requested. Essentially, tokens are acquired strictly in the task order. Thus F5's token requests are only processed after F4's, and F5 can only receive tokens to its objects after all previous requesters have relinquished them.

The ordered token acquisition is violated when tasks are nested or when the datasets are not declared eagerly. Nesting can cause concurrent unrolling of parallelism. Non-eager datasets permit multiple tasks to execute before any of them declares its dataset. In these cases concurrently executing tasks can attempt to acquire tokens simultaneously (and also out of order). Now cyclic dependency on tokens can arise, leading to deadlocks. In all of these cases we leverage order to resolve the deadlock by breaking it in the favor of the older task. The younger task is aborted, any

granted tokens are reclaimed from it, and granted to the older task. (The younger task is re-executed in the correct order with the help of speculation recovery.) The hierarchical task IDs are used to determine the task order.

Performing I/O. Parallel execution, irrespective of the programming interface used, complicates I/O operations. Program's computations may execute in parallel, but most I/O must be performed in a serial order. In multithreaded programs the programmer must explicitly coordinate the I/O operations to ensure that they are performed in the correct order. To do so efficiently can complicate the design. The Pthreads implementation of Pbzip2 in the last chapter is an example of the added complexity (Listing 5.4). Speculative Multithreading approaches, such as Transactional Memory, can further complicate matters since speculatively executed computations may have to be re-executed. If they perform operations such as network I/O, those actions may not be retractable.

Ordered execution naturally simplifies I/O operations. Despite the parallel execution, tasks that perform I/O to an interface naturally execute in the program order, since the interface acts as a shared object. This relieves the programmer from any explicit coordination.

However, I/O performed by speculative tasks can result in inconsistent state if the task is re-executed. If the I/O is idempotent, e.g., file reads and writes, Parakram provides support for automatically rolling back such operations as a part of the misspeculation recovery. For non-idempotent I/O, e.g., the network operations, Parakram permits tasks to perform I/O only when the tasks become non-speculative. The programmer can invoke such linear or nested tasks tasks using the `pk_ic()` API. It is always safe to execute such tasks when they are ready to retire. The GRX submits the task enclosed in `pk_ic()` when it reaches the ROL-head.

Exceptions. Handling exceptions is another aspect complicated by parallel execution, especially if the execution is speculative. Ideally, it is desirable to report exceptions to the programmer in the program order so that the user can track the exception to the dynamic instance of the corresponding task.

Ordered execution is also helpful in handling exceptions. The GRX catches user-invoked exceptions, `pk_except()`, as well as system exceptions, e.g., SIGSEGV. They are reported to the programmer or the excepting task only when the task becomes non-speculative.

Furthermore, if so directed, the GRX can roll back and re-execute the excepted task. This feature may be used if the user expects the exception condition to be transient. We applied this feature gainfully in DeMR and Labyrinth. Since a task in DeMR traverses the graph while another is modifying the graph, it can read inconsistent data. While this is a dependence violation, and will be treated as such, reading inconsistent data is also an exception condition. However, once the updates to the graph are complete, the data will become consistent. In this case the exception condition was transient and it is desirable to retry the task without throwing an exception to the programmer. This also serves the purpose of correcting the dependence violation.

In the case of Labyrinth, we used exception support to reduce dataset processing overheads. We will revisit Labyrinth in the next chapter once we see the Execution Manager's operations in more details next.

If the task is non-speculative, and the programmer did not request re-execution, the exception must and will be reported to the programmer.

6.4 Execution Manager Prototype

To evaluate Parakram we developed a software prototype of the Execution Manager in the form of a compiler- and OS-agnostic C++ runtime library. The library supports the Parakram APIs and implements the mechanisms described in this chapter. Applications to be parallelized are compiled with the library; the library becomes a part of the program that it dynamically parallelizes. The library itself is implemented using the Pthreads API. However, it abstracts away the underlying concurrency mechanisms, such as threads, synchronization operations, etc., from the programmer, as originally intended. The library design follows the overall architecture presented in Figure 6.5.

We describe the prototype operations in three parts: basic dataflow execution, speculative

```
1  schedule ( ) {
2    while ( always ) {
3      obtain work from Task Pool
4      // work can be a task or the program continuation
5      if dataset declared {
6        clone work's (write set ∪ mod set)
7      }
8      execute work
9    }
10 }
```

Figure 6.13: Logical operations of the thread scheduler.

execution, and miscellaneous features. We will describe the operations with the help of pseudocode, which will be self-evident from the figures.

6.4.1 Basic Dataflow Execution

Runtime Basics.

At the start of a program, the runtime creates threads, usually one per hardware context available to it. A double-ended work queue (deque) is then assigned to each thread in the system. Tasks are submitted for execution by queuing them in the work deques. The work deques collectively represent the Task Pool (Figure 6.5 ❹). Each thread also uses a task-stealing scheduler, which continually seeks work from the Task Pool by querying the deques, and dequeuing work from them if available. Figure 6.13 shows the logical operations performed by the scheduler. The details will be seen below.

Discovering Tasks for Parallel Execution.

At the onset, all but one of the processors idle, waiting for work to arrive in the deques. Execution of the program begins on a single processor and unfolds in a fashion similar to sequential execution.

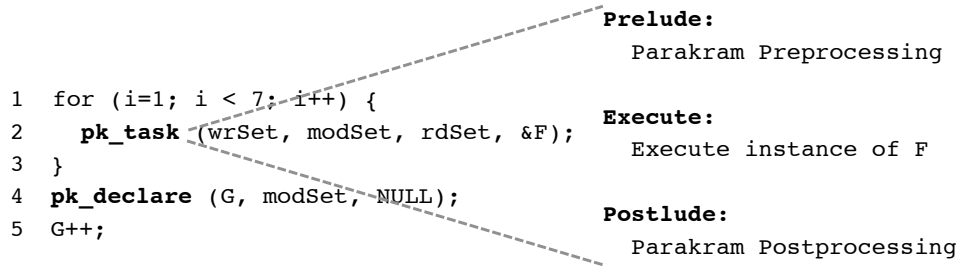


Figure 6.14: Logical Execution Manager operations to process a dataflow task.

When a function to be considered for parallel execution, e.g., via `pk_task()`, is encountered, the Sequencer in the runtime is activated. The Execution Manager expands the API into three decoupled phases: **Prelude**, **Execute**, and **Postlude**, as logically shown in Figure 6.14. Execute may not be immediately performed after Prelude, as will be seen, but Postlude follows Execute.

In the Prelude phase, shown in more detail in Figure 6.15 ❶, the Sequencer first assigns a hierarchical ID to the task. The Sequencer then logs an entry for the task at the end of the ROL. It then dereferences pointers to objects in the read and write sets, and attempts to acquire the tokens. Successful acquisition of tokens leads to the Execute phase, detailed in Figure 6.15 ❷, in which the task is submitted for (potentially parallel) execution. Specifically, the runtime pushes the program continuation (remainder of the program past the `pk_task()` call) onto the thread's work deque, and executes the task on the same thread. A task-stealing scheduler, running on each hardware context, will cause an idle thread to steal the program continuation and continue its execution, until it encounters the next `pk_task()`, repeating the process of submission and pushing of the program continuation onto the work deque. Thus the execution of the program unravels, in parallel with executing tasks, and possibly on different hardware contexts rather than on one hardware context. The Postlude phase is described later in the subsection.

Tokens and Dependency Tracking.

The Token Manager implements a *Token Protocol* to manage the dependences between tasks and

```
1  Prelude ( task ) ❶ {
2    enqueue task in ROL
3    if task is nested {
4      mark task as control-dependent on parent
5    }
6    if task is eager {
7      Acquire tokens 1.1 // See Figure 6.16
8      if all tokens not acquired {
9        Shelve task 1.2
10       return to program
11     } else {
12       Shelve program continuation
13       Execute task // See below
14     }
15   } else {
16     Shelve program continuation
17     Execute task // See below
18   }
19 }
20
21 // All tokens acquired
22 Execute ( task ) ❷ {
23   if dataset declared {
24     clone task's (write set ∪ mod set)
25   }
26   execute task 2.1
27 }
28
29 // Task completed
30 Postlude ( task ) ❸ {
31   Release tokens 3.1 // See Figure 6.16
32   Restart misspeculated tasks ( task ) // See Figure 6.20
33   Retire ( task ) 3.2 // See Figure 6.17
34 }
```

Figure 6.15: Prelude, Execute, and Postlude phase operations of the Execution Manager.

```
1
   Token Availability
2    Initial State:
3      Read Token: None granted.
4      Write Token: Not granted.
5
6    Read Token is available if:
7      Write Token is not granted.
8
9    Write Token is available if:
10     no Read Token is granted AND
11     Write Token is not granted
```

```
   Acquire Read/Write Token:
14 acquire ( task ) {
15   enqueue in Wait List
16   if younger task acquired token {
17     Rectify misspeculation () {
18     // See Figure 6.19
19     }
20   }
21   for each waiting task in Wait List {
22     if token is available {
23       grant token to waiting task
24     }
25   }
26 }
```

```
   Release Token (Read/Write):
29 release () {
30    return token
31    while Wait List is not empty {
32      if token is available {
33        grant token
34        if waiting task is ready {
35          submit task to Task Pool
36         }
37       } else {
38         break
39    }
40 }
```

Figure 6.16: The Token Protocol. Definition of availability, read/write token acquistion, and token release.

implicitly creates the dataflow graph. The Token Protocol is shown in Figure 6.16. During the program execution, when an object that inherits `token_t` is allocated, it receives one write token, unlimited read tokens (limited only by the number of bits used to represent tokens), and a **Wait List**. Tokens are acquired for objects that the dataflow tasks operate on, and are released when the tasks complete, or when the task uses `pk_release()` to explicitly release the objects.

A token may be granted only if it is available. Figure 6.16 gives the definition of availability of read and write tokens (lines 2-11), and shows the token acquisition protocol (lines 14-23). The Wait List tracks the current in-flight task(s) that have been allocated tokens, and the tasks to which the token could not be granted at the time of their requests. To prevent deadlocks, all token requests from a task are processed before proceeding to the next task, possibly in parallel with other executing tasks, and a token is granted strictly in the order in which it was requested. Hence an available token is not granted if an earlier task enqueued in the Wait List is waiting to acquire it. For example, if a read token is available but a prior task requires a write token to the object, it cannot be granted to the requester, ensuring that tasks acquire tokens in the program order.

Shelving Tasks/Program Continuations.

After a task's requests are enqueued in the Wait Lists of all the objects (Figure 6.16, line 14), and if it is unable to acquire all of the requested tokens, the task is shelved (Figure 6.15 1.2). While the shelved task waits for the dependences to resolve, the runtime looks for other independent work from the program continuation to perform, as above.

`pk_serial()`, `pk_barrier()`, `pk_declare()`, `pk_read()`, and `pk_write()` can cause the program continuation to be shelved. The continuation is enqueued in the Wait List of the object(s) identified in `pk_declare()`, `pk_read()`, and `pk_write()`. `pk_serial()` and `pk_barrier()` cause the program continuation to be shelved until all relevant executing tasks complete. In this event, the continuation is shelved on a special runtime structure (and not a Wait List).

Since the program is now suspended in either case, the context looks for previously shelved tasks that may now be unshelved, via task stealing.

Executing Tasks.

Once the scheduler obtains a task from either its own deque or steals a task from another thread's deque, the objects in its mod set and write set are cloned. The object clones are held in the HB. The control is then transferred to the task to perform the actual work (Figure 6.15 ❷.1).

Completion of Tasks.

When a dataflow task completes execution it returns the control to the runtime, which initiates the Postlude phase (Figure 6.15 ❸). In this step all acquired tokens are released. Figure 6.16 describes the token release protocol (lines 29-40). Once a token for an object is relinquished (Figure 6.16, line 30), it is passed to the next waiting task(s) in the Wait List (Figure 6.16, line 31), if present, in the same order that the tasks were enqueued. When a shelved task receives all requested tokens, it is considered to be ready for execution. The thread that grants the final token also awakens the ready task and schedules it for execution by enqueuing it in its own work deque (Figure 6.16, line 35). Then either the same thread, or another that steals the task, will eventually execute it (Figure 6.15 ❷). The process is identical if a `pk_declare()` caused the program continuation to be enqueued in a Wait List. Different threads may simultaneously attempt to release, acquire, and pass a token. The runtime manages the data races in such scenarios by using non-blocking concurrent data structures.

Retiring Tasks.

Once all tokens are relinquished, the GRX is reported about the completed task. The GRX attempts to retire the task (Figure 6.15 ❸.2). Figure 6.17 gives more details. The GRX checks whether the task is at the head of the ROL. If it is not then the task cannot be retired. If the task is at the head of the ROL, its entry is removed, the clones are removed from the HB, and the memory held by clones are is recycled.

If the continuation was shelved, the Postlude phase of the last task to finish will submit the continuation for execution, resuming the program.

```
1  Retire ( task ) {
2    while next task at ROL-head completed or excepted {
3      if next task excepted {
4        Handle exception ( next task ) // See below
5      }
6      if next task requested update on commit {
7        // For tasks invoked via pk_task_uc()
8        Acquire tokens ( next task )
9        // All tokens should get acquired since this is the oldest task
10       submit next task to Task Pool
11       exit Retire
12     }
13     for each object in next task's (write set ∪ mod set) {
14       recycle clone
15     }
16     for each object in next task's read and write set {
17       remove next task's entry from Wait List
18     }
19     remove next task's entry from ROL
20   }
21   jump to schedule () // See Figure 6.13
22 }
23
24 Handle exception ( task ) {
25   pause execution
26   for each younger task in reverse ROL order {
27     abort younger task
28     for each object in younger task's (write set ∪ mod set) {
29       restore from clone
30     }
31   }
32   report exception to user
33   // Further action is user-dependent
34 }
```

Figure 6.17: GRX operations to retire a task.

Thus, by shelving dependent tasks, scheduling them for execution as soon as their dependences have resolved, and executing other independent tasks in the meantime, the Parakram achieves the basic dataflow execution.

6.4.2 Speculative Execution

Each thread in the Execution Manager tracks whether it is currently executing a control-flow task or a dataflow task. The Execution Manager automatically switches to speculative mode when a `pk_task_uc()` is invoked, or a `pk_task()`, `pk_task_uc()`, `pk_declare()`, `pk_read()`, or `pk_write()` is invoked from a dataflow task, since the task may have been invoked, or the dataset may have been declared out of program order. In this mode, the Execution Manager submits tasks to the Task Pool in anticipation of the need to rectify misspeculations arising from out-of-order token processing.

Nested Tasks.

When the Sequencer encounters a task invoked from a dataflow task, indicating that it is nested, the Prelude, Execute, and Postlude phases remain the same as before. In the Prelude phase, when the entry for the task is logged in the ROL (after the task has been assigned its hierarchical ID), the task is inserted in the position which reflects the task's relative order among the tasks currently in the ROL (Figure 6.15, line 2). The hierarchical ID is used for this purpose. (There are alternative methods to insert entries in the ROL.) The invoked task is also marked as control-dependent on the parent task (Figure 6.15, line 4).

Non-eager Datasets.

When the Sequencer encounters `pk_declare()`, `pk_declare_ia()`, `pk_read()`, or `pk_write()`, it initiates non-eager dataset handling, as shown in Figure 6.18. The Sequencer marks the parent task as speculative. Tokens are acquired next.

```
1  non-eager declaration ( task ) {
2    mark parent task as speculative
3    if in-order token acquisition requested {
4      while oldest task in ROL whose dataset has been declared but not processed {
5        Acquire tokens ( task ) // See Figure 6.16
6        if all tokens acquired {
7          submit oldest task to Task Pool
8        } else {
9          Shelve oldest task
10         jump to schedule () // See Figure 6.13
11       }
12     }
13     jump to schedule () // See Figure 6.13
14   }
15 }
```

Figure 6.18: Execution Manager operations to process non-eager dataset.

During token acquisition, The Token Manager first inserts the request in the ROL (Figure 6.16, line 15) in its correct position (using the hierarchical task ID). In this process it can identify whether a younger task was issued the requested token. If it was not, the processing proceeds as before. If the younger task was issued the token and it performed a conflicting access, then the younger task executed speculatively and has violated a dependence. The GRX is informed about the misspeculation.

Misspeculation Recovery.

The GRX initiates misspeculation recovery, as shown in Figure 6.19. It walks the ROL and creates a transitive closure of tasks data-dependent or control-dependent on the precedent task. They are all designated to be misspeculated. Misspeculated tasks are aborted. Their modifications are unrolled by restoring the objects in their datasets from the clones. Any state unrolling is performed from the youngest to the oldest task. If a misspeculated task was control-dependent, its entry

```
1  Rectify Misspeculation ( misspeculated task ) {
2    // tasks_to_abort is an ordered set of tasks sorted by task ID
3    tasks_to_abort = misspeculated_task
4    for each task younger than misspeculated task in ROL {
5      if task is data-dependent on any task in tasks_to_abort {
6        tasks_to_abort += task
7      }
8      if task is control-dependent on any task in tasks_to_abort {
9        tasks_to_abort += task
10     }
11   }
12   for each task in tasks_to_abort in reverse {
13     abort task
14     if task is control-dependent on any task in tasks_to_abort {
15       for each object in task dataset {
16         restore state from clone
17         remove task from Wait List
18       }
19       remove task from ROL
20     } else {
21       if task is speculative {
22         for each object in task dataset {
23           restore state from clone
24           remove task from Wait List
25         }
26         enqueue task in misspeculated task's Restart List
27       }
28     }
29   }
30 }
```

Figure 6.19: Operations performed to rectify misspeculation.

```
1  Restart misspeculated tasks ( given task ) {
2    for each task in restart list of given task {
3      Restart misspeculated tasks ( task )
4      submit task to Task Pool
5    }
6  }
```

Figure 6.20: Restarting misspeculated tasks.

is removed from Wait Lists of its dataset and the ROL (because control-dependent tasks will be invoked again). Of the remaining, if the task was marked speculative, its entry is removed from Wait Lists of its dataset (they will re-declare their datasets). These tasks are enlisted in the Restart List of the precedent task. Now the precedent task's execution can begin.

Completed tasks.

When a task completes, in addition to processing as before, it also now checks whether any previously misspeculated non-eager data-dependent tasks are waiting for its completion. Such tasks are in the restart list. A transitive closure of tasks in the Restart List are submitted to the Task Pool (Figure 6.15, line 29 and Figure 6.20).

Retiring Tasks.

Recall that tasks invoked via `pk_task_uc()` update state lazily. When a task is being retired, the GRX processes its dataset and requests the Token Manager to acquire the tokens (Figure 6.17, line 14). At this point, since the task is the oldest (it is being retired), all its tokens will be acquired and the task is submitted for execution to the Task Pool. When the task is (eventually) scheduled, it can then complete its processing, by committing its locally computed state to the architectural state .

All other operations remain the same as in basic dataflow execution.

6.4.3 Other Operations

Scheduling and Balancing Load.

The OTS is implemented in the form of asynchronous distributed schedulers, one assigned to each thread. There are two key aspects to the scheduling mechanism: task-stealing and continuation scheduling. Each thread's scheduler seeks work when the thread is idle. The scheduler first executes tasks from its own thread's deque when available, failing which it randomly steals from others threads. This distributed, asynchronous system avoids bottlenecks that result from centralized resources, and helps balance the system load.

The work deques are polymorphic–they can hold tasks and program continuation. The thread that awakens a shelved task or program continuation, enqueues it in its own deque, potentially executing it next. Scheduling dependent tasks on the same thread as the precedent task permits the dependent task to migrate to the thread that may have already cached the data.

Although popular, continuation scheduling and random task-stealing are not ideal. In continuation scheduling, when a task is invoked it is preferred that the task execute on the same context, while the program continuation may move to another context (via task-stealing). This operation was described in the description of basic dataflow execution earlier. This scheduling scheme is adequate when the tasks are reasonably large. But for small tasks, it can be grossly inefficient. Continuation scheduling results in cache misses every time the continuation moves, and task-stealing can unnecessarily chew up memory bandwidth. Furthermore, this scheme does not take into account task order, which contains valuable information about the user's intention.

We implemented two further enhancements in the OTS. First, we implemented task-distribution in addition to task-stealing. In this scheme, the main program executes on a single context, and invoked tasks are distributed to other processors. Other processors need not continuously seek work; work is assigned to them The user may instruct the Execution Manager to chose task-distribution over task-stealing.

Second, we ensured that tasks are enqueued in the work deques in a sorted order, such that

older tasks are scheduled for execution before younger tasks. This scheme ensures that possibly precedent tasks execute before other tasks. It becomes especially critical in speculative execution. Any task can be executed speculatively. Younger tasks may get scheduled before older tasks, only to misspeculate. If care is not taken, this can result in wasted work, and even slowdowns.

Memory Allocation.

Programs often use dynamic data structures which are allocated and deallocated as the program executes. Additionally, the Execution Manager employs several data structures, e.g., the ROL, the HB, and the clones, for its own operations. The Execution Manager uses dynamic memory allocation and deallocation to minimize its memory overheads. These operations are performed for almost every task it processes. Dynamic memory allocation poses two challenges to Parakram. The first arises due to poor performance of prevailing memory allocators [8], which is an issue that affects multiprocessor programming in general, but more so Parakram-like systems that may use the memory allocators more frequently. The second challenge arises due to speculative execution. Speculative tasks that allocate and/or deallocate structures may be re-executed, which can leave memory allocators in inconsistent state and exacerbate memory use.

Parakram runtime addresses both challenges by implementing its own high performance memory allocator. The allocator creates thread-local pool of memory resources from which both the user program and Parakram allocate structures. The allocator maintains only as large a pool as the program requires. By implementing its own allocator, the runtime can also support speculative allocation, deallocation, misspeculation recovery, and re-executions.

All allocator requests by a task are logged. Any allocation requests are immediately serviced, but deallocation requests are deferred until the task retires. Objects allocated in a speculative task are automatically deallocated if the task is squashed. To recover from misspeculations, the log is used to gracefully undo operations.

Data Structures.

The runtime is essentially a parallel program whose threads operate asynchronously on its own internal data structures. The ROL, Wait Lists, work deques, and memory allocator pools can be potentially accessed concurrently. We investigated multiple designs to implement these data structures [40, 68, 91, 126, 167]. Concurrent data structures are already complex to design. Frequent allocation and deallocation of fields in the data structures further complicate matters (e.g., the ABA problem [167]). The runtime uses a mix of strategies, tuned to specific uses of the structures, to implement high performance designs.

Whenever possible, the runtime creates thread-local private structures, e.g., the memory allocator pools. In all other cases it uses non-blocking concurrent data structures. For the work deques we implemented a scheme similar to the one proposed by Chase and Lev [40]. For the ROL and Wait Lists we used combining synchronization schemes similar to the ones proposed by others [68].

6.4.4 Example Executions

We now illustrate the runtime operations using code examples by simulating their parallel execution. We first present the basic dataflow execution and then the speculative execution.

Basic Dataflow Execution.

Consider the the example code in Figure 6.8a. Assume that it is executing on a three-processor system. Figure 6.21a shows the initial state of the system. P1 is executing the main program, and objects have been constructed. Objects A to F are depicted along with their tokens. $R = n$ indicates that n read tokens have been granted; W indicates the write token has been granted (none at this time). Wait Lists of objects A, B and D, currently empty, are also shown. (We omit the ROL operations from this basic dataflow case, but describe them in the speculative part.)

Figure 6.21b shows the state of the system as the program begins execution (the same `while` loop as in Figure 6.8). Events are identified using ❶ time stamps. The first invocation of F, F1, with write set {B, C} and read set {A} is encountered. Write tokens for B and C, and a read token for A

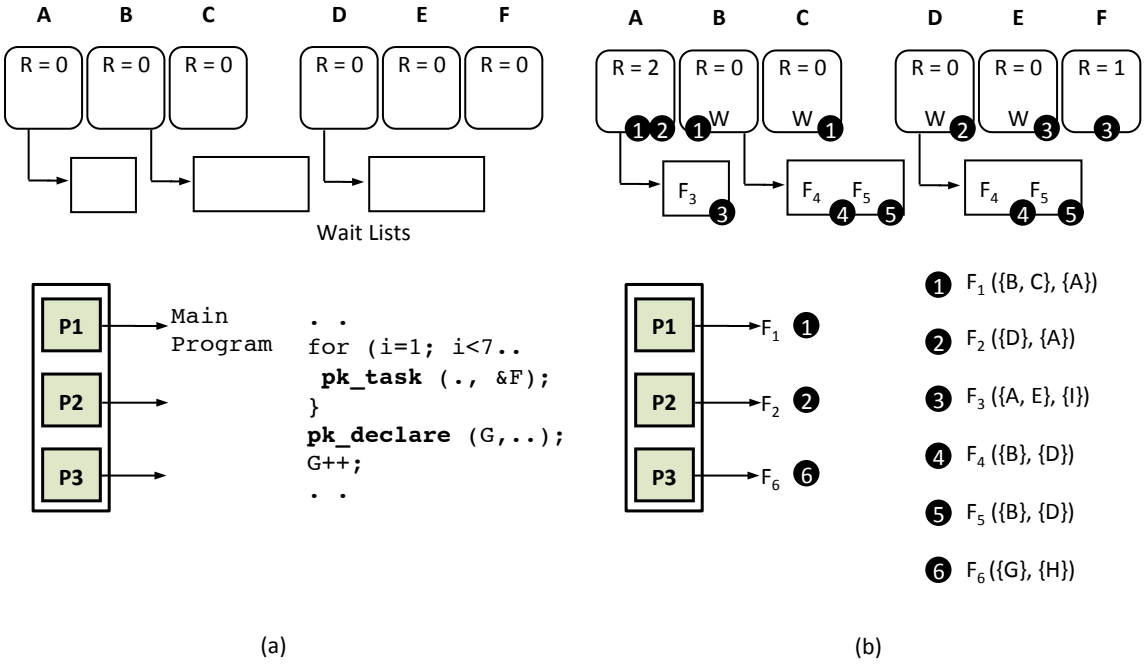


Figure 6.21: Example execution. (a) Initial state of the system. (b) State of the system after processing the six invocations of task F, denoted as: F write set read set.

are acquired ❶ and the task is delegated for execution on P1 ❶, and the program continuation is pushed on to the deque of P1.

From there, P2 steals the continuation, and encounters the second invocation, F2. It acquires tokens for objects D and A, and delegates F2's execution on itself ❷. Note that at this time, two read tokens for object A have been granted. Next, the third invocation, F3, attempts to acquire a write token to A, and fails (however it succeeds in acquiring tokens to E and F ❸). It is hence shelved and enqueued in the Wait List of A ❸. The next invocation, F4, also fails to acquire a read token to object D and a write token to B, and is enqueued in their Wait Lists ❹. Likewise, the following invocation, F5, has to be enqueued in object B and D's Wait Lists ❺. While the shelved methods await their tokens, the main program context on P3 reaches the final invocation of F from the loop, F6, which acquires tokens to G and H (not shown), is delegated for execution on P3 itself ❻.

Upon completion, F1 releases the tokens back to objects B, C and A (❼ in Figure 6.22a). This

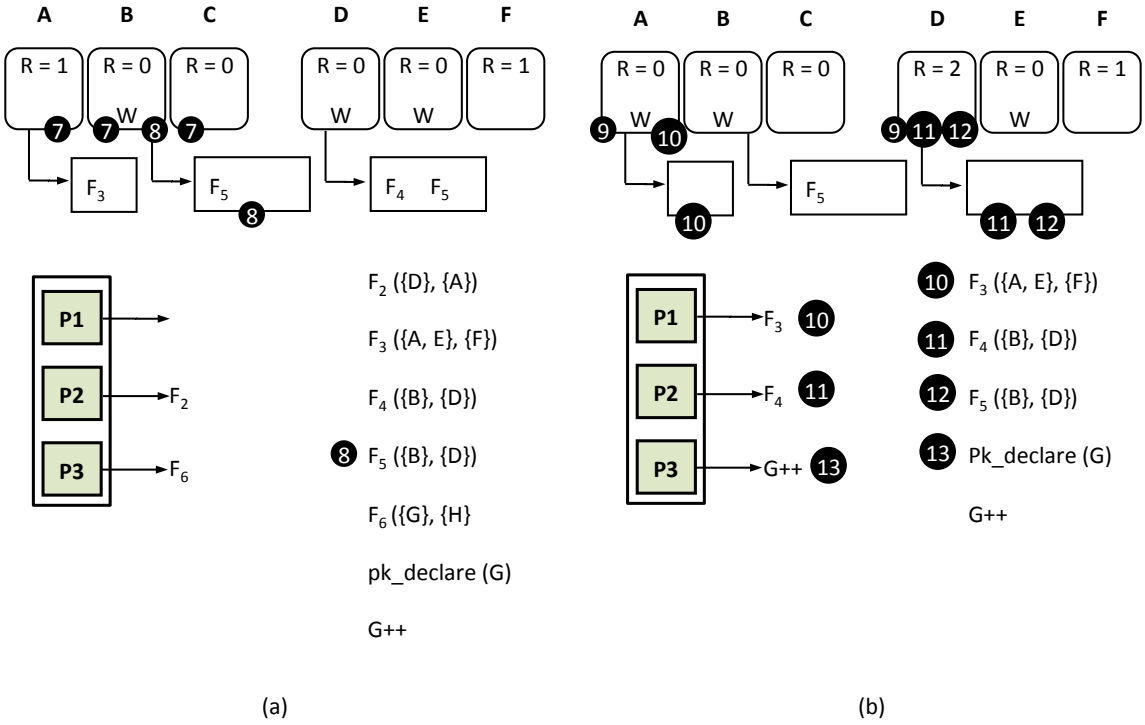


Figure 6.22: Example execution. (a) State of the system after invocation F1 completes execution. (b) State of the system after invocations F2 and F6 complete execution.

causes B's write token to be granted to F4 ❽. However F4 cannot execute yet since not all of its requested tokens have been granted. Next P1 steals the program continuation from P3 and encounters `pk_barrier()`. It causes the runtime to shelve the program continuation beyond `pk_barrier()` in G's Wait List (not shown) and await completion of all tasks accessing G (only F6 in this case), before advancing further.

Eventually, F2 and F6 complete (Figure 6.22b), and release their tokens ❾. Now that all of its read tokens are available, the write token to A is granted to F3 ❿. Since F3 now has all its tokens, it is enqueued in the work deque, from where the next available context will execute it ❿. Similarly, once the write token is returned to object D, read tokens are granted to F4 and F5 (⓫, ⓬). This causes F4 to transition to the ready state, and be scheduled for execution on an available context ⓫. Completion of F6 causes the runtime to execute the program continuation (`G++`) ⓭. F5 will be

scheduled for execution once F4 completes and releases the tokens to objects B and D.

Thus the basic dataflow execution takes place.

Speculative Execution Example.

We next present an example of speculative execution and also show the ROL operations. We use the example program from Figure 6.9a, and assume the same three-processor system. The execution is summarized in Figure 6.23. Recall that task D reads from object O which task C writes to, creating a RAW dependence between D and C (the dashed arrow in the Figure). Figure 6.23a shows the task invocations from the program at different time epochs and Figure 6.23b shows the execution along with the ROL operations. In this example we focus on the ROL operations and speculation, and not as much on the Token Protocol.

As before, the main program begins execution on a single processor, and the runtime sets up its schedulers and data structures on different processors. In epoch t1, task A is invoked. An entry for A (which subsequently becomes subtask A1 when B is invoked) is logged in the ROL (the head is to the left). Before A1 modifies any data, its modset is checkpointed and stored in the HB. The HB, which is not shown, mirrors the ROL in structure, and in the insert and retire operations. The HB stores a computation's checkpoint whereas the ROL holds an entry for the computation. HB may be logically combined with the ROL.

In epoch t2, when task B is invoked, subtask A1 has completed. Since it is at the head of the ROL, and hence non-speculative, it is retired and its entry is removed. A1's checkpoint from the HB is also removed. Any token requests it may have performed are removed from the corresponding Wait Lists. Next, entries for tasks B1 and A2 are logged in the ROL in the program order, and they are scheduled for execution.

In epoch t3 A2 completes. But since an older task, B1, has not yet retired, A2 cannot be retired. Task D and A3 are invoked, their entries are logged in the ROL, after A2. The runtime recognizes that D is a nested task, and because tasks preceding it in the ROL have not completed, its order in the program may not be determinable. Nonetheless, since resources are available, tasks A3 and D

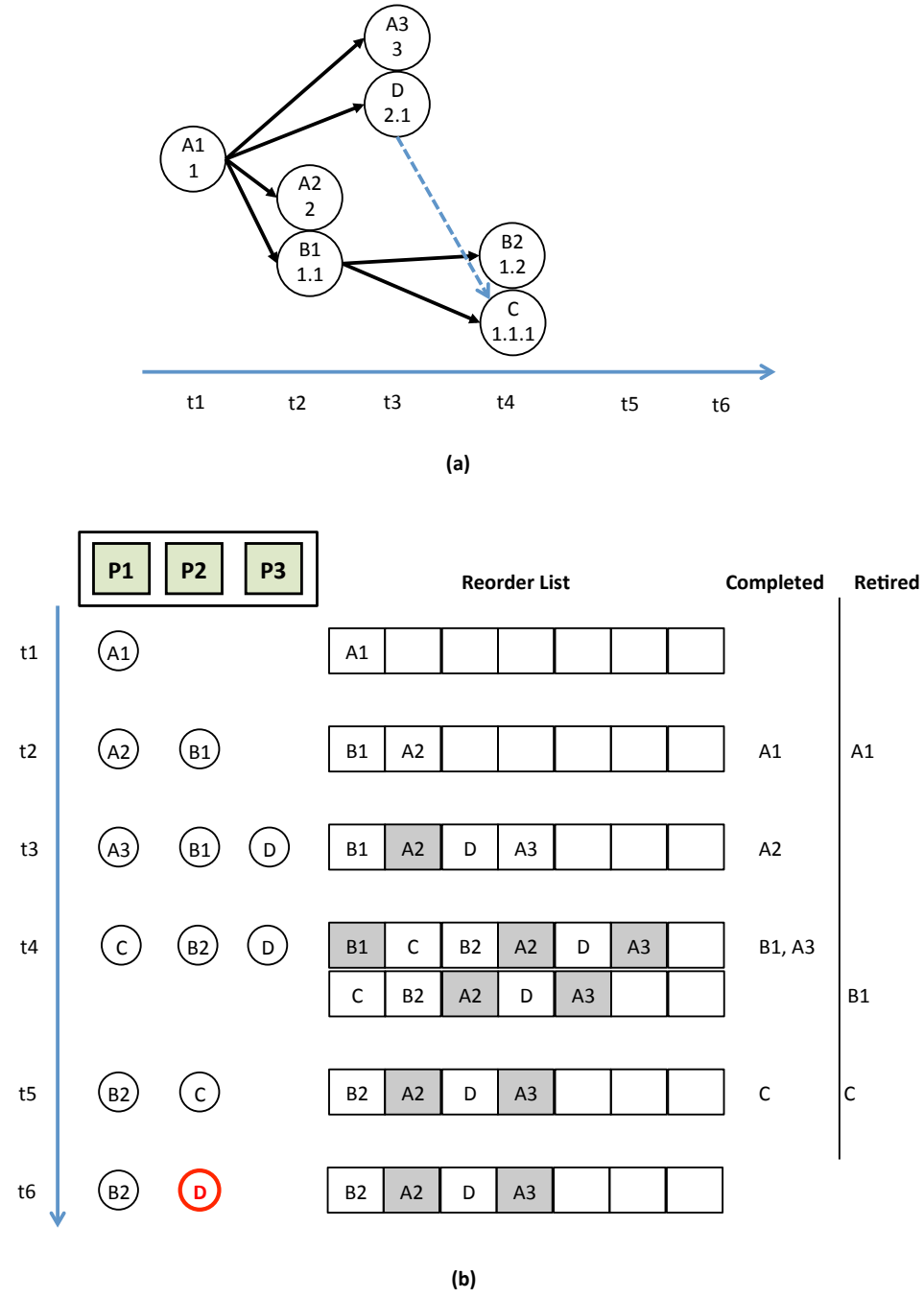


Figure 6.23: Speculative execution of the program in Figure 6.9a. (a) Task invocation order. (b) Execution schedule on three processors and corresponding Reorder List operations. Shaded entries in the Reorder List indicate completed tasks not yet retired.

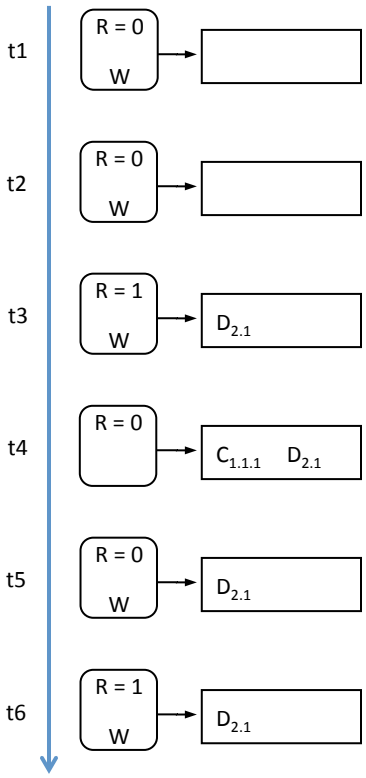


Figure 6.24: Token manager operations on object O. D acquires token speculatively, but is found to be misspeculated when the request for C arrives.

execute speculatively.

Before D executes, it acquires the read token to O. The token protocol operations, only for object O, are shown in Figure 6.24. An entry for task D, which includes its ID, is logged in O's Wait List, as shown in Figure 6.24, epoch t3.

In epoch t4, tasks B1 and A3 complete. The ROL changes are shown in two steps. Task C and subtask B2 are invoked, entries for them are logged in the ROL, after B1, but before A2, indicating their relative positions. Note that the hierarchical ID assists in this process. Next, B1 is retired and its entry is removed from the ROL (and any checkpointed modest from the HB, and token requests from the Wait Lists).

Before C executes, it tries to acquire the write token to O. When inserting an entry for C in the Wait List, the Token Manager discovers that the read token was granted to a computation with a younger ID and thus detects misspeculation. From the Wait List the Token Manager can ascertain that D is dependent on C, and informs the GRX about the misspeculation. The GRX aborts D (or could have waited until D completes). D's checkpoint from the HB is used to restore any state that it may have modified. Tokens acquired by D are returned. Next, tokens for task C are acquired, and C is submitted for execution.

Once task C completes, in epoch t5, it returns its tokens. The token to O is passed to task D. Since task C is at the head of the ROL, it is retired (Figure 6.23b, epoch t5)–its entry from the ROL, checkpoint from the HB, as well as its entry from object O (and from any other objects in its dataset) are removed from the corresponding Wait List (Figure 6.24, epoch t5). Next, task D is scheduled for execution since it now has all its tokens (Figure 6.23b, epoch t6).

Note that when misspeculation was detected, only the misspeculated task was aborted and needed to be restarted in the above example. It is also possible that misspeculated tasks have completed when the misspeculation is detected, or the misspeculated task has already invoked control-dependent tasks. For example, above when task C tries to acquire the token to O in epoch t4, the younger task D may have already completed and returned its token to O. Although now the token to O will be available for C to acquire, the Token Manager will detect the misspeculation because D's token request will still be in the Wait List. The Token Manager will report the misspeculation to GRX. GRX will deem D and any tasks transitively data- or control-dependent on D, to have misspeculated. GRX restores the state the misspeculated tasks may have modified by using their checkpoints from the HB. Token requests for the misspeculated tasks are removed from the respective Wait Lists. Entries for the control-dependent tasks are removed from the ROL. Finally, GRX submits C for execution, and once it completes, D and its dependent tasks are restarted in the dataflow fashion.

6.4.5 Summary

Parakram's Execution Manager uses a range of mechanisms, dataflow execution, speculation, globally-precise interrupts, task scheduling, and the like, to reap parallelism and support the APIs the Parakram programming model provides to programmers. While the Execution Manager presents an ordered view of the execution to the programmer, it also takes advantage of the order for its own operations, such as resource management. With linear and nested tasks, eager, JIT, and lazy datasets, eager dataset release, and barriers, Parakram admits arbitrary parallel algorithms.

Parakram does not limit itself to performing just computations. It makes for a more comprehensive solution, adequately addressing often neglected aspects of multiprocessor programming–I/O, exceptions, resource management, and efficiency. Furthermore, it forms the basis for enabling new capabilities. In one example of its broader utility, in Chapter 8 we apply it to recover from frequent exceptions. We have explored yet another application, to improve system efficiency, elsewhere [177, 178]. Executing a multiprocessor program efficiently requires regulating the exposed parallelism–increasing or decreasing the parallelism to match the available resources. Regulating the parallelism is easier in the ordered execution than in the nondeterministic execution because the ordered execution can ensure forward progress more easily.

7 Experimental Evaluation

The greatest value of a picture is when it forces us to notice what we never expected to see.

— John W. Tukey (1977)

Over the last three chapters we presented the Parakram approach and the design details of its prototype. Chapter 5 described the common patterns that arise in parallel algorithms, and Parakram APIs that enable them in ordered programs. Parakram realizes the parallelism patterns through a combination of the programming APIs and a supporting Execution Manager. In comparison to the established multithreaded programs, we had stipulated that: (i) Parakram must deliver the benefits of ordered programming, (ii) it must not compromise expressiveness or complicate programming in other ways, and (iii) and it must not compromise performance. In this chapter we evaluate whether the Parakram prototype meets these three criteria.

To evaluate Parakram, we developed programs listed in Table 5.10. They were selected from benchmark programs commonly used in parallel processing studies. They cover the range of parallelism patterns identified in Chapter 5, and stress the different aspects of the Parakram design. We partially evaluated Parakram's expressiveness in Chapter 5 by examining how the Parakram APIs can be used to express the different patterns using ordered programs. We complete the expressiveness evaluation in Section 7.1 by examining how the patterns are actually realized at run-time.

In Section 7.2 we evaluate Parakram's ability to exploit parallelism and deliver performance. To evaluate the performance, we experimented on commodity multiprocessor systems. We study two aspects of performance: speedups achieved by Parakram and the overheads it incurs due to its mechanisms.

Our study, using the wide range of algorithms, shows that Parakram can match multithreaded programs in expressing parallelism. It also matches multithreaded programs in performance in almost all but a few cases that use very small tasks or implement highly nondeterministic algorithms.

7.1 Expressiveness

Codes developed for the benchmarks using Parakram closely resembled the sequential versions intended to run on a uniprocessor. No threads were created and no synchronization primitives were used. No explicit work distribution or re-creation of the original execution sequence was required. Moreover, no pattern-specific algorithm structures were needed to exploit parallelism.

We evaluate Parakram's expressiveness with the help of the MAPLE design patterns (Section 5.1). Recall that MAPLE comprises four design spaces, the Algorithm Structure, the Program Structure, Execution Structure, and Concurrency Tools. Each design space comprises multiple design sub-spaces. The design spaces and sub-spaces are summarized in Table 7.1. We analyze how Parakram supports the patterns that arise in three of the four design spaces: the Program Structure, Execution Structure, and Concurrency Tools. Recall that Parakram does not provide any special support for the Algorithm Structure, because none is needed beyond using a capable enough programming language, like C++. The following description is organized along the lines of the design sub-spaces.

Data Structure Design. Choice of data structures, static allocation of data, and dynamic allocation of data are the patterns that arise in Data Structure Design.

Parakram relies on object-oriented designs and treats individual objects as atomic unit of data, but does not restrict the programmer from using any type of data structures. Scalars, block-allocated arrays, pointer-based structures, etc., are all permissible. For example, Swaptions uses a 1D array, Labyrinth and Cholesky decomposition use 2D arrays, RE and Genome use hash tables, DeMR implemented graphs uses arrays and linked lists, and Barnes-Hut uses a tree structure.

Parakram interfaces with the program's data structures through the datasets. Pointers to objects from any of these data structures are easily included in the read, write, and mod sets, which

Design Space	Design Sub-space	
Algorithm Structure	Solution Decomposition Data Organization Dependence Analysis	
Program Structure	Data Structure Design Dataset Computation UE Composition	UE: unit of execution, i.e., task
Execution Structure	UE Discovery UE Scheduling UE Coordination	
Concurrency Tools	Context Abstraction Synchronization	

Table 7.1: MAPLE: Classification of common patterns in parallel programs. The first column lists the design spaces and the second column lists the sub-spaces in each design space.

Parakram dereferences to identify the dynamic instances of the objects being accessed by a task.

Parakram also supports both statically and dynamically allocated data structures. It provides its own memory allocator and supports either type of data structures for basic dataflow as well as speculative execution. In particular, the allocator performs better than standard allocators, e.g., the one available in the GNU C++ runtime library (commonly used by the gcc compiler), and enables rollback and re-execution to support speculative execution.

Dataset Computation. Data-dependent datasets and data-independent datasets are the two patterns that arise in Dataset Computation.

Parakram supports data-dependent and data-independent datasets through the combination of the `pk_task()`, `pk_task_uc()`, `pk_read()`, `pk_write()`, `pk_declare()`, and `pk_declare_ia()` APIs. Speculative execution is used to exploit parallelism when the datasets are data-dependent.

Cholesky decomposition, Sparse LU, CGM, Histogram, Pbzip2, Reverse Index, Swaptions, Mergesort, Barnes-Hut, and Black-Scholes use data-independent datasets. DeMR, Genome, Labyrinth, BFS, RE, and Vacation use data-dependent datasets. DeMR and RE use `pk_write()`, Genome

uses `pk_declare_ia()`, Labyrinth and BFS use `pk_declare()`, and Vacation uses `pk_read()` and `pk_write()`.

A combination of the patterns in Dataset Computation and Data Structure Design can arise in algorithms. For example, tasks in Delaunay mesh refinement operate on a graph, which is implemented using arrays and linked lists. After the initial graph is constructed, the algorithm dynamically allocates and deallocates objects to add and delete nodes from the graph. Task datasets are data-dependent. Parakram supports the combination of these patterns with the help of speculation.

UE Composition. Parakram provides two simple interfaces, `pk_task()` and `pk_task_uc()`, to express units of computations as tasks. All programs, except Labyrinth, were developed using the simple `pk_task()` calls.

Labyrinth poses a unique challenge to parallelization. Labyrinth reads a large part of a grid structure to route paths between two points. Whereas each task may read thousands of points on the grid, it updates only tens of points to route the path. All of these points are part of the dataset, but the large read set can lead to considerably large overheads (irrespective of the programming model used). Further, the route a task computes changes based on the points already occupied on the grid. Hence, a task concurrently updating the grid can influence another's dataset. We tackled Labyrinth as follows, using the Parakram APIs. First, we used `pk_task_uc()` to ensure that the grid points are not updated speculatively. Tasks are allowed to read the grid and create a local copy of the routed path. The read and write sets are then declared lazily. Parakram processes the dataset at commit, and the task had not misspeculated, the control is returned to the task. (If the task had misspeculated, it is re-executed.) The task can now commit the path to the architectural state. `pk_task_uc()` is supported with the help of speculation. Second, to reduce the read set size and reduce the token processing overheads, the read set includes only the points that are on the actual path, and not all the read points. However, to ensure correctness and that the order is maintained, after declaring the dataset, each task now checks whether any points were updated since it read them last. If any has changed, the task throws an exception using `pk_except()`, requesting the task

to be re-executed. If no point has changed, the task can proceed and update the global state from the local state, and complete successfully.

UE Discovery. Tasks in multithreaded programs can be invoked in a linear sequence or by nesting in other tasks. Parakram supports both types. It uses speculation to support nested tasks, while still maintaining ordered execution. Recursive CD, Mergesort, recursive Sparse LU, and DeMR use nested tasks. Other programs use linear tasks.

UE Scheduling. Multithreaded programs permit explicit scheduling of UEs (i.e., tasks). Tasks may be scheduled for execution as soon as they are invoked, or after a delay. In general, multithreaded programs explicitly implement scheduling patterns such as pipelining, master-worker, work queues, etc., although newer programming models like TBB and Cilk provide declarative constructs to achieve similar effects.

Parakram provides mechanisms to invoke tasks, and programmers can control when the tasks are invoked, but Parakram automates their scheduling. Parakram's internal Task Pool and dataflow scheduling, combined with the load-balancing scheduler, naturally admit patterns such as pipeline parallelism, master-worker, and work queues. For example, recall that the Pthreads design of Pbzip2 explicitly implements the pipeline parallelism, and TBB directly provides the pipeline parallelism construct. In Parakram, the code is structured like the original sequential loop, and the dataflow processing achieves the pipeline effect without user intervention. Parakram serializes dependent computations and schedules their execution when they are ready, thus automatically creating a pipeline of computations, e.g., by serializing a dependent file write after the compression, but overlapping it with compression of another block. Further, lazy task creation and work-stealing ensure that work is sought and created only when a thread is idle, thus efficiently utilizing resources.

Above capabilities not withstanding, ordered programmers are free to implement scheduling patterns explicitly, and in some cases may prefer to trade off performance and complexity. For example, DeMR code in Figure 5.13 (lines 17, 22) implements a work queue, and controls the task

invocation. The DeMR code may implement a single work queue, or a queue of work queues for better performance. In event-based coordination, e.g., to simulate a hardware circuit, the programmer may implement multiple work queues from which tasks are invoked for execution under user control.

UE Coordination. Coordinating UE execution around shared data accesses is a critical aspect of multithreaded programs. There is no one-to-one correspondence between Parakram APIs and multithreaded programs in this regard.

By automatically ordering computations at the task boundaries and ensuring data-race freedom, Parakram eliminates the need to explicitly order UEs, synchronize data accesses, or use critical regions. However, this ordering is applied at a coarser, task granularity. Furthermore, the ordering takes place *eagerly*, i.e., at the start of a task. In contrast, multithreaded programs permit arbitrarily fine-grained UE ordering.

Parakram permits programmers to start executing a task and delay the dataset declaration by using JIT or lazy datasets. It also allows tasks to release objects in their dataset eagerly, indicating they will no longer access the data, via `pk_release()`. If not released eagerly, objects are implicitly released when the task completes. Eager release allows other dependent tasks to proceed sooner, instead of waiting until the task finishes. JIT and lazy datasets, combined with eager release narrow the region of code which gets serialized, effectively permitting finer grain coordination. But permitting tasks to execute before knowing their datasets hampers dataflow scheduling. Parakram relies on speculation to provide this support.

DeMR, Genome, RE, and Vacation use JIT datasets. Labyrinth and BFS use lazy datasets. We also used eager release in Genome and Vacation but, in general, found no pressing need for it in other programs.

No explicit coordination was needed to perform I/O. Reading input from a file is straightforward. Dataflow scheduling ensures that the writes happen in the correct order, e.g., in Pbzip2, Swaptions, and Labyrinth, with no user coordination. (We did not explore asynchronous parallel I/O, typically

performed in concurrent programs, in this work.)

Parakram also provides the barrier APIs, `pk_serial()` and `pk_barrier()`, for control-flow coordination. Barnes-Hut, CGM, and Mergesort use `pk_barrier()`.

Context Abstraction and Synchronization. Parakram provides no explicit notion of threads or execution context to programmers; programmers may invoke tasks using `pk_task()` and `pk_task_uc()`. Since Parakram automates the parallelization, other explicit concurrency mechanisms are neither needed nor provided.

Using Parakram APIs and its execution model, we were able to implement all of the parallelism patterns we encountered in the benchmark programs that we studied.

Result 1. *Although the eventual code structures are different from their multithreaded counterparts, Parakram ordered programs could express the same types of parallelism in all cases. Parakram's APIs were adequate to express tasks, perform I/O, declare, and release the datasets.*

7.1.1 Programmability

Although in this dissertation we have not quantified Parakram's impact on programmer productivity, especially when coding for performance, we rely on arguments to convince the reader of its positive impact. In Chapter 3 we presented the properties of ordered execution, and how they simplify programming. In Chapter 4 we described the simpler process of developing ordered programs, and compared ordered code examples with the more complex multithreaded examples in Chapter 5. Based on these arguments, examples, and our experience developing about twenty programs during this work, we conclude that ordered programming is easier than Multithreading.

Programs (1)	**Seq. (s)** (2)	**Multi-thread.** (3)	**Parallel-ism** (4)	**Task size** (5)	**Scale.** (6)	**Chkpt. size** (7)	**UE type** (8)	**DS** (9)	**Decl. type** (10)	**Input** (11)
Barnes-Hut	387.23	Pthreads	Regular	Med.	High	Med.	Lin.	DI	Eager	100K bodies, 75 steps
Black-Scholes	214.73	Pthreads	Regular	Med.	Med.	Large	Lin.	DI	Eager	10M options
Iterative CD	15.11	OpenMP	Irregular	Large	High	Large	Lin.	DI	Eager	4K×4K matrix
CGM	24.12	OpenMP	Regular	Med.	High	Med.	Lin.	DI	Eager	4K×4K matrix
Histogram	1.6	Pthreads	Regular	Small	High	Med.	Lin.	DI	Eager	1.4GB bitmap file
Pbzip2	170.38	Pthreads	Irregular	Med.	Med.	Small	Lin.	DI	Eager	673MB file
I. Sparse LU	6.95	OpenMP	Irregular	Large	High	Large	Lin.	DI	Eager	4K×4K matrix, 0.108 density
Swaptions	421.7	Pthreads	Regular	Large	High	Small	Lin.	DI	Eager	128 swaptions, 1M simulations
Recursive CD	15.11	Cilk	Irregular	Large	High	Large	Nest	DI	Eager	4K×4K matrix
DeMR	7.63	Pthreads	Irregular	Small	High	Small	Nest	DD	JIT	250K triangles
Genome	7.13	TL2 TM	Irregular	Small	Low	Small	Lin.	DD	JIT	16K nucleotides, 64 nucleotide segments 16M segments
Labyrinth	43.62	TL2 TM	Irregular	Small	Low	Med.	Lin.	DD	Lazy	512×512×7 grid, 512 paths
Mergesort	12.49	Cilk	Regular	Med.	High	Small	Nest	DI	Eager	1M elements
BFS	3.41	Pthreads	Irregular	Tiny	Low	Small	Lin.	DD	Lazy	10M nodes, degree 5
RE	15.77	Pthreads	Regular	Med.	Low	Small	Lin.	DD	JIT	1.2M packets
R. Sparse LU	6.95	OpenMP	Irregular	Large	High	Large	Nest	DI	Eager	4K×4K matrix, 0.108 density

Table 7.2: Programs and their relative characteristics. CGM = conjugate gradient method, Seq.= sequential time, Scale. = scalability, Chkpt. = checkpoint, Lin. = linear, DS = dataset, DI/DD = data-independent/dependent, Decl. = declaration.

Specification	Intel Core i7-965	AMD Opeteron 8350	Intel Xeon E5-2420	AMD Opetron 8356
Sockets	1	4	2	8
CPUs per socket	4	4	6	4
Threads per CPU	2	1	2	1
Total contexts	8	16	24	32
Clock	3.2GHz	2.0GHz	1.9GHz	2.3GHz
L1 I$, D$ per CPU	32KB	64KB	32KB	64KB
L2$ per CPU	256KB	512KB	256KB	512KB
L3$ (shared)	8MB	2MB	15MB	80MB
Linux kernel	2.6.32	2.6.25	2.6.32	2.6.25

Table 7.3: Machine configuration used for experiments.

7.2 Performance

To evaluate Parakram's performance we study the speedups it achieves on the various algorithms. Achieved speedups are influenced by multiple factors. Algorithm characteristics influence speedups. For example, small task sizes can limit scalability. In a managed runtime system like Parakram, the runtime overheads also impact speedups. The algorithm characteristics in turn can impact the overheads. For example, small tasks can exacerbate the overheads, which can also limit the scalability. Hence, in addition to measuring speedups, we also characterize Parakram's overheads.

Table 7.2 lists the programs for which we measured the speedups. Programs with different algorithm characteristics were chosen. We ensured that all design patterns are tested. The table also summarizes the program characteristics and patterns: parallelism type (column 4), task size (column 5), scalability (column 6), mod set checkpoint size (column 7), task types (column 8), dataset type (column 9) and dataset declaration type (column 10). The overheads are characterized using micro-benchmarks, which are described later.

We experimented on four machines of ubiquitous sizes, but different organizations: 4-core and 12-core, 2-way hyperthreaded Intel processor systems, and 16-core and 32-core AMD processor systems. The machine specifications are provided in Table 7.3.

We compare speedups obtained by Parakram and the multithreaded equivalent, over the original optimized sequential version. The sequential execution times are listed in column 2 of Table 7.2. The multithreaded variant used are listed in column 3. For comparison with multithreaded programs we chose the best available multithreaded implementation when multithreaded implementations were available. We were unable to locate satisfactory multithreaded designs for DeMR, BFS, and RE. We developed Pthreads parallel code for them ourselves. Galois-like [142] designs were implemented for DeMR and BFS, and a straightforward design for RE.

Besides exploiting parallelism, Parakram strives for efficient execution by balancing the system load as the program runs. For a fair comparison it was necessary that a similar optimization be applied to multithreaded programs, developed using OpenMP, Pthreads, Cilk, and TL2 TM in our study. OpenMP and Cilk runtimes do provide in-built load-balancing capabilities. OpenMP provides in-built scheduling directives that the programmer can choose; we used the directive that yielded the best speedups. Cilk automatically applied lazy task creation, continuation scheduling, and dynamic task-stealing, and provides no direct control to the programmer. Pthreads provides no such support, and leaves it to the programmer to implement it. Among the Pbzip2 programs we tested for speedups, only Pbzip2 design incorporated load balancing. In the other programs, we incorporated load-balancing by implementing virtual threads [178]. A transparent shim layer is introduced between the program and the Pthreads runtime library. Unbeknownst to the programmer, the shim layer logically divides the program threads into sub-threads at run-time, and a load balancing scheduler is used to execute the sub-threads. This scheme improved speedups in several cases, especially when task sizes are large, e.g., virtual threads improved Swaptions speedup by $\sim 2\times$. It is unclear whether TL2 TM [54], which is implemented using Pthreads, balances system load; we did not alter TL2. The two TL2 programs in our study use very small tasks, making load imbalance a non-critical issue, and hence it is unlikely that they affect the overall results of our study.

Perhaps unsurprisingly, Parakram programs in some cases outperform the multithreaded variants, in many they give similar speedups, and in some rare cases they underperform. However,

the reasons they outperform or underperform make for an interesting study. In general, the multithreaded and Parakram programs showed similar relative speedup trends, but for the absolute values. We present speedups on the 24-context Intel Xeon machine (specifications listed in Table 7.3). Programs were compiled using gcc 4.8.2, with the `-O3` and `march=corei7-avx` options. For all programs large inputs were used to measure the speedups. The input size details are listed in Column 11 of Table 7.2. Results are averages over ten runs. We present the experimental results in three groups. The first presents Parakram's speedups for *eager programs*–programs that use linear tasks and eager datasets. The second presents results for *non-eager programs*–programs that use nested tasks, or JIT datasets, or lazy datasets. Within the first two groups we examine the cases when Parakram matches or outperforms the multithreaded programs. The third group examines Parakram's performance limitations.

7.2.1 Eager Programs.

Figure 7.4 shows Parakram speedups for programs that used linear tasks and eager datasets. They do not benefit from speculation, but incur the overheads nonetheless. Speedups are plotted on the Y axis and context-counts on the X. The O.Parakram lines show the speedup for ordered execution. The D.Parakram lines show speedups for deterministic execution, i.e., when the ROL and precise-exception capabilities are switched off. For the multithreaded variants, we label the lines in the graphs with the corresponding Multithreading API.

In the first set of results for eager programs O.Parakram matched the multithreaded speedups, as shown in Figure 7.4. When algorithms are dominated by simple fork-join parallelism, e.g., Barnes-Hut, Histogram, Swaptions, and CGM, Parakram has the same opportunities as multithreaded methods to exploit the parallelism, and hence does equally well. But importantly, Parakram performs ordered execution.

When the parallelism is irregular, Parakram can exploit latent parallelism, and hence can do better, e.g., in Pbzip2, Black-Scholes, Iterative CD, and Iterative Sparse LU, as shown in Figure 7.5. For example, on Iterative CD it outperformed Multithreaded OpenMP by 3.35$\times$ (Figure 7.5).

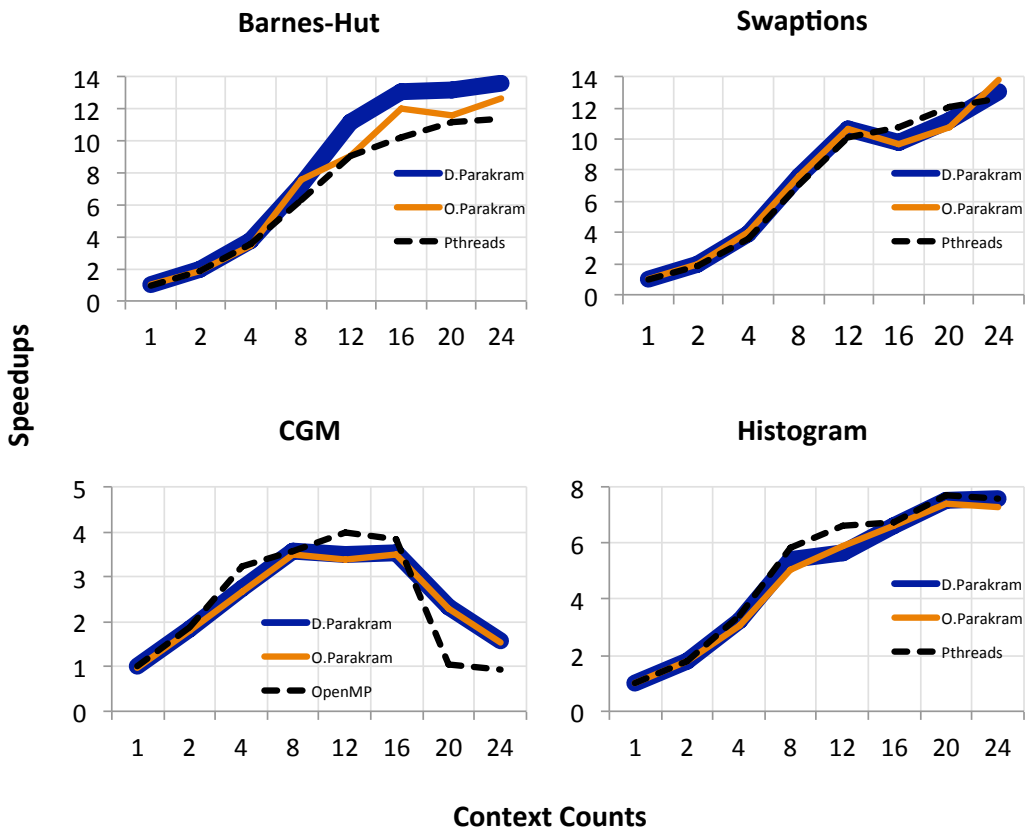


Figure 7.4: Speedups for eager programs, Barnes-Hut, Swaptions, CGM, and Histogram, in which Parakram matches Multithreading. deterministic (no ROL tracking and checkpointing). O.Parakram = ordered execution.

Performance Benefits of Order.

Consider the iterative CD code in Figure 5.3. Figure 7.6a shows the dataflow graph of its first five tasks. Next to each task (labeled node) is the epoch in which the task is invoked. In the parallel execution, after task 1 completes, task 2 can execute in epoch 2, but not tasks 5 and 6 due to their dependences. However, task 3, which is independent, can, resources permitting, but only if it can be discovered in epoch 2.

In the multithreaded paradigm, two concurrently executing tasks must be independent. Hence it is the programmer's responsibility to expose the parallelism at the right time, i.e., invoke task 3 in epoch 2. If not exposed, an unassisted system cannot reach it, even speculatively, because

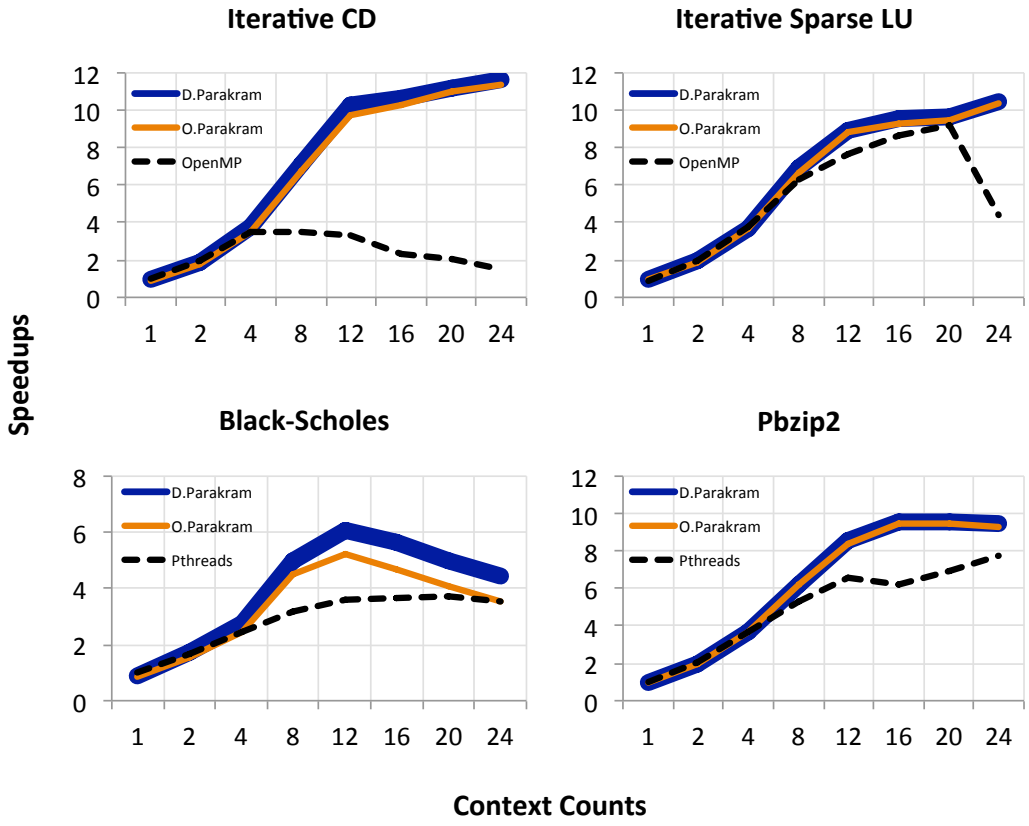


Figure 7.5: Speedups for eager programs, Iterative CD, Iterative Sparse LU, Black-Scholes, and Pbzip2, in which Parakram outperforms Multithreading. D.Parakram = deterministic (no ROL tracking and checkpointing). O.Parakram = ordered execution.

the system cannot compute the dependences. Even if the system, e.g., Transactional Memory or Galois [142], can observe and identify the data a task accesses, the task order, which is the other critical information needed to compute the dependence, is missing. Moreover, the multithreaded programming abstraction permits arbitrary placement of computations in the program's text. Hence even inspecting the program's text to discover the order is futile.

In contrast, Parakram can compute the dependences using the order and the dataset information. It builds the dependence graph and searches it dynamically, setting aside dependent tasks, seeking and executing tasks that are independent or whose dependences have resolved (Figure 7.6b). CD presents many such *latent* parallelism opportunities, which an ordered approach like Parakram can

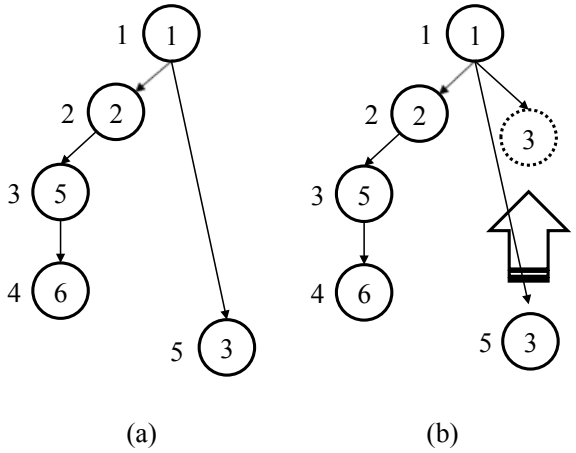


Figure 7.6: Parakram 's ability to exploit latent parallelism. (a) Task dependence graph of the first five CD tasks (nodes). Label next to a node is the epoch in which the task is invoked. (b) Preferred execution schedule.

exploit, as is also observed by Perez et al. [141]. Similar opportunities arise in Iterative Sparse LU due to irregular dependences, and in Pbzip2 and Black-Scholes when they write computed results to a file.

Multithreading does give the freedom to structure the code and expose any parallelism, which can indeed be applied to CD. But when the dependences are complex, as they are in CD, or when the dependences can only be known at run-time, programmers often take a conservative approach that is relatively easier to understand and reason about. In bargain, they trade-off performance, as the OpenMP CD code does, which they need not when using Parakram.

For eager programs, the programmer may also instruct Parakram to switch off dependence speculation (since it does not benefit eager tasks), defaulting to deterministic execution. In that case the D.Parakram lines show the speedups, which are better than O.Parakram due to reduced overheads, even if slightly (more later).

Result 2. *When algorithms have complex dependences, like Cholesky decomposition, exploiting parallelism may be more natural in the ordered method, whereas the multithreaded method may require considerable efforts.*

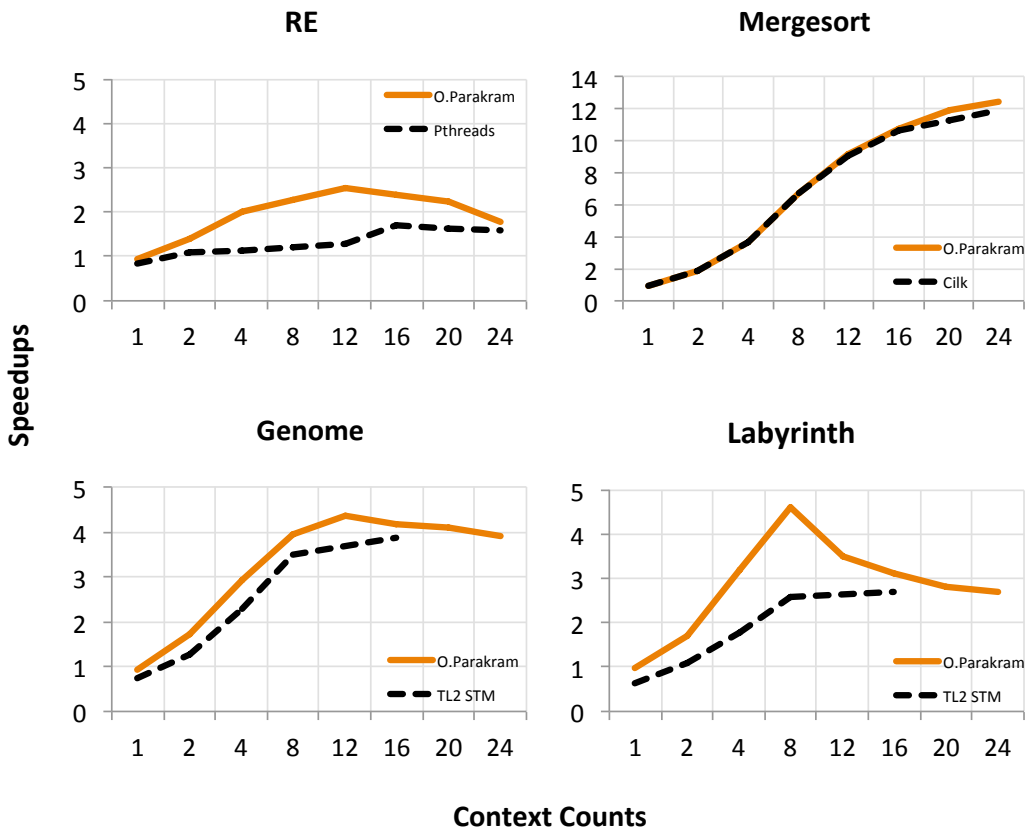


Figure 7.7: Parakram's speculative execution performance on RE, Mergesort, Genome, and Labyrinth. O.Parakram = ordered Parakram.

7.2.2 Non-eager Programs.

Parakram switches to speculative execution for programs that use nested tasks or non-eager datasets. Since speculation is always on, D.Parakram is inapplicable here. In general, non-eager programs exhibit irregular parallelism. Parakram matches or outperforms the multithreaded methods, except for DeMR and BFS, which we discuss later.

Figure 7.7 shows the speedups achieved for Genome, Labyrinth, RE, and Mergesort. Graphs show Multithreading and O.Parakram speedups (TL2 supports only power-of-two threads, hence the truncated TL2 STM lines for Genome and Labyrinth). On Genome, Labyrinth, Mergesort, and RE, Parakram matches or slightly outperforms the multithreaded variants, while providing ordered

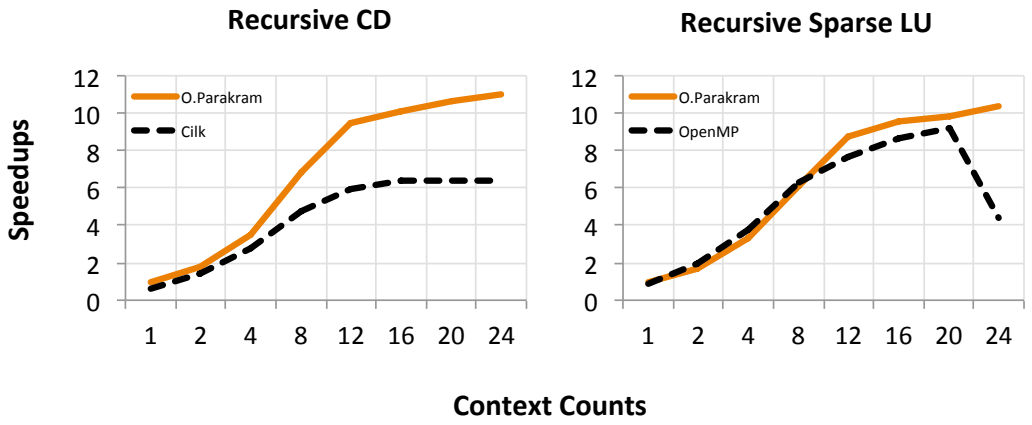


Figure 7.8: Parakram's speculative execution performance for Recursive CD and Recursive Sparse LU. Multithreaded variants are listed in Table 5.10.

execution.

Genome, Labyrinth, and RE algorithms are nondeterministic, i.e., the task order is immaterial for correctness. The artificial constraint of order in these cases does not impede the Parakram performance, as the Figure 7.7 shows. In fact, speculation allows Parakram to discover parallelism more aggressively in these cases.

The Mergesort algorithm is dominated by nested fork-join parallelism–parallel computations within a parallel region do not perform conflicting accesses. Hence, Parakram's speculative execution does not unearth more parallelism than the Cilk design. However, since the design is recursive, and hence used nested tasks, Parakram applies speculation.

Figure 7.8 shows the speedups for Recursive CD and Recursive Sparse LU, on which Parakram outperforms Multithreading. The far more complex Cilk Recursive CD [123] (the structure of which is shown in Listing 5.11) outperforms the OpenMP Iterative CD design since the Cilk design exposes more parallelism (compare the OpenMP line for Iterative CD in Figure 7.5 with the Cilk line for Recursive CD in Figure 7.8). We implemented the Cilk recursive algorithm in Parakram. Parakram yielded similar speedup (not shown). However, on the naive recursive design of Figure 5.10 Parakram outperformed the Cilk design by 60% (Figure 7.8), for the same reasons that the iterative

Parakram CD outperformed the OpenMP implementation. Cilk requires that spawned tasks do not perform conflicting accesses (unless the programmer is willing to explicitly coordinate the execution). This limits the exposed parallelism in the recursive CD design. Parakram imposes no such constraint.

Speculation is also applied to Recursive Sparse LU. Speculation benefits Recursive Sparse LU for the same reasons as it benefits recursive CD.

Misspeculations.

Algorithm properties and system size influence Parakram's misspeculations. In general, data-dependent datasets cause more misspeculations. Larger systems present more resources to speculate, and hence also cause more misspeculations. However, Parakram limits the total misspeculations by performing ordered, dataflow recovery. On average, Parakram misspeculated 3.88% of the tasks in Recursive CD, 30.42% in DeMR, 0.45% in Genome, 13.3% in Labyrinth, 1.3% in RE, and almost none in Recursive Sparse LU and BFS, on 24 contexts. Although Mergesort uses nested tasks, they do not perform conflicting accesses and hence do not misspeculate.

Parakram overcomes the artificial order in the nondeterministic Genome, Labyrinth, Mergesort, and RE algorithms by using dataflow and speculation. Despite the misspeculations, dependence speculation yields net performance benefits.

If speculation were unavailable, non-eager problems may still be implemented as ordered programs, e.g., using a Deterministic Galois-like design [135], which we also implemented in Parakram. Briefly, in such a scheme, the execution proceeds in rounds. In each round the runtime picks a set of tasks to execute in parallel. Tasks execute in two phases. In the first phase, the runtime launches a set of parallel tasks for execution. The tasks declare their datasets and pause. The runtime then acquires the tokens for each task, in the program order. Only tasks that could acquire all their tokens proceed to the next phase and complete. Tasks that couldn't acquire all the tokens re-execute from the start, in the next round along with additional tasks. Essentially the irregular parallelism is forced into fork-join type parallelism, which limits the exploitable parallelism. Speedups achieved

#	Overheads		8x core i7	16x AMD 8350	24x Xeon	32x AMD 8356
1	ROL enqueue (Figure 6.15, line 2)		100	500	100	1000
2	Token Acquire (Figure 6.16, line 22-23)		100	200	100	300
3	Token Release (Figure 6.16, line 30)		100	200	100	300
4	Token Enqueue (Figure 6.16, line 15)		575	750	575	1400
5	Token Pass (Figure 6.16, lines 31-35)	1 object	500	600	500	1000
6		>1 object	100	150	100	300
7	Prelude (Figure 6.15, ❶)	Base	500	1000	500	2000
8		Token Acquire	250	300	250	500
9		Token Enqueue	600	750	600	1500
10	Postlude (Figure 6.15, ❸)	Base	250	300	250	400
11		Token Release	300	300	300	450
12		Token pass(1)	650	700	650	1500
13		Token pass (>1)	100	200	100	200
14	Granularity	Base	3000	5000	3000	10000
15		>1 object	200	200	200	500

Table 7.9: Characterization of average Parakram overheads in cycles.

by this scheme were lower (not shown), e.g., only $\sim 1.4\times$ for Genome, instead of $\sim 4\times$ at 24 context.

Result 3. *An ordered approach need not compromise performance for a wide range of algorithms. Dataflow and dependence speculation can overcome the artificial ordering constraints.*

7.2.3 Performance Limitations of Order.

Above results not withstanding, Parakram performance can suffer due to two reasons: overheads and the ordering constraint.

Overheads.

Being a software model, the Parakram prototype has multiple sources of storage and computational overheads. The runtime's interface and mechanics add indirect calls, manage tokens and wait lists, manage the ROL, perform checkpoints, and shelve/schedule computations. These operations are often concurrent with the actual execution. For large enough tasks, these overheads are relatively small. The ROL overhead is proportional to the total tasks, and ranged from 1% to 3%. Checkpointing overheads ranged from 1% (Pbzip2) to 23.6% (Black-Scholes). The collective ROL and checkpoint overheads are evident in the difference between O.Parakram and D.Parakram lines in Figure 7.4. On average (harmonic) the combined overheads were 6.43% relative to D.Parakram.

We used micro-benchmarks to characterize the sources of overheads. Table 7.9 summarizes them. In general, overheads are proportional to total tasks and their dataset sizes. Hence we attribute overheads to each task, as a sum of: (i) a minimum cost to "manage" it (e.g., to process the ROL) and (ii) the cost to manage its dataset (e.g., to build the dependence graph). Depending on the machine, the management cost ranged from a few 100 cycles to a few 1000 cycles (row 1, Table 7.9). An additional cost of 100 to 500 cycles is incurred for each object in the dataset. As a result, Parakram requires that the task size be a minimum of 3000 to 10000 cycles, with additional 200 to 500 cycles of work for each dataset object, to achieve speedups. These overheads are specific to the current design, and may be optimized.

We instrumented the runtime to measure the overheads and frequency of events in the Prelude and Postlude phases, and the execution time of tasks, using six micro-benchmarks (described below). For fine-precision analysis, the overheads were measured in clock cycles. They were obtained by using the `rdtsc` instruction to read the x86 Time Stamp Counter. To ensure accuracy we first pinned the thread to the processor, and flushed the pipeline before and after `rdtsc`.

First, we used a microbenchmark to invoke dataflow tasks with appropriate datasets and created various conditions leading to the different overheads. The overheads are machine organization-dependent. Rounded off average measurements made on the four machines are listed in Table 7.9. We discuss the results below using the 24-context Intel Xeon data and refer to the rows in Table 7.9.

Overheads are the same for both read and write tokens.

To study the bare bones token acquisition (Figure 6.16, lines 22-23) and release (Figure 6.16, line 30) overheads, we created datasets such that the tasks in the micro-benchmark were independent. We varied the number of objects in the dataset from 1 to 10 and measured the overheads for 10,000 such tasks. On an average it took 100 cycles to acquire (Table 7.9, row 2) or release (row 3) a token. To study the token enqueue overhead, we created a pair of tasks with the same write sets. Thus the second task was dependent on the first and every token request it made was enqueued in a wait list. We invoked 10,000 such pairs, varied the write set size from 1 to 10, and measured the overheads for the second task. The enqueue logs the token request in the wait list (Figure 6.16, line 15). It allocates a data structure to hold the requester information and invokes race-free functions to log the request in the wait list, a concurrent queue [8, 91]. Hence enqueue incurs a higher overhead of 575 cycles (row 4).

To study the token passing overheads, we created a set of tasks that consumed (read) data produced (written) by a precedent task. Thus upon completion, the producing task passes tokens to the consumers. By varying the number of consuming tasks we simulated token passing to up to 10 tasks, and measured the overheads in the producing task for 10,000 such sets. Token passing first releases the token, traverses the wait list to pass it to waiting requesters, possibly enqueues ready requesters in the work deque (Figure 6.16, lines 31-35), and deallocates data structures. Passing a token too involves operations on a concurrent queue and possibly the work deque. It incurs an overhead of 500 cycles (row 5) for the first requester. In the current implementation, once the wait list traversal begins, concurrent queue processing simplifies, and hence only 100 cycles (row 6) are needed to pass tokens to every subsequent requester.

To characterize the total overheads of Prelude (Figure 6.15 ❶) and Postlude (Figure 6.15 ❸) phases, we used the same three set ups as before but measured the overall costs. The total overheads can be viewed as comprising of: (i) a base cost that is always incurred, and (ii) an additional cost proportional to the dataset size. The prelude phase performs allocation and population of data structures, checks for duplicates in the data set and processes tokens. The total base overhead

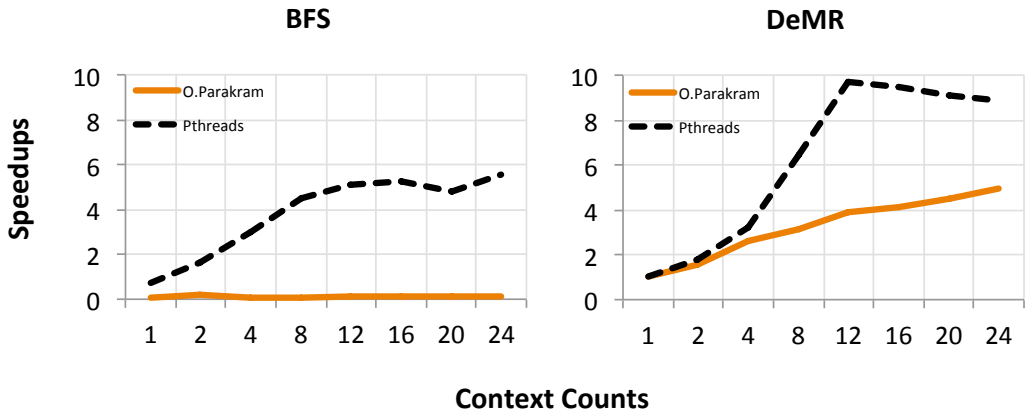


Figure 7.10: Parakram's speculative execution performance for BFS and DeMR. Multithreaded variants are listed in Table 5.10.

to submit a function is 500 cycles (row 7), and thereafter 250 cycles/object for an acquired token (row 8), or 600 cycles/object for an enqueued request (row 9). Postlude releases tokens, deallocates structures and returns to the runtime scheduler. It requires 250 cycles in the least (row 10), and 300 cycles/object to return a token (row 11) thereafter. If a token is returned and passed, it consumes 650 cycles (row 12) for the first requester, and 100 cycles for every subsequent requester (row 13).

Next we assessed the minimum task granularity needed to achieve speedups with Parakram. We invoked 10,000 tasks from the microbenchmark and increased the total number of dynamic instructions in them until the parallel execution achieved lower execution time than the sequential version. We also varied the data set sizes from 0 to 10. With empty data sets, tasks at least 3000 cycles long are profitable to parallelize (row 14). Their size needed to grow by about 200 cycles for every object added in the data set (row 15), to be profitable. The granularity results are similar to other task-based models [151].

Parakram can switch to deterministic execution (when a program comprises only eager tasks). In that case the ROL and checkpointing overheads are saved.

Given the above overheads, Parakram fails to speed up BFS (Figure 7.10) since the average BFS task size is ~1000 cycles, smaller than the needed minimum of 3000 cycles.

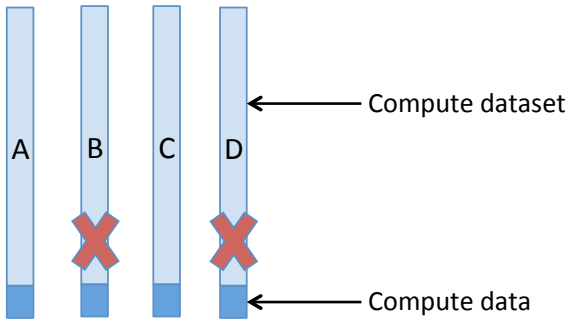


Figure 7.11: Idealized execution of DeMR.

Parakram also incurs memory overheads for checkpoints and its own data structures. Algorithm characteristics and the machine size influence these overheads. Recursive CD incurred the most, 4MB (~5% of its own memory use), on the 32-core system.

Result 4. *Beyond a minimum task size, overheads of ordered execution are reasonable relative to both, the multithreaded and deterministic execution.*

Ordering Constraint.

Although the overheads arising from mechanisms can hamper the speedups Parakram can achieve, they can be overcome, at least to a certain degree, with a better design, and perhaps even with hardware support. However, algorithms with a combination of certain characteristics pose a more fundamental challenge to the ordered method. Following a strict order in such algorithms may limit the parallelism Parakram can exploit in comparison to an multithreaded method.

Consider DeMR. Parakram achieves a speedup of 5.78, compared to 9.75 by the multithreaded variant, as the graph in Figure 7.10 shows. DeMR is *highly nondeterministic* [104]. The algorithm is nondeterministic; any task order produces an acceptable, albeit different, final solution. Furthermore, datasets are data-dependent (and hence highly nondeterministic). Moreover, tasks spend much time computing the datasets. DeMR is also highly parallel, but tasks can perform many conflicting accesses, which need to be avoided for correctness.

Consider DeMR's idealized parallel execution of four tasks in Figure 7.11. Say tasks B and D conflict, but A and C do not, either due to the execution schedule or because they do not perform conflicting accesses. The multithreaded method would deem C as completed. But an ordered execution cannot, because C's data-dependent dataset may be influenced by the data yet to be computed by B, which precedes it in the order. B will re-execute, and if C conflicts with it, C will be re-executed, leading to **order-induced** loss of parallelism (OLP). OLP is the parallelism lost due to order, and not true dependence.

Were the datasets declared eagerly, OLP could be avoided. In the above example, Parakram would have shelved C before it had begun executing, and permitted another non-conflicting task to execute, utilizing the resource. But DeMR datasets are data-dependent, and hence tasks must start executing first. Further, computing the dataset takes up almost the entire task. Hence, when a task needs to be re-executed, all previous work has to be discarded, exacerbating the loss. Note that OLP will also similarly impact a deterministic execution model [135].

When B and D conflict, a Multithreading system like Transactional Memory or Galois may re-execute both tasks immediately, in the hopes of avoiding the data race. They may conflict again. But Parakram will serialize B and D, avoiding this scenario, giving it a potential performance advantage. But frequent conflicts, as is the case in DeMR (30% tasks re-execute), lead to a net loss of parallelism. Further, DeMR's small tasks accentuate Parakram's overheads. Hence Parakram underperforms for DeMR, despite its ample parallelism in DeMR. However, speculation helps Parakram (5.78×) outperform a Deterministic Galois-like scheme (2.43× on our system).

Note that Genome, Labyrinth, and RE are also nondeterministic algorithms. However, in comparison to DeMR, Parakram fares better in these cases. In case of Genome and Labyrinth, the conflict rate is relatively lower and the task sizes are relatively larger. Hence the amount of work discarded is less and serializing the conflicting tasks is beneficial. Parallelism in RE is constrained because tasks update a few common hash tables. Hence both Parakram and multithreaded methods do equally poorly.

Result 5. *A unique combination of the following patterns in an algorithm can cause the ordered approach to underperform: multithreaded tasks that conflict at a high rate, whose datasets are data-dependent, and whose work on the data is otherwise small.*

7.3 Analyzing Ordered and Nondeterministic Executions

So far we have compared ordered execution with nondeterministic execution empirically. We next compare the speedups the two can achieve using an analytical model. Although implementation artifacts influence achievable speedups, we use the model to focus on the fundamental differences and similarities between the two approaches, without being encumbered by the implementation. Achievable speedups, as is evidenced from the CD and DeMR examples earlier, is impacted by the algorithmic properties: dataset types and task types. We will account for them in the model. We first build individual models for the two approaches, and then present a unified model for both.

Consider an ideal system with infinite resources that attempts to perform all of the program's tasks in one epoch, as depicted in Figure 7.12. Assume that the overheads associated with managing the execution are negligible in comparison to the actual work performed by each task, which are also, say, uniform in size. Assume a sophisticated nondeterministic system in which, if tasks conflict, one completes successfully and the rest are re-executed (as opposed to all having to retry). Using Figure 7.12, we will simply follow the convention that when tasks conflict, the "leftmost" completes, without losing generality. Say, tasks B and D conflict with each other. Task B will complete in the nondeterministic execution.

For this analysis, it is helpful to view Parakram's execution model as follows. When Parakram anticipates that a younger task will conflict with an older task, i.e., the younger task is dependent, its dataflow execution attempts to substitute the dependent task with another task to utilize the resources. In Figure 7.12, Parakram will attempt to group the non-conflicting tasks to the "left". Task B will execute and complete, similar to the nondeterministic execution, but Parakram will try to substitute D with another task, perhaps E.

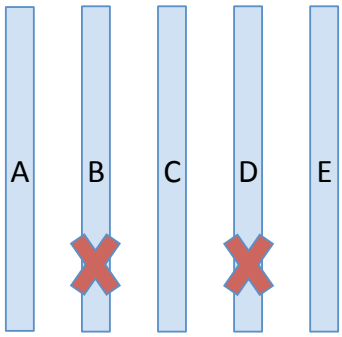


Figure 7.12: Generic tasks executing in an ideal system. Say tasks B and D conflict. Nondeterministic execution will complete B. Ordered execution will complete B and attempt to substitute D with a non-conflicting younger task.

We analyze the maximum speedup the two approaches can achieve in any epoch when N tasks are available for execution. Say on average, a task conflicts with another given task with the probability p, i.e., it does not conflict with the probability $q = 1 - p$. p may also be viewed as the pair-wise conflict probability between the tasks. To compute the total tasks that may successfully complete in an epoch, we first compute the probability for successful completion of each task. We will consider the different algorithm characteristics in turn.

Data-independent Datasets.

When tasks use data-independent datasets, in both types of execution the same tasks will conflict with each other, and both will have equal opportunities to reap the parallelism. Therefore, in Figure 7.12, task A will always complete. Task B completes if it does not conflict with A, i.e., with the probability $(1 - p)$. Task C completes if it conflicts with neither A, nor B, i.e., with the probability $(1 - p)^2$. Therefore, given N tasks, the total number of tasks that will complete, i.e., the achieved speedups, S_n (nondeterministic) and S_o (ordered), over the sequential execution are:

$$\begin{aligned} S_n = S_o &= 1 + q + q^2 + ... + q^{N-1} \\ &= \frac{1}{1-q} - q^N \qquad \text{(sum of a geometric series)} \end{aligned} \tag{7.1}$$

Data-dependent Datasets.

If the datasets are data-dependent, once again in both cases the same tasks will conflict with each other, but the two approaches behave differently depending on the task type. First consider the case when the algorithm is nondeterministic, i.e., all parallel tasks are independent of each other. As we saw in DeMR, nondeterministic execution will deem any task that has not conflicted to have completed. Therefore, the achieved speedup, S_n is the same as in Equation 7.1.

In the ordered case, since the dataset is data-dependent, Parakram cannot immediately apply dataflow and reorder the tasks. If a task conflicts, all younger tasks must wait until the conflicting task eventually re-executes, and check for conflicts with its new results. Effectively, to maintain order, the younger task that conflicts with an older task is treated as dependent even if logically it is not. Therefore, a task will complete only if (i) it does not conflict with any older task, and (ii) none of the older tasks conflict with each other. The second condition implies that no pair of older tasks must conflict. Therefore, task A always completes. Task B completes if it does not conflict with A. But, task C completes if it conflicts with neither A, nor B, and B does not conflict with A. The speedup, S_o, achieved by the ordered execution is:

$$\begin{aligned} S_o &= 1 + q + q^2 \cdot q^{C_2^2} + q^3 \cdot q^{C_2^3} + ... + q^{N-1} \cdot q^{C_2^{N-1}} \\ &= \sum_{i=0}^{N-1} q^i \cdot q^{C_2^i} \end{aligned} \tag{7.2}$$

Where C_2^i is the combination of selecting a pair of (two) tasks from i tasks, and $C_2^i = 0$ for $i < 2$. Note that in the above equation we have assumed that all tasks younger to the conflicting task are deemed incomplete, whereas in practice, only a subset of the younger tasks are likely to actually conflict with the re-executed task. Hence , equation 7.2 is a conservative estimate of the ordered execution's speedup. Nonetheless, we use this equation to keep the model simple, and to compare the worst-case speedup of the ordered execution with the best-case speedup of the nondeterministic execution.

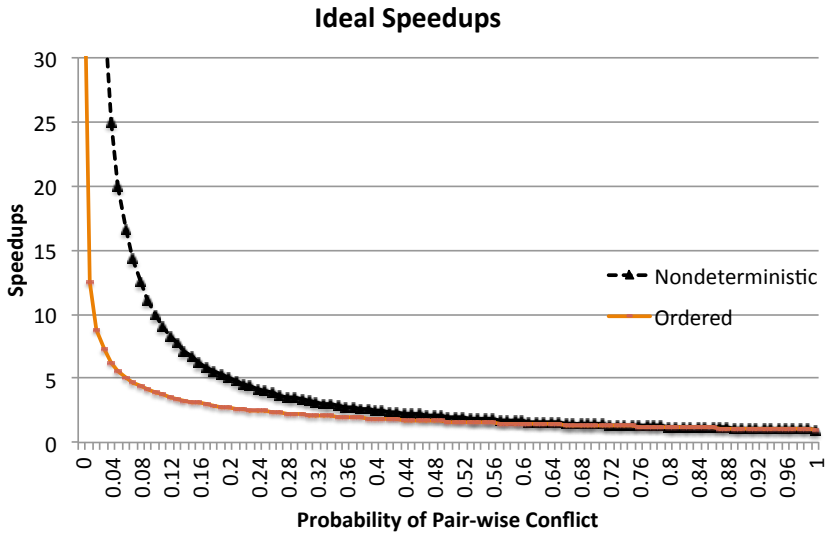


Figure 7.13: Ideal speedups on infinite resources for nondeterministic and ordered executions.

Figure 7.13 plots the Equations 7.1 and 7.2, by spanning p from 0 to 1 and using $N = \infty$. For $N = \infty$, the Equation 7.1 reduces to $S = \frac{1}{1-p}$. To obtain S_o using Equation 7.2, we solved the equation until S_o converges to ten digits after the decimal point. The Y axis plots the achievable speedup for different values of p on the X axis. When $p = 1$, i.e., all tasks conflict, or $p = 0$, i.e., none conflicts, both approaches yield the same speedup. When the probability of conflict is "high", i.e., $> 30\%$, both approaches yield similarly poor performance. However, they differ considerably at lower conflict probabilities. When the probability of conflict is only 0.01%, on infinite resources the nondeterministic approach yields a speedup of $100\times$, whereas the ordered approach achieves $12.5\times$. Nondeterministic execution outperforms ordered execution by an order of magnitude in this case. However, when the conflict probability goes up to 5%, the nondeterministic speedup is $20\times$, whereas the ordered speedup is a much closer $5.5\times$.

Next, if the algorithm were not nondeterministic, i.e., there were true dependences between tasks, then even the nondeterministic execution cannot assume that a non-conflicting task has completed when a prior task (to the "left" in Figure 7.12) conflicts. In this case the nondeterministic

speedup will be similar to the ordered speedup. Therefore, the two lines in Figure 7.13 represent the range of possible differences in speedups achieved by the two approaches when tasks are completely independent and use data-dependent datasets. In other cases, their speedups are comparable.

A Unified Speedup Model.

In a more general case, it is possible that an algorithm contains a mix of independent and dependent tasks, with data-independent and data-dependent datasets. Further, a task may conflict with different tasks with different probabilities. We provide a more generic speedup model, which is applicable to both approaches:

$$
\begin{aligned}
p_{i,j} &= \text{Probability that task } f_i \text{ conflicts with task } f_j,\ i \neq j \\
q_{i,j} &= 1 - p_{i,j} \\
d_{i,j} &= \begin{cases} 1 & \text{always for ordered execution, } i \neq j \\ 1 & \text{if task } f_i \text{ is dependent on task } f_j \text{ in the nondeterministic execution, } i \neq j \\ 0 & \text{otherwise} \end{cases} \\
a_i &= \begin{cases} 1 & \text{if task } f_i \text{ uses data-dependent dataset} \\ 0 & \text{otherwise} \end{cases} \\
S &= \sum_{i=0}^{N-1} \left[\prod_{j=i-1}^{0} q_{i,j} \cdot \left(\prod_{k=i-i}^{1} \prod_{l=k-1}^{0} q_{k,l}^{d_{i,k}} \right)^{a_i} \right]
\end{aligned}
\tag{7.3}
$$

Where $\prod_{m}^{n} = 1$ for $m > n$. The $\sum$ captures the successful completion of each task. The term, $\prod q_{i,j}$, ensures that task f_i does not conflict with any prior task. The second term, $(\prod \prod q_{k,l}^{d_{i,k}})^{a_i}$, is applied only when the dataset is data-dependent ($a_i = 1$), and the sub-term $q_{k,l}^{d_{i,k}}$ is applied when the task f_i is dependent on task f_k ($d_{i,k} = 1$). The second term accounts for any precedent task f_k that may have conflicted. When $a_i = 0$ (data-independent datasets) or $d_{i,k} = 0$ (independent tasks in nondeterministic execution) for all task pairs, and $q_{i,j}$ is the same for all task pairs, the equation

devolves into Equation 7.1. When $a_i = 1$ (data-dependent datasets) and $d_{i,k} = 1$ (dependent tasks) for all task pairs, and $q_{i,j}$ is the same for all task pairs, the equation devolves into Equation 7.2.

7.4 Summary

Ordered programs can match multithreaded programs in expressing parallelism. Ordered programs, at large, do not compromise performance. When algorithms comprise dependent computations, exploiting parallelism appears to come naturally in the ordered method, whereas considerable effort may be needed in Multithreading. In contrast, if the algorithm exhibits irregular, entirely dependence-free parallelism, and uses data-dependent datasets, the artificial constraint of order can limit the exploitable parallelism, which Multithreading is not limited by.

8 Recovering from Exceptions

Your ability to deal with surprise is your competitive advantage.

— David Allen ("Getting Things Done")

In this dissertation we have argued that an ordered approach to multiprocessor programs can simplify system use. Over the last few chapters we presented the details of the Parakram ordered approach and applied it to simplify programming. In this chapter we apply the ordered approach to simplify yet another aspect of multiprocessor system use, recovering from exceptions.

8.1 Introduction

An exception is an event that may alter the program's prescribed execution. Debugging breakpoints, OS scheduling interrupts, and hardware faults are examples of exceptions. For a variety of reasons, as we will see, exceptions are slated to increase in future systems, due to which programs may not execute efficiently, accurately, or to completion. Yet designers will desire systems that recover from exceptions gracefully, and execute programs as intended, with minimum intervention from the users.

Recovering from an exception requires diverting the execution to **handle** the exception-causing event, and once the event is handled, resuming the execution such that it can complete correctly. Handling an exception entails responding to the specific event. For example, recovering from an OS scheduling exception may be as simple as pausing the execution and resuming it at a later time, perhaps on a different resource, requiring no special handling. Recovering from a hardware fault can be more involved. It may require that the program be paused, and the fault be handled by

correcting the program's corrupted state, before resuming the execution from an appropriate point. Typically, exceptions can arise at any time during the execution.

Irrespective of the exception source, pausing the nondeterministic execution of a multithreaded program at an arbitrary point is non-trivial, as will be seen in Section 8.2. This can complicate system design and impose overheads when recovering from exceptions. Approaches to recover from exceptions in nondeterministic programs periodically *checkpoint* the program's state as it executes. Upon exception, they *recover* to a prior error-free state and resume the program, losing all work completed since. A plethora of hardware [5, 134, 145, 176] and software [32, 33, 56, 114, 120, 153, 189] approaches, striking trade-offs between complexity and overheads, have been proposed in the literature. Our analysis shows that due to the overheads, their checkpointing and recovery processes will be too inefficient to handle frequent exceptions. In fact, they lack the scalability needed for future, increasingly larger systems.

By contrast, as we show in this chapter, ordered execution lends itself well to exception recovery. Order can be used to simplify system design, and reduce the overheads related to checkpointing and recovery. Note that Parakram can already recover from misspeculation, which is but a type of exception, with the help of globally-precise interrupts. The same technique can be extended to tackle exceptions in general.

We use the Parakram model as a foundation, draw from multiple past works on recovery, and introduce new features to build a *globally-precise restartable recovery system* (GPRS) for shared memory systems. GPRS provides a generic substrate to recover from different types of exceptions, ranging from debug breakpoints to irrepairable memory faults.

Our design goal for GPRS was to preclude the need for complex hardware since future systems will likely rely on software to handle exceptions [37, 127]. In line with other production-worthy software recovery solutions [33, 114, 120], we developed GPRS as a software runtime system. GPRS is OS-agnostic, fully-functional, and operational on stock multiprocessors. It exploits the Parakram programming APIs, and combines checkpointing and log-based approaches to minimize the overheads and achieve scalability. Significantly, GPRS can handle exceptions in user code,

third-party libraries, system calls, as well as itself.

Although GPRS has wide applicability, in this dissertation we apply it to recover from non-fatal exceptions in Parakram programs. We evaluated GPRS in comparison with the conventional checkpoint-and-recovery method designed for multithreaded programs. Experiments conducted on a 24-context multiprocessor system showed that exception recovery in ordered programs can be more efficient than the recovery in multithreaded programs. Importantly, GPRS withstood frequent exceptions and scaled with the system size, whereas the conventional method did not, validating our qualitative analysis.

In Section 8.2 we overview the reasons for the increasing rates of exceptions and why handling them efficiently is important. The section summarizes and analyzes the prevailing method to handle exceptions. In Section 8.3 we describe how the ordered approach can simplify exception recovery. Section 8.4 presents the details of GPRS and its operations. Applying GPRS to other use cases not considered in this dissertation, is discussed in Section 8.5. In Section 8.6 we evaluate GPRS when applied to recover from non-fatal exceptions, such as soft hardware faults. Many proposals have been made to recover from hardware faults in multithreaded programs; they are briefly summarized in Section 8.7.

8.2 A Case for Handling Exceptions Efficiently

8.2.1 Impact of Technology Trends on Program Execution

As mobile and cloud platforms grow in popularity, and device sizes shrink, designers face energy and resource constrained, unreliable systems. Proposals are being made to address these challenges, but they can affect the program's originally intended execution, even terminally. Their unanticipated affect on the program's execution may be viewed as exceptions. Recovering from these exceptions will make these proposals practical. We analyze these proposals by dividing them into three broad categories: resource management, inexact computing, and fault tolerance. We briefly describe how they may raise exceptions during the program's execution, and make a case for why the exceptions

should be processed efficiently.

Resource Management. Managing resources to save cost or energy can impact a program's execution. Shared systems, e.g., Amazon's EC2 [60] and mobile platforms [3], can abruptly terminate a program, even without notifying the user. Systems are now incorporating heterogeneous resources with disparate energy/performance profiles [29, 30]. To maximize benefits, dynamic scheduling of computations on these resources will be desirable [24]. One can also envision scheduling computations, even interrupting and re-scheduling currently executing computations, on the "best" available resource as the resources become available dynamically.

Inexact Computing. Further, emerging programming models can impact the program execution. Disciplined approximate computing [160], a recent research direction, permits hardware to inaccurately perform programmer-identified operations in a bid to save time and/or energy [65, 66]. Emerging proposals provide a software framework to compute approximately, but with a guaranteed quality of service, by re-computing when errors are egregious [17]. In the future one can imagine integrating approximate hardware with such a software framework for even more benefits. Interestingly, some proposals tolerate hardware faults by admitting computation errors in a class of error-tolerant applications such as multimedia, data mining, etc., while ensuring that the programs run to completion [113].

Fault Tolerance. Finally, hardware design can impact the execution. Faults due to soft (transient) errors, hard (permanent) errors, and process variations can cause programs to crash or compute incorrectly on increasingly unreliable hardware [28, 99]. Hardware designers are employing techniques to manage energy consumption [81, 82] and device operations [188], which can lead to frequent timing, voltage, and thermal emergencies. Like faults, these techniques can also affect the execution. Moreover, growing system sizes makes the systems more vulnerable to such vagaries.

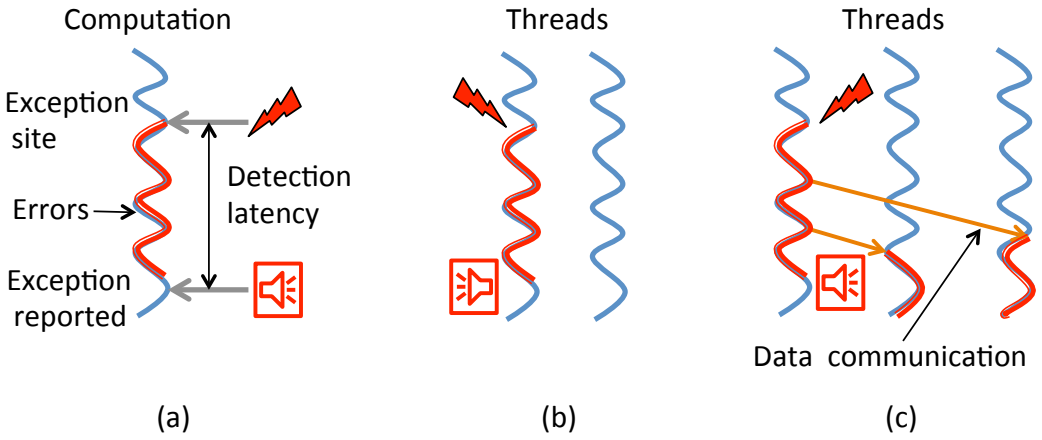


Figure 8.1: Characteristics of exceptions. (a) Latency to detect and report an exception. (b) A *local* exception affects an individual computation. (c) A *global* exception affects multiple computations.

In all of the above scenarios, the program execution may be viewed as one interrupted by **discretionary exceptions**. These are exceptions that the system allows, in bargain for other benefits, such as energy savings. If these exceptions can be tolerated, efficiently, and oblivious to the programmer, designers can effect such and perhaps yet undiscovered tradeoffs. In fact, a similar trend ensued in uniprocessors when precise interruptibility provided a low overhead means to tolerate frequent exceptions. Once introduced, designers exploited the capability to permit high frequency discretionary exceptions arising from predictive techniques, for net gains in performance. For example, techniques to predict branch outcomes, memory dependences, cache hits, data values, etc., relied on precise interrupts to correct the execution when they misspredicted. Hence we believe that recovering from exceptions in parallel programs, and resuming the execution efficiently and transparently, will be highly desirable in the future.

8.2.2 Exceptions

Recovering from exceptions is complicated by two factors. The first is the **detection latency**, the delay between the time an exception occurs and it is reported, which can range from one to many cycles (Figure 8.1(a)). For example, it may take tens of cycles to detect a voltage emergency [82],

or an entire computation to detect errors in results [17]. The second, more pertinent to parallel programs, is due to the dispersal of a parallel program's state among multiple processors. To study the complexity of handling exceptions, we categorize exceptions into two types: *local* and *global*. Local exceptions, e.g., page faults, impact only an individual thread, and can be handled locally without affecting other concurrent threads (Figure 8.1(b)). Local exceptions are often handled by using precise interrupts in modern processors [64, 82]. Global exceptions, in contrast, may impact multiple threads simultaneously, e.g., a hardware fault that corrupts a thread whose results are consumed by other threads before the fault is reported (Figure 8.1(c)). Due to the inter-thread communication in parallel programs and the system-wide impact of global exceptions, global exceptions are complex to handle. This dissertation focuses on global exceptions.

8.2.3 Recovery from Global Exceptions

The traditional method to recover from a global exception is to periodically checkpoint the program's state during its execution, often on stable storage. Upon exception, the most recent possible, error-free consistent architectural state is constructed from the checkpoint, effectively rolling back the execution. The program is then resumed from this point. There are three main types of such *checkpoint-and-recovery* (CPR) methods: *coordinated*, *uncoordinated*, and *quasi-synchronous* [62, 119].

In coordinated CPR, program threads periodically coordinate to perform consistent checkpoints. Since every checkpoint is consistent, it can be used to perform quick restarts after exceptions occur. However, the coordination process incurs the overheads of stopping the participating threads. In uncoordinated CPR, threads checkpoint independently, without the overhead of coordination. But it incurs the expense of recreating a consistent checkpoint prior to restart, which may lead to cascading rollbacks. Cascading rollbacks can cause much of the executed program to be discarded, even entirely. Quasi-synchronous CPR also performs uncoordinated checkpoints, but it additionally logs the inter-thread communication, using which it avoids the cascading rollbacks. However, quasi-synchronous CPR now incurs the additional logging overheads.

Of the three types of CPR, coordinated CPR has gained popularity in shared memory systems

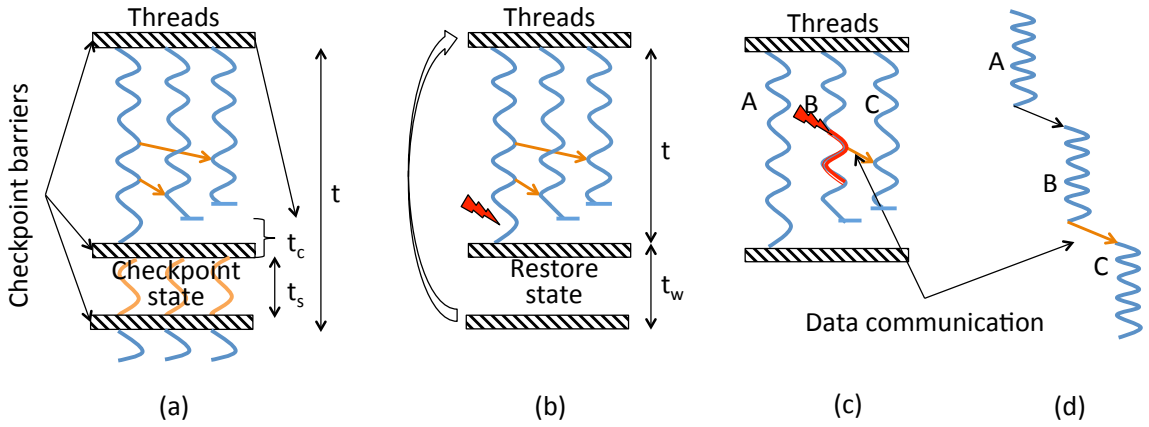


Figure 8.2: Recovering from global exceptions. (a) Conventional checkpoint-and-recovery. (b) Restart penalty. (c) Inter-thread communication. (d) Ordered computations.

since it is relatively simple, and guarantees recovery at relatively lower overheads. Hence we consider coordinated CPR here.

Parallel programs are characterized by nondeterministic concurrent threads that may communicate with each other. Hence, the execution needs to be quiesced to take a checkpoint of the state that is consistent with a legal execution schedule of the program up to the checkpoint [62]. A consistent checkpoint is essential for the program to restart and execute correctly. The lost opportunity to perform work, i.e., the loss of parallelism, during the CPR processes effectively penalizes the performance. Hence, CPR schemes incur a *checkpoint penalty*, P_c, every time they checkpoint, and a *restart penalty*, P_r, when they restart from an exception.

To checkpoint, prevailing software approaches [32, 33, 56, 120, 153, 189] first impose a barrier on all threads, and then record their states (Figure 8.2(a)). A second barrier is used to ensure that a thread continues the execution only after all others have checkpointed, to prevent the states they are recording from being modified. The checkpoint penalty (P_c) is proportional to the average time each context spends coordinating at the two barriers, t_c, recording its state, t_s, and the checkpointing frequency. Ignoring actual mechanisms and assuming contexts can record concurrently, in an n context system, for a checkpoint interval of t sec, the checkpoint penalty for the entire system is,

$P_c = \frac{1}{t} \cdot n \cdot (t_c + t_s)$.

To restart an excepted program, the program is stopped and the last checkpoint is restored (Figure 8.2(b)). In this case the potential loss of parallelism arises from the work lost since the last checkpoint, i.e., over the interval t, and from the wait time, t_w, to restore the state. Therefore, the total *restart delay*, $t_r = t + t_w$, and for a rate of e exceptions/sec, the restart penalty for the entire system is, $P_r = n \cdot e \cdot t_r$.

Recent hardware proposals can reduce the two penalties by involving only those threads (n_c) that communicated with each other during a given checkpoint interval, in the checkpoint and recovery processes [5, 145]. Hence in their case, $P_c = \frac{1}{t} \cdot n_c \cdot (t_c + \frac{n}{n_c} \cdot t_s)$ (all threads still record the state), and $P_r = n_c \cdot e \cdot t_r$.

We believe that as exceptions become frequent, the restart penalty will become critical. Intuitively, if exceptions occur at a rate faster than checkpoints, the program will never complete. Hence, for a program to successfully complete, it is essential that $n \cdot e \cdot t_r \leqslant n$, i.e., $e \leqslant \frac{1}{t_r}$. The hardware proposals can improve this to $e \leqslant \frac{n}{n_c} \cdot \frac{1}{t_r}$. Decreasing P_r by simply increasing the checkpoint frequency, i.e., decreasing t, will increase P_c, and hence may be unsuitable for high-frequency discretionary exceptions.

If exceptions become frequent, as we anticipate, to the tune of making every task, or one in every few tasks, susceptible, we believe that an online system that can quickly repair the affected program state, with minimum impact on the unaffected parts of the program, analogous to the precise interrupts in microprocessors, will be needed. CPR-based systems alternate between checkpoint and execution phases since care needs to be taken that checkpointing and execution do not interfere with each other. Also, recovery requires that the system halt while the state is being restored. For frequent exceptions, this halt-and-start approach may prove inadequate. Hence, for a practical and efficient solution, all factors of checkpoint and restart penalties need to be reduced.

8.3 An Efficient Approach to Frequent Exception Recovery

To build a practical and efficient exception recovery system for frequent discretionary exceptions, we target all the related overheads: the discarded work, the checkpointing frequency, the coordination time, the recording time, and the restart process. Parakram's execution and programming models naturally help address all of these aspects. In particular, Parakram's task-based ordered execution, globally-precise interrupts, and the programming interface can be leveraged for this purpose.

1. Ordered Computations. Ignoring threads for the moment, ideally, when an exception occurs, only the affected computations, i.e., the excepted computation and those which may have consumed the erroneous results produced by it, should be squashed and restarted. For example, in Figure 8.2(c), if only computation C consumes the data produced by B, when B excepts only B and C need restart, without affecting A. Multithreaded programs permit arbitrary and nondeterministic communication between computations, making it difficult to identify the precise dynamic producer-consumer relationships between them. Hence CPR schemes take the conservative approach of discarding all computations after an exception.

To overcome the limitations of nondeterminism, we can take advantage of ordered programs. The implicit order precludes communication from younger computations to older computations. Hence, an affected computation cannot corrupt older computations. When an exception occurs, undoing the effects of the excepted computation and all younger to it results in a program state that is consistent with the execution, as is achieved by globally-precise interrupts.

For exception recovery, a globally-precise interruptible execution can reduce the restart penalty, as compared to CPR, since not all computations, but only the excepted and younger computations need be discarded. The ordering further enables the desired *selective restart* of only the affected producer-consumer computations, and not of other unrelated older or younger computations. This minimizes the amount of the discarded work. This is also analogous to re-executing only an excepted instruction and its dependent instructions in superscalar processors, e.g., a load that misses in the cache but was presumed to have hit, without squashing all other in-flight instructions. Dataflow

execution naturally admits selective restart.

Selective restart potentially reduces the restart penalty to $P_r = e \cdot t_r$ (assuming that the average computation size is t, and time t_w is taken to reinstate its state), since only the excepted computation may need to restart, and the unaffected computations need not stall. This also enables an online system that can continuously respond to frequent discretionary exceptions while minimizing the impact on the program's unaffected parts. For a program to now successfully complete, $e \cdot t_r \leqslant n$, i.e., $e \leqslant \frac{n}{t_r}$. Thus selective restart is potentially $n\times$ more exception-tolerant than the conventional CPR ($e \leqslant \frac{1}{t}$), and $n_c\times$ more tolerant than the recent hardware proposals ($e \leqslant \frac{n}{n_c} \cdot \frac{1}{t_r}$), making it more effective in parallel systems.

Ordering provides other benefits, as we shall see in the design of the GPRS system. It helps simplify the recovery from exceptions in the GPRS mechanisms (Section 8.4).

2. Tasks. To take advantage of order, a naive approach could order the threads themselves, which may serialize the execution. Moreover, the restart penalty will be too large, given the relatively large granularity of threads. Parakram programs already decompose the program into relatively smaller tasks. Checkpoints can be taken at the start of tasks. Further, execution of Parakram tasks is free from races, i.e., tasks are ordered such that they communicate with each other at task boundaries. This execution model yields three benefits.

First, race-free tasks help to localize an exception's impact to a given task if the exception is detected before the task completes, i.e., before the erroneous results are communicated. Second, the relatively smaller tasks increase the checkpoint frequency (reduce t), further reducing the restart penalty, P_r, but potentially increasing the checkpoint penalty, P_c. The third advantage, due to task boundary checkpointing, is that at that time no other task can be communicating with it. This eliminates the need to coordinate (t_c), entirely, reducing the checkpointing penalty P_c to $\frac{1}{t} \cdot n \cdot t_s$ (average task size is t).

Although the ordered approach can reduce the checkpoint and restart penalties as outlined above, it introduces new overheads when applying the order and housekeeping for selective restart,

if we were to view exception handling in isolation. These overheads, e.g., dependence analysis and ROL management, which we have examined in detail Chapter 6, are already part of the ordered approach. But we characterize these overheads afresh from the perspective of exception handling in this chapter. When a context enforces order on a task and checkpoints its state, it can delay the task's actual execution (as seen in the last chapter). If the average delay is t_g, the total penalty of the globally-precise interruptible model, $P_g = \frac{1}{t} \cdot n \cdot t_g$. As we show in the evaluation, the overall benefits of the model far outweigh this penalty.

3. Application-level Checkpointing. Finally, we reduce the recording overhead, t_s, by resorting to application-level checkpointing, which can dramatically reduce the checkpoint sizes [33]. Instead of taking a brute-force checkpoint of the system's entire state [114] or the entire program [56, 153], only the state needed for the program's progress is recorded. Often, instead of saving data, data can be easily re-computed, especially if the computation is idempotent [148]. Several compiler proposals automate application-level checkpointing [32, 120]. Alternatively, the user's intimate knowledge of the program can be exploited for this purpose by requiring the user to annotate the program. Parakram already captures this information in the form of a task's mod set. We take advantage of this aspect.

The above principles, based on ordering computations, enable a responsive and efficient recovery model for discretionary exceptions. No system-wide barriers or other coordination is needed. Since tasks checkpoint independently, the overhead is comparable to that of uncoordinated checkpointing schemes. Although uncoordinated, the checkpoint is consistent with the ordered execution, and hence recovery does not require offline analysis, avoiding any cascading rollbacks.

We next describe one embodiment of the model, the Globally-precise Restartable System. Implementing a recovery system in practice poses additional challenges. We describe how we tackle them while automating the recovery mechanisms.

8.4 Globally-precise Restartable System

Parakram's Execution Manager can already handle exceptions, e.g., the misspeculations. The Globally-precise Restartable System (GPRS) makes this capability more generic. It can recover from simple exceptions, such as OS scheduling interrupts, as well as more complex exceptions, such as hardware faults that can corrupt the program's state. The following description is presented from the perspective of the program-corrupting exceptions, without loss of generality. An overview of the system is presented, followed by assumptions made in its design and details of its operation.

8.4.1 Overview

GPRS builds upon the Parakram Execution Manager, and adds new capabilities. First, GPRS supports exceptions that can affect any arbitrary part of the program. Parakram can handle exceptions in tasks that it knows may misspeculate or when exceptions are raised by the tasks themselves. In these cases Parakram knows precisely which tasks are affected. But a generic exception, like a hardware fault, is external to the program, and can affect any part of the program. Hence it may not always be clear precisely which tasks are affected, requiring additional support to handle such cases.

Second, GPRS adds support for recovery from exceptions in Parakram's runtime operations. Discretionary exceptions can not only affect the user code and functions invoked from it, but also GPRS's own mechanisms. For example, a fault due to voltage emergency can corrupt GPRS operations, which can ultimately corrupt the user program. GPRS is uniquely capable of handling exceptions in its own operations. For this purpose, GPRS records its operations in a Write-ahead Log (WAL) instead of checkpointing its state. The WAL mirrors the History Buffer (HB), except that it records the runtime state or operations instead of the program state. GPRS leverages the task order to optimize the logging.

Third, GPRS adds support for global exceptions, which are reported after a delay, e.g., hardware faults. Delayed reporting of exceptions can cause the execution to be out of sync with the reporting.

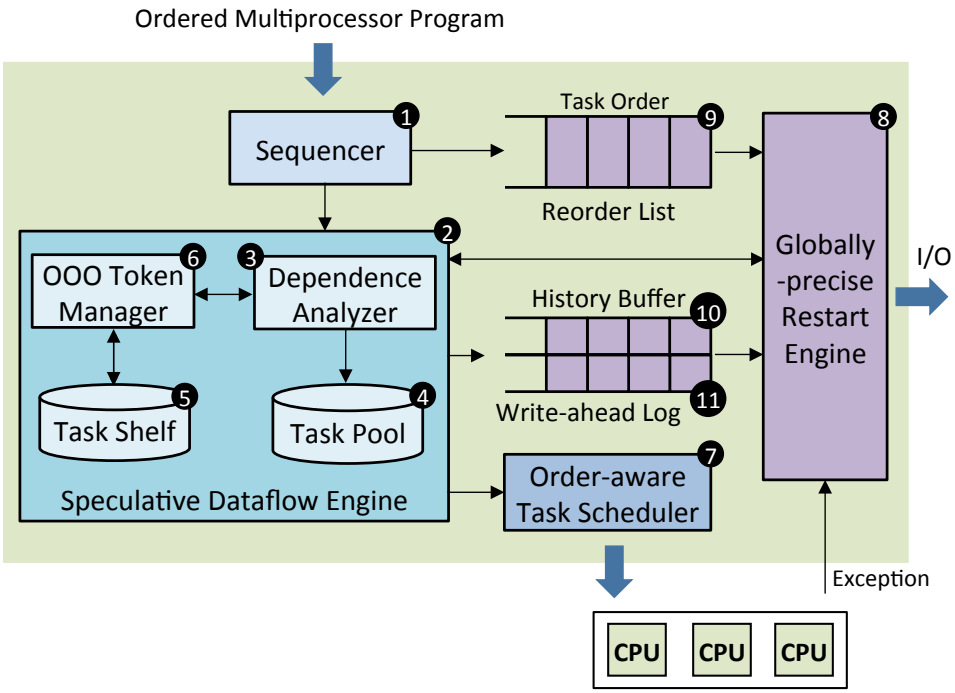


Figure 8.3: GPRS System Architecture.

GPRS must correlate the exception with the corresponding task to perform accurate and efficient recovery.

Finally, GPRS must handle I/O operations and "poorly composed" tasks when exceptions occur. When recovering from an exception, re-performing I/O, e.g., printing, may lead to incorrect execution. Not enough information may be available about poorly-formed tasks to perform proper program recovery. GPRS, like other recovery systems, also has to account for exceptions in operations that perform I/O, and transient events during the re-execution of a recovered program. GPRS can handle all of these cases.

Figure 8.3 shows the block diagram of GPRS, which has the same components as the Execution Manager, except for the added Write-ahead Log (11). Operations related to chiefly executing the program remain the same as we have seen in Chapter 6. GPRS too interposes between the program and the underlying system. It manages the program's parallel execution (essentially the same

functionality seen before), the program's state, and shepherds the execution to completion when exceptions, discretionary or otherwise, arise during the program or from the system itself. GPRS extends the C++ runtime library with its exception-handling functionality.

8.4.2 Assumptions

Designing a recovery system is heavily influenced by the assumptions made about the system and exceptions. We assumed the following.

The system can detect exceptions and report them to GPRS, possibly with a non-trivial detection latency (time taken to report the exception after it occurs). An exception can corrupt the executing program's state. During the detection latency, additional state computed by the program may get corrupted. GPRS assumes that most exceptions can be reported within a given latency, and the system can report the execution context on which the exception occurred.

It is possible that the system cannot detect all exceptions within the stipulated latency. Additionally, some exceptions cannot be traced back to an execution context, e.g., a non-correctable memory error that is detected only when the location is accessed, which may be at a point in the execution far removed from when the location was updated. On occasion, an exception may be irrecoverable, e.g., a system crash. GPRS treats such exceptions as "catastrophic exceptions". The current design does not handle catastrophic exceptions, but we describe how they can be with some enhancements.

GPRS also assumes availability of stable storage to store the WAL and the checkpoints, if recovery from exceptions that can corrupt the program's state is desired. Incorporating stable storage in a recovery system requires careful attention [165]. Access characteristics of the storage can contribute to the overall overheads. We focused on evaluating the restart overheads in this dissertation, hence we treated the main memory of the system as stable storage in our experiments. Whenever a checkpoint is logged by an execution context, GPRS assumes simplistically that the execution advances only after the checkpoint is known to have been correctly committed to the storage. Storage-related overheads are proportional to the size and frequency of checkpoints, which will be similar in both GPRS and conventional CPR for a given program. Hence, our assumptions

about the storage characteristics does not alter our main analysis.

The GPRS design, as presented next, assumes that online recovery of frequent exceptions is desired. Hence it checkpoints frequently, incurring proportional overheads. If exceptions are not expected to be frequent, the checkpointing frequency can be reduced. In general, for a given use, the checkpointed state size in GPRS and CPR will be similar.

We do not address recovery from exceptions in the actual I/O operation, e.g., hardware faults during the transmission of data over the network, although we can address exceptions in the computations that produce the data. GPRS being a software system, also assumes that an exception that corrupts the address of a main memory location being updated, is reported before the update takes place. Handling such errors is non-trivial, even in systems with hardware support, and requires special handling [175].

8.4.3 Executing the Program

The program executes as before (Chapter 6), but for the following modifications. Recall that a Parakram program comprises alternating control-flow (CF) tasks and dataflow tasks. To perform speculative execution, Parakram already relies on checkpointing the mod sets of the dataflow tasks. To handle arbitrary exceptions, GPRS now also checkpoints the mod sets of the control-flow tasks. It once again relies on the programmer to provide these mod sets using the `pk_declare()` or `pk_mod()` APIs.

Previously we saw dataflow tasks and sub-tasks being logged in the Reorder List (ROL). Now, when the program starts executing, both control-flow and dataflow tasks are logged in the ROL. Control-flow tasks are also assigned the hierarchical ID indicating their position in the total order, as before. Before a task, whether dataflow or control-flow, is executed, in addition to its mod set, its stack is checkpointed and logged in the HB. When recovery needs to be performed, this checkpoint, of the mod set and the stack, is used to unroll the task's execution.

8.4.4 Recovering from Exceptions in the Runtime

The Parakram runtime employs data structures and sophisticated concurrency algorithms in its own operations (Chapter 6). These data structures include the work queues, the memory allocator lists, the ROL, and other book-keeping structures. While the operations are invisible to the user, exceptions during these operations, e.g., hardware faults, can impact the runtime's state as well as the user-visible state.

When an exception affects the runtime, GPRS must provide two capabilities: (i) identify a restart point, and (ii) rectify the affected data structures. In addition to rolling back the program's state to an error-free state, the runtime data structures need to be brought to an error-free state that is also consistent with the program's state. To ensure correctness, GPRS effectively rolls back the execution to the oldest affected task by restoring the collective program and runtime state to that point.

Restarting the Program.

Exceptions during the runtime operations may ultimately manifest as exceptions in the user computations. We note that each runtime operation is performed on behalf of one or more tasks. Hence we treat them as such. We attribute the exception in a runtime operation to the corresponding oldest task. Essentially, the Prelude and Postlude phases (Figure 6.15) for a task get logically combined with the task for the sake of recovery from exceptions. For example, logging a task's entry in the ROL can be attributed to the task. An exception during the communication between a completing task and a dependent task (e.g., the token passing in the Postlude phase) can be ascribed to the completing task. Similarly, an exception when retiring a task from the ROL can be ascribed to the task. Once so ascribed, and the runtime state is corrected (seen next), the exception can be processed as an exception in that task. Program resumption simply forces the runtime operations to be performed again.

Managing the Runtime State.

Runtime operations and data structures are as challenging to exception recovery as nondeter-

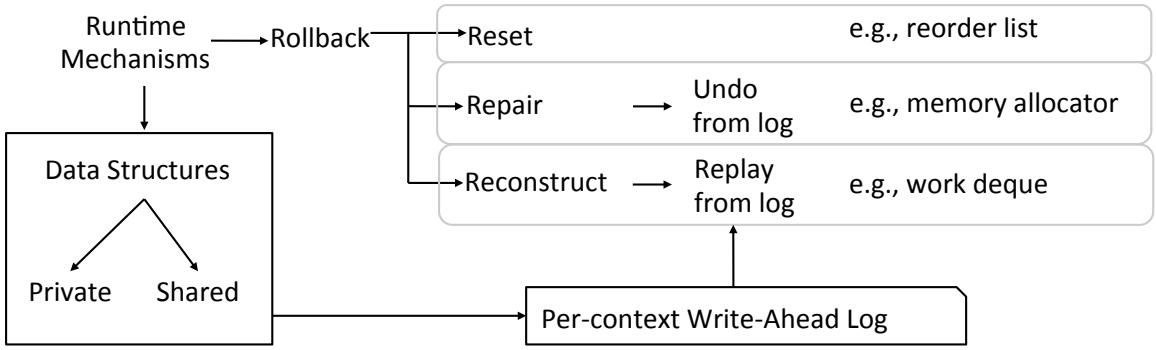


Figure 8.4: Managing runtime mechanisms and state for exception recovery.

ministic programs. The runtime mechanisms can be distilled into operations on its context-private data structures, e.g., the memory allocator structures, and shared queues, e.g., the work deques. They employ concurrency techniques at a much finer granularity than tasks. While the program computations are race-free, these operations may not be. They are in essence, nondeterministic. Therefore they cannot be subjected to trivial cloning and rollback.

We once again exploit the program order to effect the rollback of the runtime state. We use a combination of three strategies (Figure 8.4) to restore runtime data structures: (i) reset them to consistent states, (ii) "repair" them to reflect their pre-exception conditions, or (iii) reconstruct them to their pre-exception states.

Resetting a data structure can be straightforward, e.g., the ROL can be assumed to be empty when there are no in-flight tasks in the system. If the program is to be restarted assuming that all in-flight tasks should be discarded, the ROL can be simply reset.

We repair context-private structures when resetting them would result in lost work, and reconstructing them would be prohibitive. For example, restoring the state of the memory allocator, which may accumulate state over the entire duration of the program, may be more practical than reconstructing its state from the start.

We reconstruct the concurrent structures, e.g. the work deques, instead of repairing them, to avoid run-time inter-context coordination.

For the latter two strategies, we draw inspiration from the Aries [130] recovery mechanism, commonly used in DBMS, and adapt it to our model. We already assign each task a unique ID, reflective of its order. Before operating on the runtime data structures, each context uses a write-ahead-log (WAL) to independently log the logical and/or physical operation, the task ID on whose behalf it was performed, and the wall clock time when the operation was initiated (whose use will be seen below). Each context maintains a private ordered log, in which the information is recorded before the action is performed. For example, before logging an entry for a task in the ROL, the context logs the action, the task ID, and the pointer to the task, in the WAL. When allocating an object, the memory allocator first logs the action and the head pointer of the memory pool in the WAL. To deallocate an object, it first logs the action and the tail pointer to the memory pool. No inter-context coordination is needed to update the WAL since each context maintains its own private log.

The logs are used to undo/replay operations when repairing/reconstructing the structures. Log-based recovery is described in great detail by Mohan et al. [130], although the fact that each context maintains a private log simplifies the process. We assume that GPRS can store the WAL and the HB on a fault-free storage, e.g., on non-volatile memory and/or disk, when GPRS is applied to hardware fault-recovery. Hardware faults can corrupt the program's state. Non-corrupt checkpoint is needed to ensure correct recovery, hence the need for stable storage.

Although the WAL logging is uncoordinated, we can reconstruct the precise architectural state of the runtime, as follows. First we determine the identity of the excepted task. When possible, structures are reset. Next, for each aborted task, the logs are used to determine whether the operations on context-private structures, e.g., memory allocation, took effect. For example, when an object was deallocated, the logged tail pointer is compared with the current pointer to check whether the operation was indeed performed. If the operations were performed, they are undone in the reverse chronological order. Task IDs provide the inter-task order and the sequence in the log provides the intra-task order.

To reconstruct the concurrent queues, the affected structures are first identified using the

excepted task. This is done by scanning the WAL for operations performed on behalf of the task. The state of the concurrent queues just past a retired computation (no longer in the ROL) can be assumed to be reset (entries in the log for retired computations, if present, may be ignored). The state of the concurrent queues can be reconstructed by sequentially replaying the performed operations (but not executing the actual task), for each exception-free, or up to the desired task in flight.

Entries in the WAL are removed when computations retire. The WAL size can be bounded by controlling the execution.

8.4.5 Recovering from Global Exceptions

As the program executes, the Globally-precise Restart Engine (GRX) tracks the completion of tasks, and retires them as they complete exception-free. When a task completes, its entry in the ROL is stamped with the clock time, indicating the time of completion. The GRX treats both control-flow and dataflow tasks alike in this regard.

When exceptions arise, GRX reconstructs the program's error-free state and that of the GPRS, using the recorded state in the History Buffer and the WAL, as described above. It then instructs the Speculative Dataflow Engine (SDX) to resume the execution from this reconstructed "safe" point.

Because exceptions can be reported with a delay, GRX does not retire completed tasks immediately. GPRS assumes a maximum user-specified detection latency, as do other fault-tolerance proposals [176]. Once a task completes, GRX waits until the latency elapses before retiring the task. If no exception is reported, the task is retired as usual.

When an exception is reported on an execution context, all tasks and runtime operations performed by that context within the detection latency are deemed excepted, and any transitively data-dependent tasks (identified from the dataflow graph) and control-dependent tasks (identified in the ROL) are deemed to have consumed erroneous results. If a runtime operation is found to be affected, the corresponding task is said to be excepted. The affected tasks are marked as excepted, and GRX recovers and restarts from the exception.

8.4.6 Third Party Functions, I/O, System Calls, and Transient Events

Programs often perform I/O, and use system and third-party functions. These functions pose unique challenges to recovery and restart. GPRS must have access to their mod sets, and have the ability to re-execute them, which may not always be the case. Moreover, some I/O operations, e.g., network I/O, cannot be "undone", and re-executing them may lead to incorrect results if they are performed more times than intended. Not unique to GPRS, these issues arise in any recovery system. There are two aspects to tackling them: rolling them back as part of recovery, and exceptions arising during their operations. Ordered execution can simplify both.

A certain class of I/O operations, e.g., file read and writes, can be made idempotent [161]. As an example, GPRS implements its own version of file I/O system calls, which have been made idempotent. GPRS can track the associated internal state and checkpoint them as needed, with no further input from the user, and can automatically perform their rollback and recovery. Similarly, GPRS also implements its own version of the memory allocator, and automatically recovers its state when needed. GPRS's implementations of the file I/O calls and the memory allocator eventually use the system-provided calls for I/O and memory allocation, which too need to be exception-tolerant to provide end-to-end exception tolerance. The GPRS implementations are adequate examples of how such functions can be made exception-tolerant.

Handling non-idempotent computations, e.g., network I/O, or a third-party function whose mod set is unknown, requires some help from the user. The user can invoke such functions using `pk_task_ic()` and identify them to GPRS. While other tasks may execute in dataflow fashion, functions invoked via `pk_task_ic()` execute only when they reach the ROL-head. Thus such functions do not execute out of the program order, and their rollback is unnecessary if exceptions occur in other tasks. However, exceptions in these functions also need to be handled.

Exceptions in tasks that produce data for non-idempotent I/O can be handled in the same way that the output-commit problem is handled, by waiting for the exception-detection latency before committing the data to output [175]. Of course, the same approach may be also applied

to non-idempotent I/O. An exception in functions with unknown mod sets can be treated as a catastrophic exception, whose handling will be seen later, although this feature is not supported by GPRS at present.

Non-repeatable Events The non-repeatability of certain operations, such as system calls, e.g., `random()`, asynchronous events, e.g., interrupts, and uninitialized variables can cause the program to diverge from its original execution, after exception-recovery. If repeatability is desired during a program's re-execution, the non-repeatable events can be recorded, including their outcome where applicable, along with the associated dynamic task ID. The events can be played back, or their recorded outcomes can be supplied to the precise related tasks when they re-execute. Care needs to be taken that the outcomes are recorded only when they are known to be non-faulty. Uninitialized variables can be caught using software engineering tools, and reported to the programmer during the program development.

8.4.7 Exception-recovery Operations

Chapter 6 already describes the general Parakram operations, implemented by the Globally-precise restartable engine (GRX), to recover from exceptions that affect only the user program. We now describe GPRS operations to handle generic exceptions. The operations are described in two steps: the Basic Restart operations (BR), and the Selective Restart (SR) operations.

Basic Restart.

Basic Restart (BR) takes a conservative approach. It presents two user-selectable recovery choices. When a task excepts, either all current in-flight tasks can be discarded, like conventional CPR, or only those younger to the actually affected task can be discarded.

BR design is simpler than SR, and relies on minimum logging. Only the ROL and memory allocator operations are logged. When an exception occurs, GPRS pauses the program. The GRX identifies the excepted and affected tasks, based on the recovery choice made, and marks them so

in the ROL. It then aborts all executing younger tasks, and permits older tasks, if any, to complete. Once the execution quiesces, GRX initiates recovery. It makes the ROL consistent with the retired tasks in the WAL, and retires tasks that can be. It walks the ROL in the LIFO order, restoring data using their clones. Next, it repairs the memory allocator using the log. Since all tasks have either completed or terminated, all runtime data structures (work deques, ROL, wait lists, WAL, etc.), except for the ROL-head and its entry in the WAL, are reset. The program state now reflects sequential execution up to the (only) task in the ROL, from where the program may be resumed.

An exception in the very first control-flow task of the program, i.e., the program segment between the program start and the first dataflow task, which has executed only on a single context, causes the program to re-execute from start.

Selective Restart.

Selective Restart (SR) minimizes the amount of discarded work after an exception, and hence yields higher performance than BR. In comparison to BR, it performs additional logging to facilitate selective restart of only the affected tasks. In addition to the ROL and allocator operations, each context logs all the operations performed in the Prelude and Postlude phases, e.g., the token protocol operations, work deque operations, etc. When an exception is signaled, the program execution is paused while the recovery is performed. The GRX identifies excepted and affected tasks, and marks them so in the ROL. The IDs of affected tasks are used to identify the affected data structures, which are either repaired (e.g., memory allocator) or reconstructed (e.g., ROL) using the WAL. The affected tasks' data are then rolled back using the clones. Entries for control-dependent tasks are removed from the ROL, the respective Wait lists, and the WAL. Once the recovery completes, all paused computations resume while the remaining affected tasks are submitted to the Task Pool.

An exception in a work deque is handled as a special case. When a work deque operation excepts, e.g., when a task was being enqueued or dequeued, no task is affected since its execution has not begun. The affected deque is reconstructed by replaying the operations performed on it after the last retired computation. The rest of the recovery process remains the same.

Thus GPRS recovers from exceptions.

8.4.8 Recovering from Exceptions During the Recovery

It is possible that an exception arises during the recovery process. If the exception is reported before the process is complete, the current recovery can be discarded and the recovery itself can be restarted. If the exception is reported after the process completes, the exception will manifest as an exception in the re-executing tasks, which are handled in the same fashion as above.

8.5 Other Applications of GPRS

We provide a few examples of how GPRS, with appropriate enhancements, may be used to simplify other aspects of system use.

Recovery from Catastrophic Exceptions. In case of a "catastrophic" exception, e.g., in which the integrity of the system is suspect, or to migrate a program, or to analyze the execution, or when an exception cannot be attributed to a specific execution context, **global checkpoints**, possibly committed to stable storage, can be used. A global checkpoint is the checkpoint of the entire program's state. It is often created incrementally as the program executes. To create a global checkpoint in GPRS, it is sufficient to checkpoint: (i) a task's mod set as the task retires, and (ii) the registers and the PC of the following task. A checkpoint created by this process is sufficient to resume the program either on the same machine once it is restored, or another machine. Since in the GPRS execution subsequent tasks may modify the objects in the task's mod set by the time it retires, the task's mod set must be cloned immediately after it completes (cloning objects prior to a task's start may not always be necessary, as is the case in this application). The clone memory may be recycled when the task retires. Only the retiring context need perform the checkpoint, while others may continue to perform work. No system-wide barriers or other coordination is needed. When a catastrophic exception occurs, the last known good global checkpoint may be used.

Handling More Frequent or Infrequent Exceptions. GPRS assumes that exceptions may approximately occur at the same rate as tasks, and hence checkpoints at task boundaries. If the exceptions are relatively more frequent, i.e., multiple exceptions may occur during a single, perhaps long running, task, then for an even finer-grained recovery the task's mod set may be checkpointed more frequently (at the expense of additional overheads). The runtime can interrupt the task periodically to perform these checkpoints.

On the other hand, if exceptions are not expected to be as frequent, then the checkpointing frequency can be reduced. Checkpoints may be taken every few tasks to reduce the checkpointing overheads. As tasks execute, objects can be checkpointed, and already checkpointed objects need not be checkpointed until they are encountered in the next checkpointing interval. This process requires additional housekeeping to track the objects already checkpointed in a given interval.

Overall, for a given checkpointing frequency and checkpointing scheme, the size of the checkpointed state by both GPRS and CPR will be similar.

Debugging and Testing. Globally-precise interrupts enables traditional debugging capabilities similar to those of sequential programs. At any instance, either the system-wide concurrent execution state or a sequentially consistent state, which is repeatable, can be obtained. Users may inspect program state and resume execution as they would in a sequential program, e.g., by using breakpoints. One can envision implementing debugging mechanisms, e.g., reverse execution and reverse debugging, using the basic ability to "rollback" computations. Reproducing an execution and halting it, or creating logs precisely when desired to test programs is also straightforward. Enhanced ROL management, e.g., retiring computations from the ROL using different criteria (such as user-specified conditions) can also be used to provide enhanced debug capabilities.

Fault Detection. We have applied ordered execution to recover from exceptions once they are detected. A common approach to detecting hardware faults is to run redundant temporal or spatial replicas of the program, possibly on different systems. The results produced by the replicas are

periodically compared. Any divergence is treated as a fault. This requires that the replicas produce identical results (when fault-free, and for the same input). Ordered execution naturally yields such an execution. The values of the mod set and the stack state of each task can be compared between the replicas, possibly when the tasks retire. Establishing identity between computations of the replicas is straightforward given their sequential order. The global checkpoint may be used for a more system-wide fault detection. No additional handling is needed to execute the replicas temporally, possibly even interleaved with each other, or to execute them on heterogeneous systems. Nondeterministic execution does not lend itself well to such redundancy-based fault detection.

Computer Security. Precisely interruptible execution that also produces repeatable results across runs is also desirable in systems in which security is important [57, 83]. Often in these systems, the execution is examined once malicious activity is suspected. In GPRS, since global checkpoints of two different executions of the same program (for the same input) will always be identical, they can be used to audit an execution against a known-good execution. Accountable systems can use the audit to identify break-ins, deliberate manipulations, platform mis-configurations, and also identify the responsible party. The global checkpoint may be analyzed to detect malicious actions, and so can the operations resulting from re-execution of a program from a given checkpoint. Importantly, all of the above security checks may be applied offline, once an execution completes, or online, simultaneously with the execution, with or without a lag.

8.6 Experimental Evaluation

We evaluated GPRS by applying it to recover from program-corrupting exceptions, such as transient hardware faults. Exceptions were "injected" during the program's execution. Selective Restart (SR) was used to recover from them. We evaluated SR's overheads, and its ability to recover from exceptions at different exceptions rates and system sizes (represented by the execution context counts). We compare GPRS with conventional CPR.

Program	Baseline Execution Time (s)							
	1	2	4	8	12	16	20	24
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Barnes-Hut	309.6	158.86	83.21	42.72	32.57	29.62	29.46	28.4
Black-Scholes	301.59	151.64	78.2	42.86	35.33	38.12	42.83	48.58
Pbzip2	287.14	147.12	77.34	62.45	61.3	61.77	61.58	63.66
Swaptions	421.7	212.18	105.86	52.92	40.45	36.83	33.19	32.32
RE	18.77	3.24	10.5	9.78	10.38	11.04	13.02	13.73
Histogram	2.04	1.09	0.6	0.32	0.29	0.26	0.23	0.31

Table 8.5: Baseline execution time of the programs at different context counts, with no support for exception recovery.

We emulated conventional CPR using the Base Restart (BR) implemented for Parakram programs in GPRS, instead of CPR implemented in multithreaded programs. This allows us to compare the restart-related aspects without the unrelated differences arising from different programming models and the global barriers used in CPR. Like CPR, we used BR to discard all current in-flight work when exceptions occur. Further, BR is already more efficient than CPR, since BR does not perform the barrier-based checkpoints. (As our other related work on conventional CPR shows, barrier-based checkpointing can incur high checkpointing overheads [80].) Hence, for this study, using BR allows us to focus on the restart penalty, which is likely to have a larger impact on recovery in future systems.

We used Barnes-Hut, Black-Scholes, Pbzip2, Histogram, Swaptions, and RE in the evaluation. From among the programs we studied in this dissertation, these programs were picked to span different task sizes, checkpoint sizes, performance scalability, total tasks launched, and the frequency of system calls (I/O and memory allocation). Swaption task sizes are relatively large, Barnes-Hut, Black-Scholes, and Pbzip2 tasks are smaller, and RE and Histogram tasks are very small. Barnes-Hut and Swaptions scale the best, Black-Scholes, Pbzip2, and Histogram less so, and RE barely. Mod sets in Black-Scholes were the largest. Pbzip2. Histogram, Swaptions, and RE mod sets were small in size. Mod set sizes for Barnes-Hut were in-between. Histogram launched the least number of

tasks (as many as contexts in the system), and Black-Scholes the most, 400K. We used `pk_declare()` to provide the mod sets of the control flow tasks in the programs (in addition to the mod sets for the dataflow tasks). All programs invoked system calls, RE, Barnes-Hut, and Black-Scholes more frequently than others. Black-Scholes, Pbzip2 and Swaptions also write to files. Experiments were conducted on the 24-context machine (specifications are given in Table 7.3). The baseline execution time of the programs on different context counts, when no exception recovery support is included, is provided in Table 8.5.

Exception Model and System Assumptions.

Many fault-detection techniques have been proposed [125, 136, 175], and are beyond the scope of our work. We assume that mechanism(s) in the system detect faults and/or other types of exceptions, and report the excepting context to GPRS, with a maximum latency of 400,000 cycles, as have others [176]. We emulated this by launching one additional thread in GPRS, which uses Pthreads signals to periodically signal the other contexts, and randomly designates one of them as "excepted". We assume that in the interim, the program state is corrupted (the test setup does not actually corrupt the state). Although GPRS implements exception-tolerant file I/O and memory allocator functions, the system I/O functions and the GCC memory allocator that it eventually uses are not. Hence we blocked the signals around the system and GCC calls, effectively delaying the exception signal until it was safe to process them.

We also assumed the availability of stable storage to maintain the logs. Recovery from a corrupt stable-storage, exceptions that cannot be attributed to a context, and system-level policies, e.g., handling permanent, repeating or non-recoverable failures are beyond the scope of our experiments. We did not model the storage characteristics; our experiments used the system's memory as storage for both CPR and GPRS.

We also assume a single-fault model. We stress-tested the system under various exception rates (without emphasizing the distribution) since our focus was on restart overheads. As will be seen, since the current GPRS design tackles exceptions at task granularity, the program's task size dictates

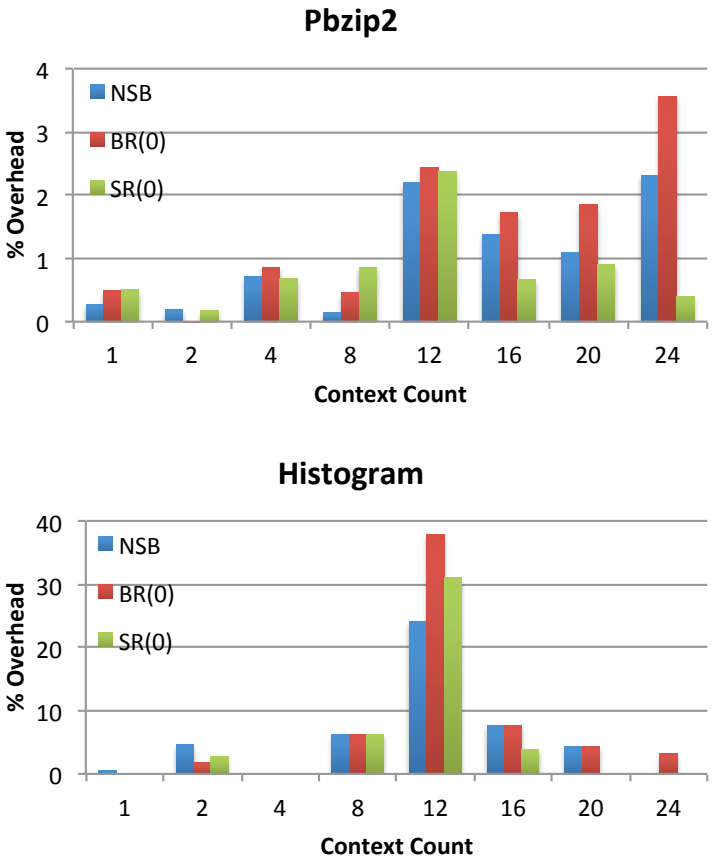


Figure 8.6: Basic overheads for Pbzip2 and Histogram. NSB bars show the ROL management and checkpointing overheads. BR(0) and SR(0) bars show the overheads once the signal blocking is applied to system functions.

how well the program can tolerate the exceptions. If tasks are small, the program can tolerate relatively high rate of exceptions, and if the tasks are large, the programs can tolerate relatively lower rate of exceptions. Hence in our experiments, we varied the exception rates to match the program's characteristics. This allowed us to emulate conditions in which individual program tasks are susceptible to exceptions.

We summarize the key results by analyzing the graph bars in the figures to follow. The figures show the overheads incurred by the exception-tolerant programs over their non-exception-tolerant

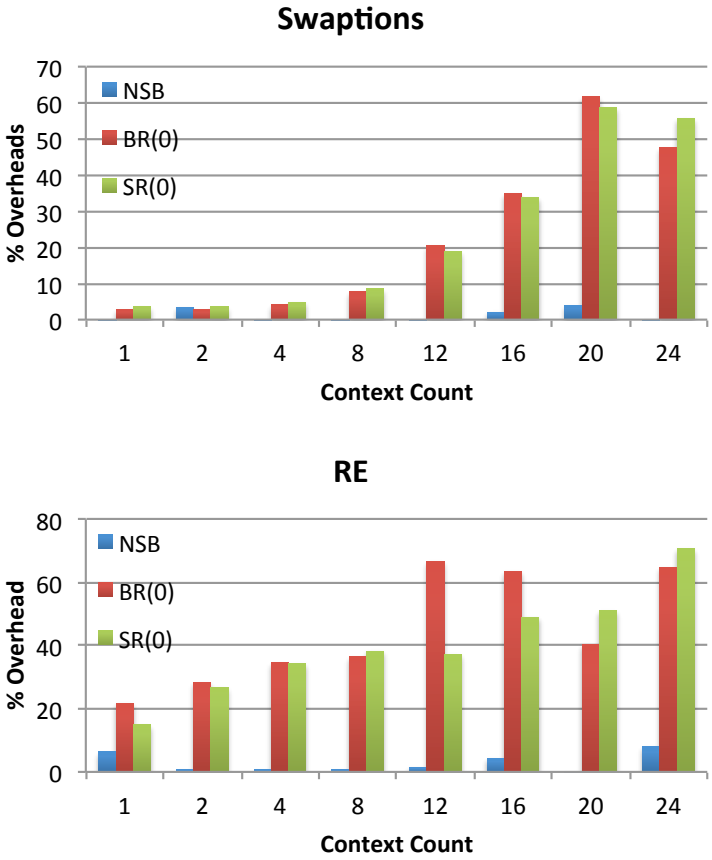


Figure 8.7: Basic overheads for Swaptions and RE. NSB bars show the ROL management and checkpointing overheads. BR(0) and SR(0) bars show the overheads once the signal blocking is applied to system functions.

versions (baseline) for the given context count. We studied the impact on overheads under the following conditions: (i) increasing context count, (ii) recovery using BR and the GPRS selective restart (SR), and (iii) increasing exception rates. The Y-axis plots the overheads in percentage. The X-axis plots the context count. Overheads consider the entire program, including all file I/O operations. The discussion is divided into basic (no exceptions) and exception recovery overheads.

Basic Overheads.

Refer to Figures 8.6, 8.7, and 8.8. The *NSB* (no signal blocking) bars show the BR overheads when

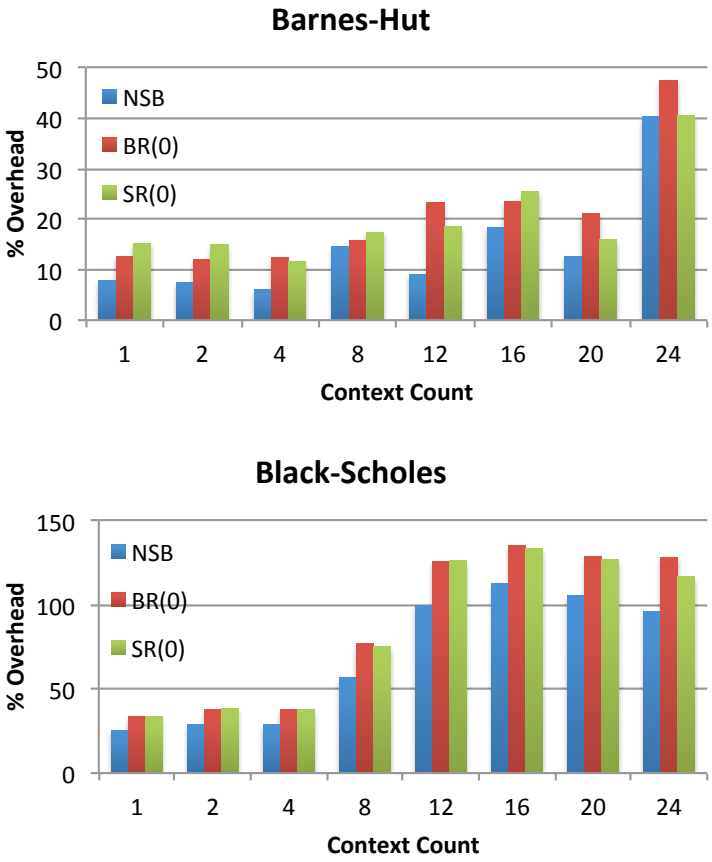


Figure 8.8: Basic overheads for Barnes-Hut and BlackScholes. NSB bars show the ROL management and checkpointing overheads. BR(0) and SR(0) bars show the overheads once the signal blocking is applied to the system functions.

no exceptions occur and the Pthreads signals are not blocked (the baseline execution times over which these overheads were measured are listed in Table 8.5). Thus they show the basic BR overheads which arise from ROL management and checkpointing. In general, across all programs, the ROL management overheads are small (<10%). The checkpoint penalty is program and context-count dependent.

Pbzip2 and Histogram overheads are shown in Figure 8.6, and Swaptions and RE overheads are shown in Figure 8.7. Since these programs mostly perform idempotent computations, they

require relatively less checkpointing. Hence, they expose the ROL management overheads. ROL overheads are fairly constant in these four programs, with the maximum of ~10% (RE, 24 contexts). Histogram shows shows over 20% overheads at 12 contexts, but we believe that this is an artifact of the experimental setup. Histogram's total execution time at 12 contexts is about 0.3s, and hence even a slight perturbation in the experimental setup can dramatically affect the measurements.

Figure 8.8 shows the basic overheads for Barnes-Hut and Black-Scholes. They perform a large number (150K and 400K) of tasks, and still incur low ROL management overheads, but their overheads are relatively higher due to the checkpoint penalty. Barnes-Hut and Black-Scholes checkpoint large amount of data. The NSB bars at one context expose the checkpoint penalties, ~8% and ~25%, respectively. In general, BR's checkpoint penalty is similar to SR's, across all contexts, as expected from their similar checkpointing mechanisms and the total state checkpointed. As the contexts increase, a program's execution time and the absolute checkpoint penalty decrease, but the penalty can be up to 135% of the execution time (Black-Scholes, at 16 contexts). We note that the actual overhead is not rising, but the checkpointing starts affecting the program's scalability, which manifests as overheads. This will be seen again when we start injecting exceptions during the execution. The harmonic mean of the total basic overheads across all programs and contexts is 25% (not shown).

The *BR(0)* and *SR(0)* bars in the three Figures, 8.6, 8.7, and 8.8, show BR and SR overheads with unblocked signals, but at 0 exceptions (baseline execution times are in Table 8.5). The bars at 1 context show that the absolute overheads are quite small, ~15% in the case of RE. Hence they incur signal-blocking overheads, proportional to the frequency of the system calls. Barnes-Hut, Black-Scholes, Swaptions, and RE perform relatively more frequent system calls (file I/O and/or memory allocation), incurring larger blocking-related overheads, up to ~70% in the case of RE at 24 contexts. Once again, the absolute overheads of signal blocking are not increasing, but signal blocking is affecting the program's scalability, which is manifesting as overheads. *SR(0)* incurs the logging overheads for selective restart, over and above the *BR(0)* overheads. Data shows that it is almost negligible in most cases, but can manifest as ~15% in the case of RE at 20 contexts.

Program	Expt. /s	Rcvry.	Delivered Exception Rate							
			1	2	4	8	12	16	20	24
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
Barnes-Hut	1	BR	0.992	0.973	0.958	0.955	0.853	0.833	0.912	0.869
		SR	0.994	0.985	0.980	0.938	0.947	0.943	0.927	0.903
	2	BR	1.974	1.921	1.854	1.792	1.720	1.757	1.660	1.598
		SR	1.983	1.961	1.947	1.940	1.883	1.892	1.877	1.756
Black-Scholes	1	BR	0.953	0.915	0.862	0.799	0.799	0.824	0.816	0.822
		SR	0.954	0.918	0.926	0.805	0.804	0.820	0.833	0.820
	2	BR	1.907	1.819	1.703	1.589	1.581	1.611	1.641	1.568
		SR	1.909	1.827	1.689	1.567	1.578	1.625	1.637	1.585
Pbzip2	0.05	BR	0.045	0.045	0.036	0.042	0.043	0.044	0.043	0.041
		SR	0.045	0.044	0.035	0.048	0.032	0.032	0.032	0.031
	0.2	BR	0.179	0.162	0.150	0.111	0.117	0.119	0.116	0.116
		SR	0.179	0.148	0.133	0.141	0.144	0.145	0.127	0.137
Swaptions	0.033	BR	0.031	0.029	0.025	0.033	0.033	0.033	0.033	0.027
		SR	0.031	0.033	0.025	0.033	0.033	0.033	0.033	0.033
	0.04	BR	0.038	0.036	0.033	0.027	0.020	0.033	0.027	0.031
		SR	0.038	0.036	0.040	0.040	0.040	0.040	0.038	0.040
RE	5	BR	2.616	1.755	1.136	1.091	0.949	0.866	0.638	0.580
		SR	2.604	1.823	1.050	0.524	0.324	0.226	0.159	0.089
	10	BR	5.250	3.606	3.456	3.254	2.845	2.552	2.265	2.302
		SR	5.263	3.553	2.141	1.048	0.556	0.400	0.260	0.171
Histogram	5	BR	∞	∞	∞	∞	2.174	2.326	0.000	1.923
		SR	∞	3.509	4.054	4.255	5.263	2.381	0.000	2.941
	10	BR	∞	∞	∞	∞	∞	9.068	8.271	∞
		SR	∞	∞	8.661	8.955	7.143	7.317	7.143	6.061

Table 8.9: Injected exception rate (Expt.) and the delivered rate of exceptions at the different context counts (columns 4 to 11), for the two types of recovery (Rcvry.) systems. ∞ indicates that the program failed to complete.

Exception Recovery Overheads.

We now examine the overheads of recovering from non-zero exceptions. Using Linux signals to deliver exceptions in the experimental setup is imprecise. Linux controls when a signal is actually

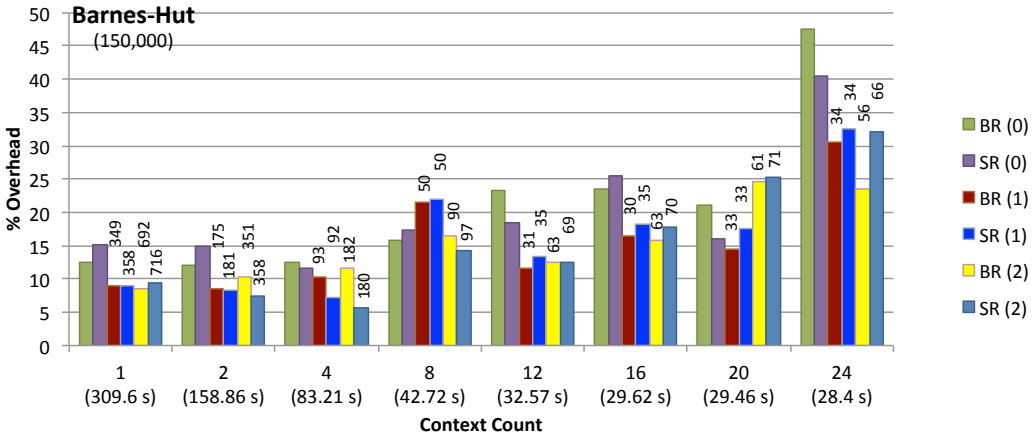


Figure 8.10: Barnes-Hut recovery overheads over the baseline execution time (listed under context-count) at different exception rates and context counts. Total tasks in the program are listed under the name. BR(e) = Base Restart, SR(e) = Selective Restart, at e exceptions/s. Total exceptions at the given rate are listed on the bars.

delivered to a thread, based on a number of factors, such as thread scheduling decisions. Signal-blocking makes the setup even more imprecise. Hence our experimental setup does not precisely deliver the injected rate of exceptions. Table 8.9 lists the injected rate of exceptions (column 2) for the different programs (column 1), and the actual achieved rate of exceptions at the different context counts (columns 4 to 11), for the two types of recovery systems (column 3).

Refer to Figures from 8.10 to 8.15. The *BR(e)* and *SR(e)* bars show overheads at *e* exceptions/sec for each context count. When compared to overheads at 0 exceptions/sec, they denote the restart penalty (the *BR(0)* and *SR(0)* bars are redrawn in these graphs). The baseline execution time in seconds (wall-clock time, including all file I/O operations) for each context count is listed underneath. The total number of exceptions handled are listed on top of the bars. Each graph title gives the program name and the total number of tasks it executes. Program characteristics influence the restart penalty. Results show that when BR incurs a relatively large restart penalty, SR can reduce it, especially at larger context counts, as predicted by our model (Section 8.3).

Barnes-Hut (Figure 8.10) and Black-Scholes (Figure 8.11), due to their relatively small-sized tasks, can easily tolerate exception rates of 1/sec and 2/sec in both runtimes. In both programs, a

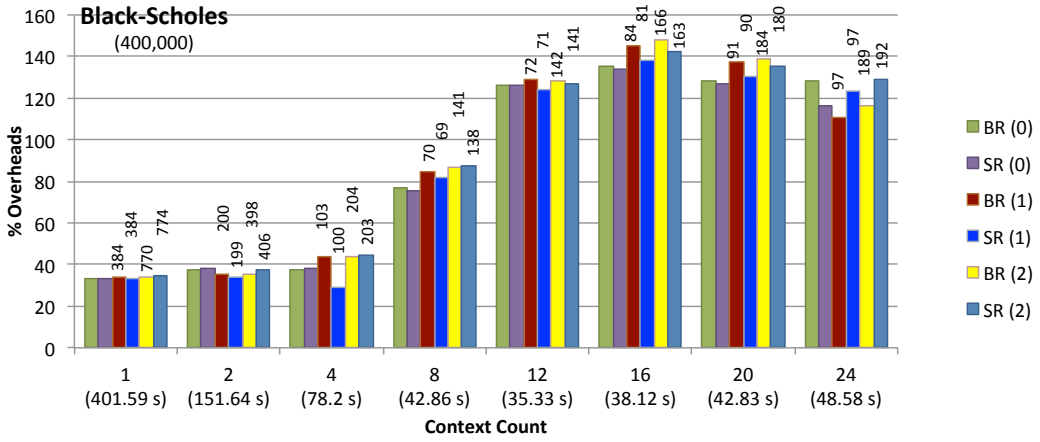


Figure 8.11: Black-Scholes recovery overheads over baseline execution time (listed under context-count) at different exception rates and context counts. Total tasks in the program are listed under the name. BR(e) = Base Restart, SR(e) = Selective Restart, at e exceptions/s. Total exceptions at the given rate are listed on the bars.

total of only a few thousand computations get discarded, and hence both BR and SR yield similar execution times, in general, across all contexts. The restart penalty can be up to ~20% (Black-Scholes, at 16 contexts). The large number of tasks in Black-Scholes, and their checkpointing (the most among all programs) impede the scalability, creating an effect of relatively large total checkpoint penalty. This is true of both the recovery systems. But since the task size is small, work lost at recovery is low, and hence the BR restart penalty is low. Since the checkpoint penalty dominates the restart penalty, SR is unable to do much better than BR for a given exception rate and context count. Further, increasing the exception rate from 1/sec to 2/sec has little impact on the total restart penalty due to the small task sizes.

Pbzip2 (Figure 8.12), Swaptions (Figure 8.13), and RE (Figure 8.14) present a different scenario, due to their relatively large, small number of total tasks, and little checkpointing. This effectively creates a relatively small total checkpoint penalty, but a larger BR restart penalty since more work is lost at recovery. While both BR and SR start out with similar execution times for one and two contexts, SR performs consistently better as the exception rate grows, especially at higher context counts (e.g., the rightmost two bars for each context count–lower is better). In the case of BR,

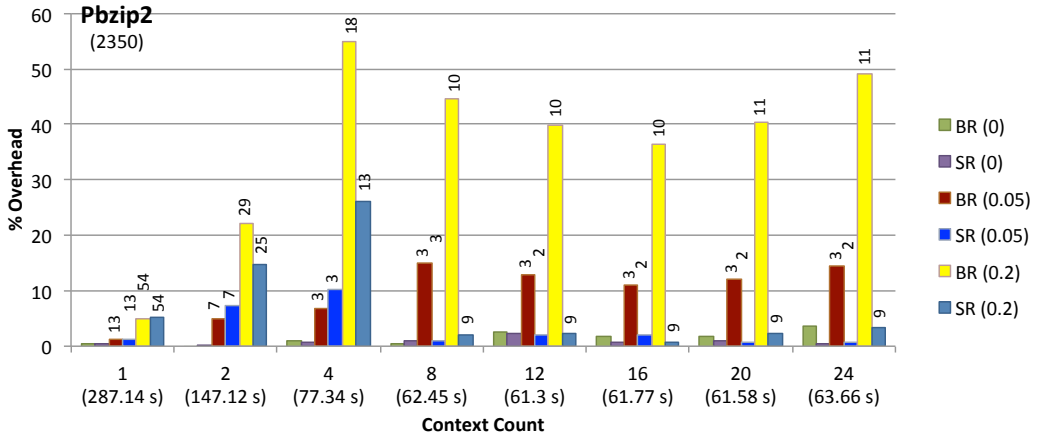


Figure 8.12: Pbzip2 recovery overheads over baseline execution time (listed under context-count) at different exception rates and context counts. Total tasks in the program are listed under the name. BR(e) = Base Restart, SR(e) = Selective Restart, at e exceptions/s. Total exceptions delivered are listed on the bars.

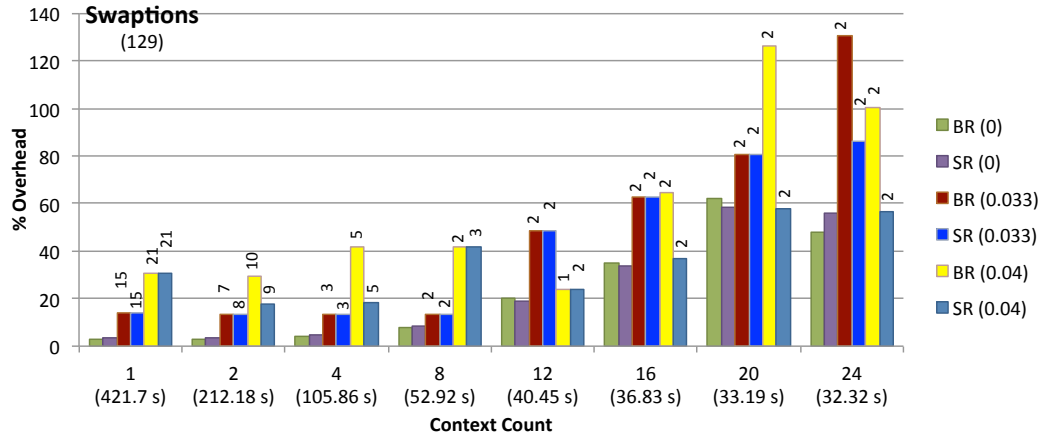


Figure 8.13: Swaptions recovery overheads over baseline execution time (listed under context-count) at different exception rates and context counts. Total tasks in the program are listed under the name. BR(e) = Base Restart, SR(e) = Selective Restart, at e exceptions/s. Total exceptions at the given rate are listed on the bars.

the problem is compounded since the increased execution time increases the exposure to more exceptions. Note that, in general, for the three programs (Pbzip2, Swaptions, and RE), while *BR(e)* grows, *SR(e)* remains relatively constant, close to *SR(0)* and *BR(0)* for a given context count. These

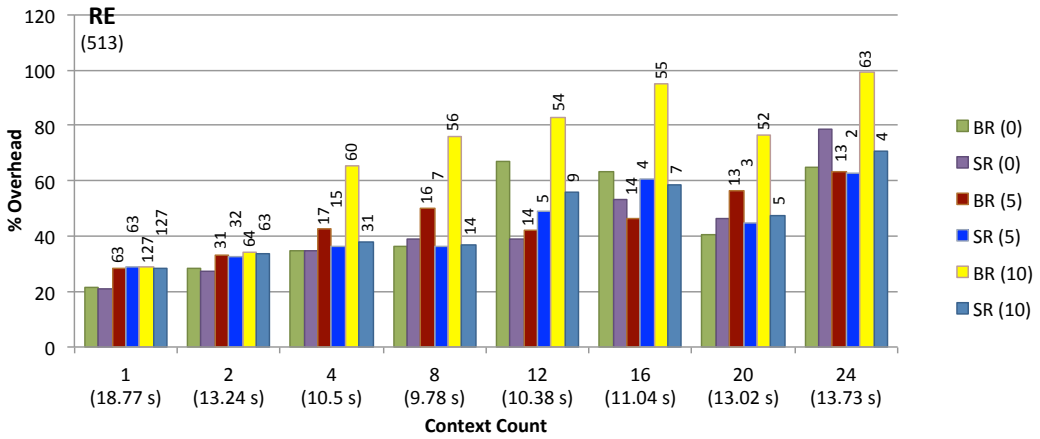


Figure 8.14: RE recovery overheads over baseline execution time (listed under context-count) at different exception rates and context counts. Total tasks in the program are listed under the name. BR(e) = Base Restart, SR(e) = Selective Restart, at e exceptions/s. Total exceptions at the given rate are listed on the bars.

results highlight selective restart's ability to handle more exceptions by reducing the restart penalty.

Whereas the other programs comprised fixed-size tasks, Histogram creates only as many tasks as contexts, and hence its task size grows inversely with the number of contexts. As contexts decrease, the total tasks decrease. Hence the total checkpoint penalty decreases, while the BR restart penalty grows (Figure 8.15). The program fails to complete at exception intervals shorter than the growing task sizes (shown as ∞ in the graph). We present data for histogram at 5 and 10 exceptions/sec. SR fares better at these rates, although for small number of contexts, one and two, the task sizes are too large even for it to complete.

Exception-tolerance Scalability.

As the exception rate increases, for a given task size, the restart penalty grows. Beyond a certain exception rate, BR fails to complete the program. Figure 8.16 shows an example. It plots the execution time of Pbzip2 and Swaptions at exception rates varying from 0.03/sec to 2/sec, on 20 contexts. The execution time grows until BR fails to complete Pbzip2, at 0.71 exceptions/sec , and Swaptions at 0.06 exceptions/sec. SR, however, maintains a stable execution time for the ranges

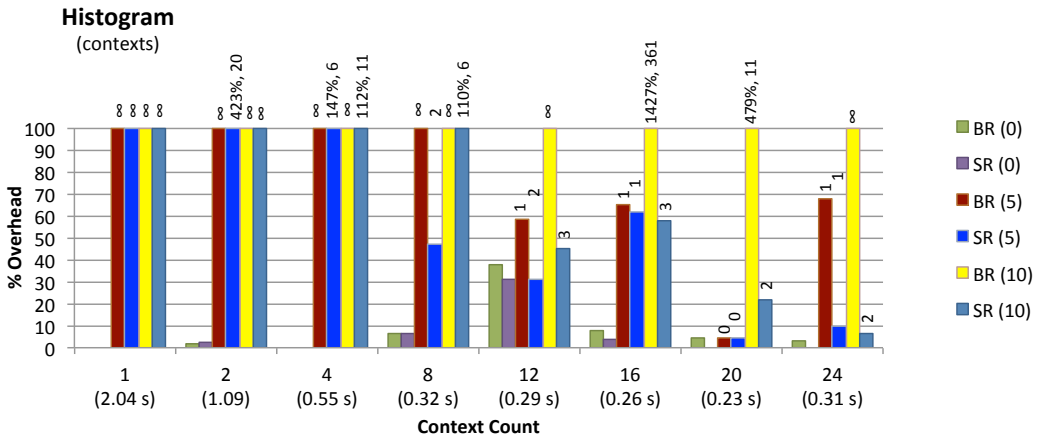


Figure 8.15: Histogram recovery overheads over baseline execution time (listed under context-count) at different exception rates and context counts. Total tasks in the program are listed under the name. BR(e) = Base Restart, SR(e) = Selective Restart, at e exceptions/s. Total exceptions at the given rate are listed on the bars. Overheads larger than the graph scale are listed on the bars, followed by total exceptions; ∞ exceptions => ∞ overheads.

shown. Similar trend holds for the other programs, and at other context counts.

Next, we stressed the two recovery systems to test their exception tolerance limits. Figures 8.17(a) and 8.17(b) plot the exception rates (X axis) for BR and SR, respectively, and the execution time (Y axis) of Pbzip2 for 1 to 24 contexts. The restart penalty, and hence the execution time increases with the exception rate until the *tipping rate*, also tabulated separately in Figure 8.18, when the program cannot be completed. For BR (Figure 8.18, 2nd column) this point is around a single exception rate for all contexts, ~1/sec (0.67 to 1.12), as expected from our analysis ($e \leqslant 1/t$). The tipping rate shifts to higher exception rates with increase in the number of contexts for SR (Figure 8.18, 3rd column), 6.8 to 71.43 exceptions/sec, going from 2 to 24 contexts, also as expected ($e \leqslant n/t$), thus validating selective restart's efficacy. Also note that for $n = 1$ context, both SR and BR have the same tipping rate, (~1/sec), as also predicted by the analysis.

In summary, the above results, based on a real machine, demonstrate GPRS's viability. They show selective restart's tolerance to high exception rates where the conventional CPR would fail. As future computers provide more contexts and exceptions increase, either due to hardware failures

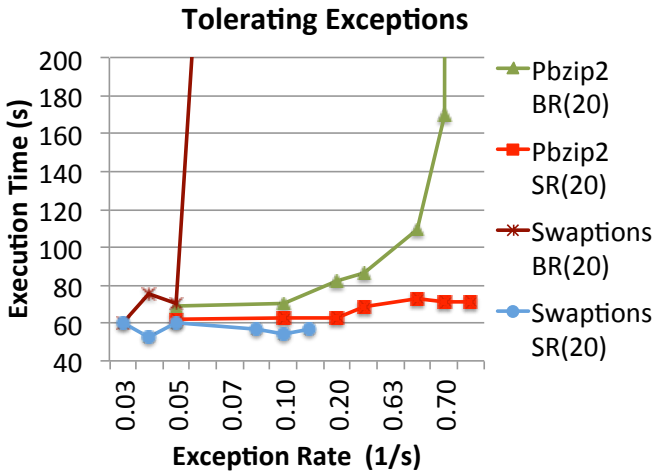


Figure 8.16: Exception-tolerance of Base Restart and Selective Restart at different exception rates for Pbzip2 and Swaptions at 20 contexts.

or new resource management techniques, GPRS can provide a low-overhead approach to handle exceptions.

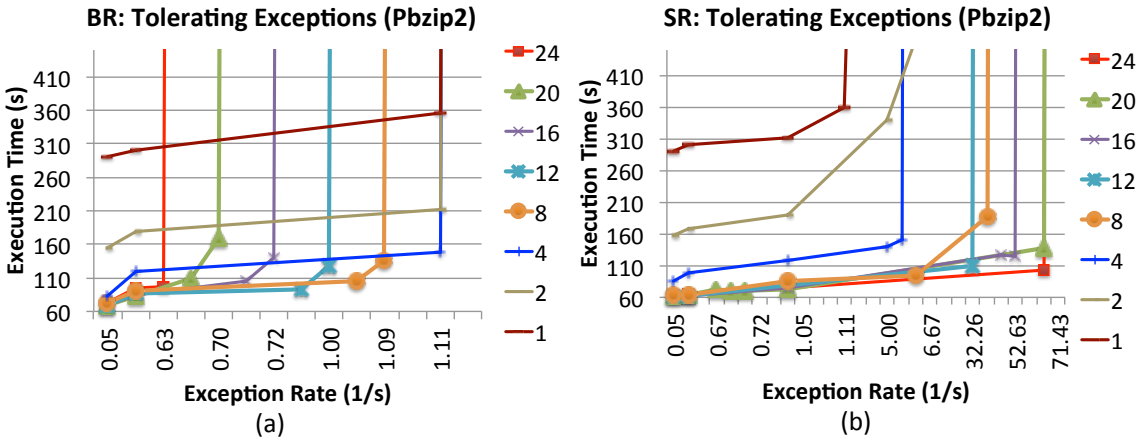


Figure 8.17: Exception-tolerance of Base Restart and Selective Restart at different exception rates for Pbzip2. (a) BR Pbzip2, from 1 to 24 contexts. (b) SR Pbzip2, from 1 to 24 contexts.

Pbzip2 Tipping Rate		
# Contexts	BR faults/s	SR faults/s
24	0.67	71.43
20	0.71	71.43
16	0.73	55.56
12	1.05	33.33
8	1.1	50
4	1.12	5.88
2	1.12	6.8
1	1.12	1.14

Figure 8.18: Exception rate at which Pbzip2 does not complete, for contexts 1 to 24.

8.7 Related Work

We are unaware of another proposal that achieves globally-precise restartable execution of an ordered multiprocessor program, like GPRS. However different proposals, mostly focused on multithreaded programs, support the different capabilities addressed by us.

Hardware and software CPR systems have been traditionally used to enable restartable execution [5, 33, 62, 176]. The different proposals trade off system complexity with the various overheads. CPR schemes may be employed within [120] or without [114] the user programs, in hardware [5] and/or in software [50]. System-level checkpointing [114], implemented outside the program, is oblivious to program operations and simply copies modified memory locations to create a checkpoint. On the other hand, application-level checkpointing [120] relies on the user to checkpoint from within the program. Hybrid schemes that combine the strengths of the two have also been proposed [33]. Our work relies on application-level checkpointing. Use of CPR to provide QoS guarantees in cloud-computing type environments can also be easily envisioned [189]. CPR schemes may also be used to debug programs [44].

Our Base Restart implementation is the most similar to the CPR scheme. Almost all of the above proposals require program execution to pause while its state is begin restored. Selective Restart in GPRS selectively restarts only the affected computations while allowing other computations to

proceed.

Our prior work also achieves Selective Restart-like capability, but in multithreaded programs, by totally ordering the execution [80] and using design features similar to those presented in this chapter.

8.8 Summary

Ordered execution can be beneficial beyond just programming. In this chapter we presented ordered execution's utility in recovering from exceptions that can be handled only after the execution has advanced past the point where the exception occurred. Handling such exceptions requires unwinding the execution to a point before it was corrupted, and resuming again from that point with the correct state. The availability of an ordered view of the execution simplifies this process and reduces overheads. Nondeterministic execution, due to the lack of a precise expected flow of execution, is not easy to roll back and resume.

The ability to pause the execution at a precisely desired point and inspect its state, has utility beyond what we have studied in this dissertation. Ordered execution enables this ability.

9 Related Work

> *...we [the Moderns] are like dwarves perched on the shoulders of giants [the Ancients], and thus we are able to see more and farther than the latter. And this is not at all because of the acuteness of our sight or the stature of our body, but because we are carried aloft and elevated by the magnitude of the giants.*
>
> — Bernard of Chartres (circa 1100)

Speeding up programs by executing coarse-grained computations concurrently on multiple resources has been a topic of research since early days of computing [75]. This effort has received renewed and more intense attention recently. Over three hundred proposals, spanning software and hardware, have been made to simplify multiprocessor programming.

Most related proposals abandon order between computations to exploit the parallelism, although different proposals abandon order to different extent. In contrast, this dissertation introduces methods to express parallelism as ordered programs, and introduces techniques to perform parallel, yet ordered, execution. Other proposals neither perform ordered execution of ordered programs, nor propose such comprehensive techniques to exploit parallelism. Moreover, other proposals seldom apply themselves to such a wide range of algorithms or to different aspects of system use.

We compare and contrast the work in this dissertation with related work along two main dimensions: the conceptual approach (Section 9.1) and the techniques employed (Section 9.2). We also briefly compare the properties of ordered execution identified by us with properties exhibited by other execution models.

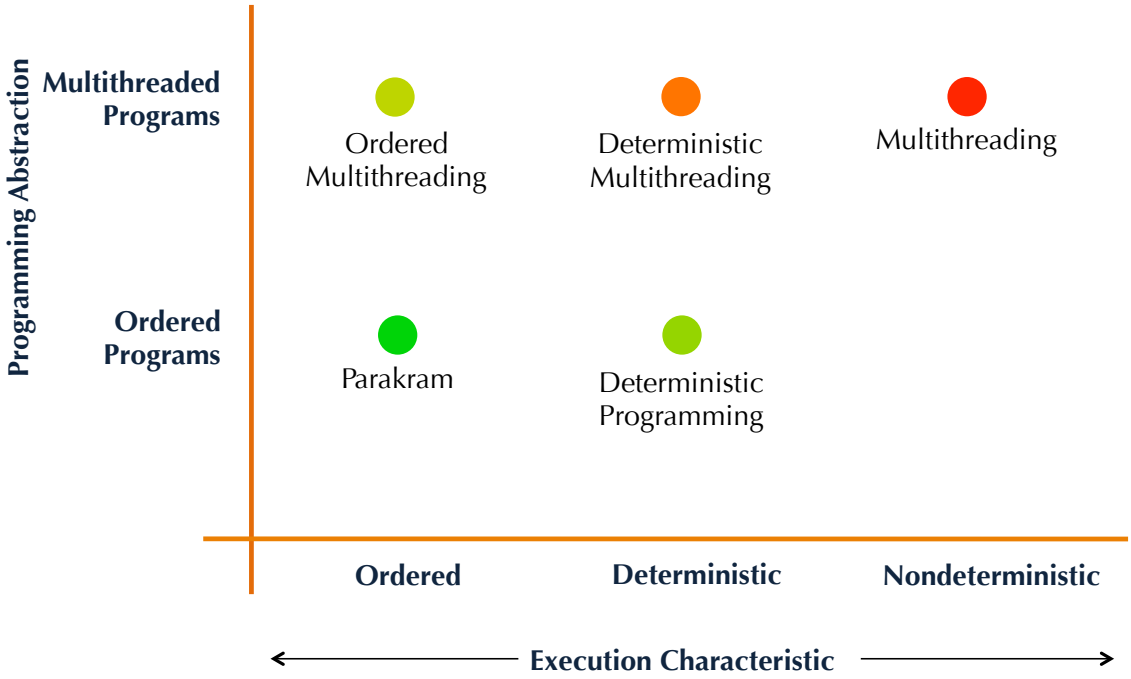


Figure 9.1: Categorizing multithreading programming abstractions. Proposals are grouped based on the programming model they use and the execution characteristic of the model.

9.1 Approach to Multiprocessor Programming

To compare different approaches, we broadly classify them using Figure 9.1. The proposals are categorized along two axes. The first, plotted on the Y axis, depicts the programming abstraction employed by the proposal: ordered and multithreaded. The second, plotted on the X axis, depicts the execution characteristic: ordered, deterministic, and nondeterministic. Parakram is also plotted on the graph. We highlight the different categories in our classification with the help of a few prominent examples. Table 9.2 summarizes the comparison based on the properties of program execution.

Multithreading.

On the top right of the classification graph, we first plot the classic *Multithreading* approach. We include well-known APIs, e.g., Pthreads [1] and MPI [70], among others, in this category. Multi-

Property	Multithreading	Deterministic Multithreading	Ordered Multithreading	Deterministic Programming	Parakram
Deterministic	✗	✓	✓	✓	✓
Interruptible	✗	✗	✓	✗	✓
Intuitive	✗	✗	✗	✓	✓
Ordered	✗	✗	✓	✗	✓

Table 9.2: Key differences between Parakram and other multiprocessor programming methods based on program execution properties.

threading approaches rely on nondeterministic, multithreaded programs. Instructions within a computation are ordered, but may not be across computations. In Multithreading approaches, programmers must ensure that parallel computations are independent, and dependent computations are serialized. Multithreading is a double-edged sword: it gives the utmost flexibility to express parallelism and control the execution, but complicates programming and system use in the process.

Over time many proposals have increasingly simplified various aspects of Multithreaded programming, but without fundamentally giving up nondeterminism. Some permit programmers to provide high level directives to a runtime system. For example, OpenMP [48] provides compiler directives to simplify the expression of fork-join parallelism. More recent proposals, such as Cilk [71] and TBB [151], simplify expression of certain parallelism patterns, such as recursive decomposition and pipeline parallelism. These proposals may also improve execution efficiency by automatically balancing the system load. Parakram requires and provides no special support for specific patterns.

Transactional Memory (TM) has also been proposed to ease Multithreaded programming by not requiring explicit synchronization between parallel computations [86]. TM-based systems provide declarative interface to designate independent computations as transactions. TM ensures Atomicity and Exclusion for these computations. X10 [39] and Habanero [38] combine many features of Pthreads, OpenMP, TBB, Cilk, and Transactional Memory. They too can ensure Atomicity and Exclusion in user-designated regions of the code. But the programmer still needs to coordinate the control-flow between dependent computations. In comparison, Parakram programs are SAFE, and require no explicit coordination.

Galois [142] provides high-level programming constructs, e.g., system-defined set containers and associated functions similar to containers in C++, that are well-suited for graph algorithms. Like Parakram, Galois also relies on user annotations. Galois uses a TM-like speculative scheme to parallelize the execution. Hence it too ensures the Atomicity and Exclusion properties. Galois also permits the programmer to indicate whether conflicting tasks must be executed in the program order, providing deterministic execution in such cases. If not so indicated, the execution is nondeterministic.

Others have proposed compiler-assisted analysis to simplify programming by guaranteeing safety properties, such as data race-freedom and atomicity [13, 69, 92]. In another approach, a programming framework assisted by tools extracts parallelism from sequential code [31]. However, it assumes a three-phase dependence pattern in loops and performs non-deterministic speculative execution of programs.

Multithreading proposals, whether basic or programmer-friendly, ultimately execute programs nondeterministically. Moreover, proposals may simplify one aspect of programming but not another, and may introduce new issues. For example, Cilk can simplify divide-and-conquer type algorithms, but only if the parallel tasks are independent. TM can eliminate explicit synchronization, but may now require the programmer to reason the specific TM's execution semantics, which can also be non-trivial [86, 121]. Galois provides high-level constructs, but they are helpful only in a certain class of problems. Hence in general, programming remains complex. In contrast, Parakram consistently provides the simpler, ordered programming and ordered execution interface, analogous to the well-understood sequential paradigm.

Deterministic Multithreading.

In the second group we include the *Deterministic Multithreading* approaches. Like Multithreading, they start with multithreaded programs, but introduce determinism during their execution [16, 18, 19, 21, 46, 51, 52, 97, 115, 135, 137]. They partially order the computations, in addition to the instructions within the computation, by ordering all accesses to a given datum. The order is

repeatable, and hence they partially eliminate nondeterminism. However, these approaches can penalize the performance [20], sometimes severely, and may require complex hardware support [51, 52, 97]. Moreover, the introduced order can be arbitrary and system-specific. Hence these proposals, except deterministic Galois [135], are not portable

Some proposals may use a program-agnostic method to order the computations [18, 19, 51, 52, 97, 135]. Hence, although the execution is repeatable, it may not be predictable. Other proposals order the execution at synchronization points in the program [21, 115, 137]. Their execution may be predictable, but to what degree, has not be studied.

In general, it is unclear how effective Deterministic Multithreading is in practice. Although their proponents claim benefits, we are unaware of a proposal that has been actually applied to simplify system use. In contrast, Parakram's execution is deterministic, interruptible, intuitive, and portable. We have shown its utility in simplifying programming and exception recovery.

A set of proposals closely related to Deterministic Multithreading fall in the category of **record-and-replay** techniques (not included in the classification graph). Once a multithreaded program has been run, these proposals can ensure that the same execution schedule replays in subsequent runs. They log the sequence of memory interleavings in the original run, and enforce that sequence in the subsequent runs. Some proposals add hardware complexity to the system [96, 132, 133, 187]. Other proposals are software-only solutions, but trade off performance with the replay precision [10, 58, 98, 105, 108, 139, 156, 184]. Parakram naturally admits precise repeatability without the need for explicitly logging and replaying what can be a large number of events.

Ordered Multithreading.

In the third category, we recently proposed the *Ordered Multithreading* approach [80]. It too starts from multithreaded programs, but provides a totally ordered view of the execution, like Parakram. Like Kendo [137], Grace [21], and DTHREADS [115], it orders the computations at synchronization points, but instead of a partial order, it introduces a total order. We showed that the total order helps in exception recovery, similar to Parakram. However, the order is neither intuitive nor portable, and

performance can suffer in some cases.

Deterministic Programming.

In the fourth category of our classification, *Deterministic Programming*, we group approaches that also perform deterministic execution, but start from ordered programs [2, 9, 26, 34, 63, 73, 87, 89, 141, 146, 154, 166, 179, 183]. Instead of an arbitrary order, these proposals enforce a user-provided order. The order may be specified explicitly [2, 34, 87, 166, 179] or implicitly through the program's text [9, 63, 73, 89, 141, 146, 154, 183]. Further, these proposals use the order to automate the parallelization, relieving the programmer from explicitly coordinating the execution. Since the order is derived from the program, it is portable and more intuitive. Hence these proposals claim simplicity over multithreaded programs. Although most proposals are based on software runtime systems, some hardware proposals have also been made [67], and some are compiler-assisted [26, 63].

Deterministic Programming is appealing, but whereas some proposals show it performs well [9, 141, 154], others have shown otherwise [88]. Further, our analysis shows that these approaches may not match multithreaded programs (Multithreading) in expressing and exploiting parallelism. Moreover, the order, although not arbitrary, is deterministic only for accesses to a given datum. As we showed in Chapter 2, in practice deterministic order alone is insufficient to simplify system use. Software runtime-based Deterministic Programming proposals are closest to Parakram. We elaborate on these proposals further.

Parakram supports a wide range of parallelism patterns, and bridges the performance gap between ordered and multithreaded programs. The closely related Deterministic Programming proposals, summarized in Table 9.3, lack in one aspect or another. In general they lack Parakram's expressiveness, performance benefits, and the ordered appearance. For example, Parakram supports arbitrary dependences between nested tasks, others don't. They either don't support nesting, or only conservatively [63, 100, 154], or permit only independent nested tasks [183]. None supports lazy dependence management, unlike Parakram. Only Jade [154] supports fine-grain concurrency

Proposal	**Comp. Indep.**	**Dep. Analysis**	**Patterns**						**Dep. Mgmt.**	**Exc., I/O**
			Data	D.set	UE Disc.	UE ordering				
			Dyn.	D.-dep.	Nested	Task Dep.	Nest Dep.	Granularity		
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
Jade [154]	No	Dynamic	No	No	Yes	Conservative	Coarse	Fine	Eager	No
SMPSs [141]	No	Dynamic	No	No	No	Limited	No	Coarse	Eager	No
Galois [142]	No	Dynamic	Yes	Yes	No	Limited	No	Coarse	Eager	No
PROMETHEUS [9]	Yes	Dynamic	No	No	No	Limited	No	Coarse	Eager	No
DPJ [26]	No	Static	No	No	No	Limited	No	Coarse	Eager	No
OoOJ [100], DOJ [63]	No	Stat.+Dyn.	Yes	No	Yes	Conservative	Coarse	Coarse	Eager	No
US [183]	Yes	Dynamic	No	No	Yes	Restricted	No	Coarse	Eager	No
S.Dataflow [76]	Yes	Dynamic	No	No	No	Limited	No	Coarse	Eager	N/Y
KDG [89]	No	Dynamic	Yes	Yes	No	Limited	No	Coarse	Eager	No
Parakram	**Yes**	**Dynamic**	**Yes**	**Yes**	**Yes**	**Arbitrary**	**Fine**	**Fine**	**Eager, Lazy**	**Yes**
Multithreading	Yes	-	Yes	Yes	Yes	Arbitrary	Fine	Fine	Eager Lazy	-

Table 9.3: Ordered proposals and their key capabilities. (1) Proposals; (2) Compiler independence; (3) Dependence analysis; (4) Dynamic data structure updates in concurrent UEs; (5) Data-dependent datasets. (6) Nested UEs; (7) Task dependence types: conservative for nested tasks, restricted for nested tasks, or limited in general; (8) Dependence granularity between nested UEs; (9) Granularity of UE coordination; (10) Dependence management. (11) Runtime support for exceptions and I/O. Capabilities of Multithreading are listed for comparison.

management between dependent tasks and Galois [142] supports data-dependent datasets, like Parakram. Although several perform dataflow execution [63, 100, 141], and Galois uses TM-like speculation, none speculates dependences. TaskSs [67] is a hardware proposal for SMPSs [141], and has similar limitations. No proposal employs the range of parallelization techniques that Parakram employs. Only Parakram handles exceptions and I/O gracefully. The table enumerates other comparisons.

9.2 Techniques

Parakram, like TM, ordered TM [185], and TLS [150, 174], uses speculation and similar mechanisms. TM is nondeterministic, and speculates that tasks are *race-free*. Parakram is ordered and speculates that tasks are *independent*. TLS, usually compiler- or hardware-assisted, is ordered and has been mostly applied to sequential programs. TM, TLS, and Parakram monitor the datasets, in one way or another, to identify conflicts. Parakram obtains the dataset as soon as possible, whereas TM and TLS in most cases obtain them as the tasks execute. This allows Parakram to minimize the loss of parallelism due to conflicts.

TLS and TM (ordered or not) have a limited ability to exploit parallelism past conflicting tasks. TM may apply back-off schemes to reduce the conflicts in the hopes of opportunistically avoiding the data race. TLS relies on order to let the oldest task proceed, but will resubmit the younger tasks for re-execution immediately. By contrast, compiler- and hardware-agnostic Parakram explicitly builds a dataflow graph. It can shelve blocked tasks, freeing resources to exploit latent parallelism in a larger window. A conflicting task is re-executed only after the current precedent tasks have completed, instead of immediate or arbitrarily delayed re-execution.

Parakram requires that programmers declare the datasets. With hardware support, TM and TLS can automatically identify the task datasets. This also allows them to handle what are poorly composed tasks for Parakram, more gracefully.

Many runtime systems, like Cilk, OpenMP, PROMETHEUS, and TBB, employ dynamic load

balancing techniques. Parakram too employs similar schemes, but includes the task order in its decisions.

9.3 Execution Properties

On occasion, systems guarantee properties of their program execution models, like Parakram, providing a framework for users to reason about the execution. TM systems provide Atomicity and Isolation [86]. Atomicity in the SAFE properties of Parakram pertains to visibility of a task's updates to succeeding tasks, whereas TM's Atomicity pertains to "all-or-nothing" execution semantics– either a transaction appears to have executed or not. The Isolation property in TM is analogous to Exclusion defined by us.

Database systems often support the well-known ACID properties: Atomicity, Consistency, Isolation, and Durability [84]. These are more application-oriented, whereas SAFE are more execution-oriented. Atomicity and Isolation in ACID are analogous to Atomicity and Isolation in TM, and compare along the same lines with Atomicity and Exclusion in SAFE. There is no analog of Consistency and Durability, which pertain to the state the database system ensures is visible to the user, in SAFE.

Often database and TM transactions are Serializable, i.e., they appear to execute in some (arbitrary) serial order. Sequentiality in SAFE is a stricter property; tasks appear to execute in the specific serial order defined by the program, and not some arbitrary serial order.

Memory consistency models, which are closer to the program execution, define the order in which memory operations of a multiprocessor program appear to have been performed to the programmer [4, 94]. Sequential consistency, analogous to Serializability, ensures that they appear to be performed in some serial order. It is the strictest among the memory consistency models proposed in most of the literature and used in products. Sequentiality in SAFE is even stricter, due its adherence to the program order.

10 Conclusions and Future Work

We're just starting. It's a bit evolutionary and there are an awful lot of chicken-and-egg problems. You have to have parallel computers before you can figure out ways to program them, but you have to have parallel programs before you can build systems that run them well.

— Chuck Thacker (2010)

Multiprocessors are now ubiquitous, and every major processor vendor continues to have them on its product roadmap. But using multiprocessors is not easy. The prevailing approach to program them is based on the belief that order must be discarded from programs to secure performance. But discarding order diminishes programmability and usability. Hence, in this dissertation we have questioned whether order should be abandoned.

10.1 In the Past

Although parallel processing has been prevalent for several decades, it was largely confined to high-performance computing (HPC) in which performance is paramount. In the past, a few domain experts developed a few program, mostly embarrassingly-parallel, for a small number of parallel systems. Parallel systems have been traditionally designed with the view that ordering impedes performance. For example, memory consistency models forsake sequential consistency, which attempts to order memory accesses by the system's multiple processors, and adopt weaker models to avoid the performance penalty, despite the resulting software complexity [4, 94]. More recently, TM systems have similarly realized the need to order transactions, but not too strictly, to avoid performance penalties [49]. A combination of these factors–relatively few users, programs with

mostly regular parallelism, and the need for uncompromising performance–likely resulted in the multithreaded programming models, which disregarded order, provided basic features, and put much of the control in the user's hands. But over the last decade parallel systems have reached the mainstream, at once exposing parallel programming challenges to multitudinous programmers and to all types of programs, making productivity and system usability no less important than performance.

10.2 In this Work

The thesis of this dissertation is that we must embrace order in multiprocessor programs. Order can make multiprocessors easier to program and use. At the same time, order need not obscure parallelism. Parallelization techniques can be used under the hood to overcome the order, and secure performance. We tested this thesis, and proposed supporting methods and techniques. Much of our proposal has drawn inspiration from sequential programs and the modern microprocessor.

We analyzed the properties of a sequential program's ordered execution on the microprocessor, and studied the benefits of order to the usability of computing systems. We identified four properties that result from ordered execution, Sequentiality, Atomicity, Flow-control independence, and Exclusion, which simplify programming and system use. The conventional, multithreaded parallel programs lack these properties. Based on this analysis we argued for ordered programs and their ordered execution on multiprocessors, analogous to the sequential paradigm, to obtain the attendant benefits.

We proposed that programmers develop parallel algorithms but express them as ordered programs. To compose programs, programmers need reason about only individual computations, but not their dynamic parallel execution. A programmer-oblivious entity would parallelize the program's execution. We stipulated that, in addition to providing the benefits of order, our approach must match multithreaded programs in expressing and exploiting parallelism, and not diminish programmability or usability in other ways.

The proposed ordered approach, Parakram, comprises a programming model and an execution model. We analyzed the common parallelism patterns that arise in multithreaded programs. Parakram's programming model provides a small set of APIs to express these patterns as object-oriented, task-based ordered programs. Programmers annotate the task with local information, its dataset.

Parakram's execution model parallelizes the program's execution using principles that the microprocessor has successfully used. Parakram discovers the program's tasks, and dynamically builds a task dataflow graph using the dynamic objects the tasks access. Parallelism is exploited by executing tasks concurrently in the dataflow order. However, not all parallelism patterns are well-suited for dataflow execution. To realize such patterns without compromising parallelism, Parakram resorts to speculative execution. Despite the parallel, out-of-order, and speculative execution, at any instance the program appears to have executed as per the program order. To achieve this ordered view, Parakram introduces the notion of globally-precise interrupts in multiprocessor programs.

We built a software prototype of Parakram in the form of a C++ runtime library. The library provides the APIs to develop ordered programs, and implements the execution model. The runtime is implemented as a distributed, asynchronous design, and employs non-blocking concurrent data structures and associated mechanisms to orchestrate the execution. The library was used to develop benchmark programs that spanned the parallelism patterns. Programs were run on commodity multiprocessor systems, and the obtained speedups were measured. The program development process and the achieved speedups were tested against the stipulated criteria for success.

Developing ordered programs was far less complicated than multithreaded programs. Ordered programs closely resembled the original sequential programs in structure. Annotating the programs as needed by Parakram APIs entailed providing information that was already used to develop the parallel algorithm. We note that the community at large is coming to a consensus that programmer annotations in multiprocessor programs are acceptable, and may even be essential [90, 107, 164]. Annotating the Parakram code posed only a modest burden. But importantly, to develop and debug the programs we did not have to reason about the dynamic parallel execution. Nor did we have to

reason about the program's interaction with the system's artifacts, such as the memory consistency model. Parakram programs were able to express all of the parallelism patterns that arose in the benchmark programs. We have not yet encountered a pattern that could not be expressed using Parakram. We conclude that ordered programs can be as expressive as multithreaded programs.

The ordering constraint can be overcome to exploit the parallelism in ordered programs in almost all cases. Except for algorithms with very small task sizes, or highly nondeterministic tasks, Parakram matched the performance of the Multithreading methods. In programs with complex inter-task dependences, dataflow execution enabled Parakram to exploit parallelism more naturally than the Multithreading method. In such cases Parakram outperformed the multithreaded program, unless considerably more efforts were to be spent to optimize the multithreaded implementation. In general, the dataflow and speculation principles are able to unearth parallelism, especially latent parallelism past blocked computations, easily. Although in principle multithreaded programs are equally capable of exploiting parallelism, the onus is on the programmer to do so.

Above results notwithstanding, order does come at a price in some cases. Maintaining order can penalize performance in two ways. The first is due to overheads, and the second is due to the ordering constraint itself. Parakram performs additional work to maintain order and marshal the execution. Associated overheads and Parakram's implementation influence the minimum task size needed to achieve speedups. Although the current implementation can be optimized, the minimum task size needed for profitable application of Parakram will likely be larger than the Multithreading method. When task sizes are sufficiently large, which is the more common case in parallel processing, the overheads do not matter.

The ordering constraint poses a more basic limitation when an algorithm exhibits a unique combination of properties. Insisting on order constrains the exploitable parallelism in algorithms that (i) use small, order-agnostic computations, (ii) process data-dependent datasets, and (iii) have ample parallelism, but whose tasks conflict at high rates. In such cases, dataflow and speculation are defeated by the ordering constraint, whereas the Multithreading method can opportunistically exploit parallelism by ignoring order. Although the Multithreading approach outperforms the

ordered approach, it produces nondeterministic results, and of course suffers from the related issues. In such cases, the ordered approach presents a choice to trade off performance for productivity.

We developed an analytical model that captures the theoretical similarities and differences in speedups attained by the ordered execution and the nondeterministic execution of multithreaded programs, for different types of algorithms. The model provides a simple, yet effective tool for a systematic comparison of the performance capabilities of the two approaches.

To test the utility of the ordered approach beyond programmability, we applied Parakram to recover from frequent exceptions. Once again we were able to exploit order to build a system more efficient than the conventional checkpoint-and-recovery method. Parakram's globally-precise interrupts were extended to recover from exceptions, such as soft hardware errors. Parakram's dataflow execution could be leveraged to perform selective restart, which discards only affected tasks, unlike the conventional method which may discard more work. As a result, Parakram's ability to recover from frequent exceptions scaled with the system size, where the conventional method failed altogether.

The Final Analysis.

The conventional wisdom may have been that order obstructs parallelism, and reaping parallelism requires explicitly identifying independent computations. Indeed, parallelism requires that computations be independent. However, this dissertation shows that independence between computations need not be explicitly identified. Computations may be listed in an order, for convenience. But the ordering constraint can be overcome by establishing dependences between computations, and treating the absence of dependence as independence. Once dependences are established, the parallelism can be reaped by avoiding occupying a resource with work that will conflict with other work. Instead, the resource can be used more gainfully by performing work that will not. And if an algorithm has parallelism, then our study shows that non-conflicting work can be found. This process of establishing dependences and orchestrating the resource use, can be automated by first establishing the computations' order and datasets during the execution. The sooner the order and

the datasets are known, the more effective is the approach.

The Final Word.

Based on our experiments and experience with Parakram, we conclude that the ordered approach has met the criteria of simplifying programming and system use, without compromising performance at large. Order, after all, may not have the performance drawbacks that seem to be the conventional wisdom. In conclusion, with the support of time-tested parallelization techniques, ordered programs and their ordered execution may yet prove viable to program multiprocessors.

10.3 In the Future

As this dissertation shows, ordered execution of ordered programs holds promise. But more questions need to be answered to make the approach practical and realize the promise. These questions range from the applications of the approach, to the low-level concurrency mechanisms used in the implementation of the approach. We divide these questions into three broad research topics: application, programmability, and design. The first explores the utility of ordered execution in simplifying system use and design. The second and third enhance the programming and execution models, with the goal of widening Parakram's reach to more types of problems.

10.3.1 Applying Ordered Execution

In this dissertation we applied the ordered approach to a wide range of algorithms and explored its limitations. Given the minimum task size needed by the current design, we were unable to apply it extensively to certain algorithms, such as graph analytics and circuit simulation, that use small task sizes and generate work that has specific scheduling constraints [89]. They are similar to DeMR, but they may additionally constrain the execution order of newly created tasks, e.g., simulating a gate in a circuit only after the simulation time has reached a certain epoch. These algorithms typically use concurrent priority queues in the Multithreading implementation. Of course, before Parakram can be applied to them, its overheads must be first reduced (to be seen below). But say, they can be,

then what additional challenges might we have to overcome? What opportunities might an ordered approach present for such cases? Particularly, how will an efficient concurrent priority queue be implemented in the ordered paradigm?

Yet another useful direction will be to apply Parakram to program heterogeneous systems, such as ones comprising GPGPUs. Current GPGPU programming models [162] are similar to the Multithreading approach. Can we apply an ordered approach to develop GPGPU kernels? In one trial, we applied a simple ordered programming interface to GPGPU computations. The runtime aggregated the same type of computation invoked on different objects, into a container to be delegated to the GPGPU. It automatically managed the transfer and the co-ordination of data objects between the CPU and GPU. This initial work showed encouraging results [79]. This direction gains significance as GPGPUs integrate more tightly with the CPU, presenting an opportunity for finer-grain coordination between tasks intended for the two [30, 155, 168]. How can programmers take advantage of this integration? Our initial work can be enhanced to compose ordered programs comprising a mix of GPGPU and CPU computations. An interesting question to answer will be, if and how will we maintain order, especially when GPGPU kernels execute?

We saw the application of the ordered execution to recover from exceptions in Chapter 8. In practice, recovering from errors spans a wide range of issues, not all of which we considered. Larger systems are more susceptible to errors. Error recovery is a critical issue in HPC systems [36]. It will be worthwhile to apply our approach to a large system, perhaps by pursuing a hierarchical design [45], and address the related challenges. In particular, we hypothesize that knowing the precise inter-task communication can simplify the overall design. Can order help further? A key challenge a large system will pose to the ordered approach is the design of the shared Reorder List structure. Clearly a simple centralized structure will create bottlenecks. A hierarchical design can be explored instead. Such a design will also suit well to domain-oriented recovery [45]. Might there be additional hurdles to the ordered approach? Logging checkpoints also contributes to high overheads in fault-tolerant systems. Ordered execution already permits independent concurrent logging–how does it compare with the traditional methods? Can logging be made more efficient

with the help of order? Further, we took a simplistic view of the stable storage in the GPRS design. How does a more realistic storage impact the GPRS design? Finally, a software-only design may not be robust enough to tackle all types of errors. Some hardware-assist, e.g., to checkpoint data, may make the design more robust. In that case, what would be the right hardware-software design?

Ordered execution may also prove useful in managing system resources. A runtime system that manages the program's execution is in a vantage position to also manage the resources. This capability can be especially useful in a heterogeneous system. The runtime can be made intelligent and system-aware, shielding the programmer from the system details. The runtime can seek to optimize the execution to better utilize resources, or to take advantage of the heterogeneity by dynamically mapping a task to the best-suited, among currently available, resources. Ordered execution can simplify migration of tasks, or even the entire program across systems, or between logical partitions within a system. This capability can be applied to optimize the execution for energy, power, resources, and other metrics.

Order can also be useful to detect faults and make systems secure. Both applications can benefit from the ability to pause and analyze the program's precise state. Redundant execution is a popular technique to detect faults [175]. That multiple executions, spatially or temporally redundant, produce the same results at intermediate points is key to such techniques. Ordered execution and globally-precise interrupts naturally give this capability. Designing, implementing, and analyzing such a system, and comparing it with traditional approaches will make for a useful study. The same capabilities can also be applied to secure systems.

10.3.2 Programmability

In the dissertation we had hypothesized that ordered programs simplify programming. We have provided evidence by way of code examples and our experience. As a next step, a comparative user-study can evaluate this aspect more systematically [157]. The study might consider algorithms of different types, and recruit programmers to compare ordered programming with Multithreading, e.g., using TM. The study should include programming for performance. We have also claimed

that ordered execution improves debuggability. Integrating a gdb-like debugger with the Parakram runtime library could form a part of this project, to evaluate the overall programming ecosystem. A key aspects of integrating the debugger will be to ensure that it provides a "feel" of sequential execution while permitting debugging of the parallel execution [111].

Parakram's current programming APIs were adequate to explore its potential. A simpler interface will promote wider deployment. We believe that annotations that capture the programmer's intent, at least the key aspects, can go a long way in meeting the challenges multiprocessor systems pose. Some of this was evidenced in our work, e.g., we relied on user-specified datasets. How can we minimize the amount of information we seek, and simplify its specification?

As a first step, can we automatically formulate a task's read set, write set, and mod set? This will ease the programmer's burden, as well as eliminate related programming bugs. A compiler can readily generate a task's mod set by using static program slicing [95]. A compiler can also identify a task's read and write sets using static program analysis. It is necessary that the read and write sets be precise (without being incorrect) to avoid creating superfluous inter-task dependences, but also be coarse in granularity to avoid including individual memory locations. Hence the analysis must capture only coarse-grain objects that are truly shared between tasks, which requires high fidelity pointer analysis. But static program analysis can be conservative due to imprecise pointer alias analysis, to the point of obscuring most parallelism, and can also be slow to perform. Tasks that concurrently operate on object fields can also make the analysis conservative. A worthwhile direction will be to combine pointer analysis proposed for multithreaded programs [158] with recent work on pointer analysis [85], and improve its precision using shape analysis [159] and other abstract interpretation techniques [169]. This analysis may be further refined and sped up with the help of dynamic program analysis, similar to some recent work [63]. The dynamic analysis can be more precise and fast, and augment the conservative static analysis at run-time. Interestingly, Parakram's speculation capability may enable aggressive dynamic analysis and ease the pressure on the static analysis.

Next, can we seek programmer's inputs to help in the above automated analysis? As a first step,

language extensions, similar to Jade [154], can be added to identify shared objects and parallel tasks. As a next step, a type and effect system, along the lines of DPJ [26] and TWEJava [92], might be pursued for ordered programs. With the help of programmer annotated data types, a compiler can infer effects that can assist the analysis. This analysis may also be used to provide safety guarantees. Dynamic effect checking may also be combined, to improve precision and still retain strong safety guarantees [116].

10.3.3 Design

Finally, the Parakram design can be enhanced to bring more and more programs, especially ones with small task sizes, within its purview. The design can be enhanced by adding more concurrency mechanisms, and improving the current implementation, perhaps using hardware support.

So far we have taken advantage of certain algorithmic properties, e.g., data-independent datasets, to parallelize the execution. Additional properties of the algorithm, e.g., commutativity, can be exploited to expose higher degrees of parallelism. Others have done so in multithreaded programs [25, 143]. Can the same be done in ordered programs and ordered execution? Maintaining order will be a key challenge, although speculative execution may be leveraged for the purpose. We can exploit the fact that the ordered appearance need be provided only when asked for. We can permit computations that commute to actually violate the order, as long as we can reconstruct the ordered appearance. Note that this is different from dependence speculation presented earlier in the dissertation, which may violate order only temporarily. We hypothesize that programs with small task sizes, like Breadth-first Search and other graph algorithms with similarly small task sizes, will benefit from such a scheme. How will programmers identify the properties to Parakram? What will the performance benefits be?

The software runtime can be optimized by employing lower overhead concurrency mechanisms. The runtime uses three main concurrent data structures: the Reorder List, work deques, and Wait Lists. Their current implementation is better suited for relatively large tasks. For example, the work deques support random task stealing, which helps to balance the system load, an issue that

arises when tasks are large. These data structures can be manipulated concurrently by the runtime from different threads. This requires the use of appropriate synchronization mechanisms, such as FENCEs, which incur overheads. For programs with smaller tasks, different designs may prove better. First, the right design for each of the data structures needs to be chosen [68, 167]. Second, a design that limits concurrent accesses to them to reduce the needed synchronization may be more suitable. In such a design, a single thread may be responsible to operate on the data structures, and possibly on behalf of the other threads. If followed, how will other threads communicate their requests? Will the design reduce overheads? Still, no one design may be suitable for all types of algorithms. An adaptive approach that dynamically switches between designs, based on program properties, may widen Parakram's applicability to more algorithms. How can we identify the properties at run-time? Can this scheme be applied across program phases?

To further reduce overheads and improve performance, hardware support may be added. The communication and synchronization overheads in the runtime can be characterized. Targeted hardware features can be added to reduce them. What additional hardware features might help Parakram achieve its objectives efficiently? Can the caches be used to hold speculative state, but without limiting the amount of state that can be held? Can the characteristics of the Parakram execution model simplify hardware design, e.g., of the memory subsystem [43, 138]? A carefully partitioned hardware-software Parakram design may make for a compelling multiprocessor solution.

Almost all high-productivity parallel programming models are supported by runtime systems. The runtime systems are parallel programs themselves, as is the Parakram runtime. Developing Parakram presented the same set of challenges that we had set out to solve for the general multiprocessor programmer! Yet another interesting direction would be to study whether TM-like hardware can simplify the development of parallel runtime systems. We postulate that a generic TM may not be required, but some basic features, enough to ensure atomicity and isolation of small code regions, may be adequate for expert developers who cannot entirely avoid multithreaded programming.

Bibliography

[1] IEEE Standard for Information Technology-Portable Operating System Interface (POSIX) - Thread Extensions. *IEEE Std 1003.1c-1995*, pages i –22, 1995. doi: 10.1109/IEEESTD.2000.92296.

[2] Axum Programmer's Guide. Microsoft Corporation. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx., 2009.

[3] *Android Developr's Guide*. http://developer.android.com/guide/components/activities.html. Google, 2013.

[4] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996. ISSN 0018-9162. doi: 10.1109/2.546611. URL `http://dx.doi.org/10.1109/2.546611`.

[5] Rishi Agarwal, Pranav Garg, and Josep Torrellas. Rebound: scalable checkpointing for coherent shared memory. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 153–164, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: http://doi.acm.org/10.1145/2000064.2000083. URL `http://doi.acm.org/10.1145/2000064.2000083`.

[6] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3. URL `http://dl.acm.org/citation.cfm?id=290940.290988`.

[7] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70431-5.

[8] Matthew D. Allen. *Data-Driven Decomposition of Sequential Programs for Determinate Parallel Execution*. PhD thesis, University of Wisconsin, Madison, 2010.

[9] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization Sets: A dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 85–96, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504190. URL http://doi.acm.org/10.1145/1504176.1504190.

[10] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629594. URL http://doi.acm.org/10.1145/1629575.1629594.

[11] Ashok Anand, Chitra Muthukrishnan, Aditya Akella, and Ramachandran Ramjee. Redundancy in network traffic: findings and implications. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 37–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6. doi: 10.1145/1555349.1555355. URL http://doi.acm.org/10.1145/1555349.1555355.

[12] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine philosophy and instruction-handling. *IBM J. Res. Dev.*, 11(1):8–24, January 1967. ISSN 0018-8646. doi: 10.1147/rd.111.0008. URL http://dx.doi.org/10.1147/rd.111.0008.

[13] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. Sharc: Checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 149–158, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375600. URL http://doi.acm.org/10.1145/1375581.1375600.

[14] K. Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, March 1990. ISSN 0018-9340. doi: 10.1109/12.48862. URL http://dx.doi.org/10.1109/12.48862.

[15] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, pages 103–108, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0089-6. doi: 10.1145/1866835.1866854. URL http://doi.acm.org/10.1145/1866835.1866854.

[16] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924957.

[17] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806620. URL http://doi.acm.org/10.1145/1806596.1806620.

[18] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736029. URL http://doi.acm.org/10.1145/1736020.1736029.

[19] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924956.

[20] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: how well does it actually pound nails? In *The 2nd Workshop on Determinism and Correctness in Parallel Programming*, WODET '11, 2011.

[21] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640096. URL http://doi.acm.org/10.1145/1640089.1640096.

[22] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640096. URL http://doi.acm.org/10.1145/1640089.1640096.

[23] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[24] Filip Blagojevic, Costin Iancu, Katherine Yelick, Matthew Curtis-Maury, Dimitrios S. Nikolopoulos, and Benjamin Rose. Scheduling dynamic parallelism on accelerators. In *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, pages 161–170, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-413-3. doi: 10.1145/1531743.1531769. URL http://doi.acm.org/10.1145/1531743.1531769.

[25] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 181–192, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145840. URL http://doi.acm.org/10.1145/2145816.2145840.

[26] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640097. URL http://doi.acm.org/10.1145/1640089.1640097.

[27] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-oriented analysis and design with applications, third edition*. Addison-Wesley Professional, third edition, 2007. ISBN 9780201895513.

[28] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005. ISSN 0272-1732. doi: 10.1109/MM.2005.110. URL http://dx.doi.org/10.1109/MM.2005.110.

[29] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54 (5):67–77, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941507. URL http://doi.acm.org/10.1145/1941487.1941507.

[30] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, 2012. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2012.2.

[31] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: 10.1109/MICRO.2007.35. URL http://dx.doi.org/10.1109/MICRO.2007.35.

[32] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 235–247, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: 10.1145/1024393.1024421. URL http://doi.acm.org/10.1145/1024393.1024421.

[33] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Recent advances in checkpoint/recovery systems. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 282–282, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://dl.acm.org/citation.cfm?id=1898699.1898802.

[34] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, August 2010. ISSN 1058-9244. doi: 10.1155/2010/521797. URL http://dx.doi.org/10.1155/2010/521797.

[35] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, November 2009. ISSN 1094-3420. doi: 10.1177/1094342009347767. URL http://dx.doi.org/10.1177/1094342009347767.

[36] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014. ISSN 2313-8734. URL http://superfri.org/superfri/article/view/14.

[37] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 198–209, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-5585-8. doi: 10.1109/HPCA.2013.6522319. URL http://dx.doi.org/10.1109/HPCA.2013.6522319.

[38] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. doi: 10.1145/2093157.2093165. URL http://doi.acm.org/10.1145/2093157.2093165.

[39] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852. URL http://doi.acm.org/10.1145/1094811.1094852.

[40] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: 10.1145/1073970.1073974. URL http://doi.acm.org/10.1145/1073970.1073974.

[41] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM. ISBN 0-89791-989-0. doi: 10.1145/277651.277696. URL http://doi.acm.org/10.1145/277651.277696.

[42] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, pages 274–280, New York, NY, USA, 1993. ACM. ISBN 0-89791-582-8. doi: 10.1145/160985.161150. URL http://doi.acm.org/10.1145/160985.161150.

[43] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4566-0. doi: 10.1109/PACT.2011.21. URL http://dx.doi.org/10.1109/PACT.2011.21.

[44] Jong-Deok Choi and Janice M. Stone. Balancing runtime and replay costs in a trace-and-replay system. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD '91, pages 26–35, New York, NY, USA, 1991. ACM. ISBN 0-89791-457-0. doi: http://doi.acm.org/10.1145/122759.122761. URL http://doi.acm.org/10.1145/122759.122761.

[45] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 58:1–58:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389075.

[46] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522735. URL http://doi.acm.org/10.1145/2517349.2522735.

[47] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 141–150, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-8186-0861-7. URL http://dl.acm.org/citation.cfm?id=52400.52417.

[48] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998. ISSN 1070-9924. doi: 10.1109/99.660313. URL http://dx.doi.org/10.1109/99.660313.

[49] Luke Dalessandro, Michael L. Scott, and Michael F. Spear. Transactions as the foundation of a memory consistency model. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *DISC*, volume 6343 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2010. ISBN 978-3-642-15762-2. URL http://dblp.uni-trier.de/db/conf/wdag/disc2010.html#DalessandroSS10.

[50] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: http://doi.acm.org/10.1145/1815961.1816026. URL http://doi.acm.org/10.1145/1815961.1816026.

[51] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 85–96, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508255. URL http://doi.acm.org/10.1145/1508244.1508255.

[52] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: A relaxed consistency deterministic computer. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 67–78, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950376. URL http://doi.acm.org/10.1145/1950365.1950376.

[53] Melvil Dewey. *A Classification and subject index for cataloguing and arranging the books and pamphlets of a library*. Amherst, Massachusetts, 1876.

[54] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-44624-9, 978-3-540-44624-8. doi: 10.1007/11864219_14. URL http://dx.doi.org/10.1007/11864219_14.

[55] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975. URL http://doi.acm.org/10.1145/360933.360975.

[56] Jason Duell, Paul Hargrove, and Eric Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. White paper, Future Technologies Group, 2003.

[57] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844148. URL http://doi.acm.org/10.1145/844128.844148.

[58] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4. doi: 10.1145/1346256.1346273. URL http://doi.acm.org/10.1145/1346256.1346273.

[59] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3802-0. doi: 10.1109/ICPP.2009.64. URL http://dx.doi.org/10.1109/ICPP.2009.64.

[60] EC2. Amazon EC2 Spot Instances. http://aws.amazon.com/ec2/spot-instances/, 2009.

[61] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Syst. J.*, 41(1):111–125, January 2002. ISSN 0018-8670. doi: 10.1147/sj.411.0111. URL http://dx.doi.org/10.1147/sj.411.0111.

[62] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002. ISSN 0360-0300. doi: 10.1145/568522.568525. URL http://doi.acm.org/10.1145/568522.568525.

[63] Yong hun Eom, Stephen Yang, James C. Jenista, and Brian Demsky. DOJ: Dynamically Parallelizing Object-oriented Programs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 85–96, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145828. URL http://doi.acm.org/10.1145/2145816.2145828.

[64] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL http://dl.acm.org/citation.cfm?id=956417.956571.

[65] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151008. URL http://doi.acm.org/10.1145/2150976.2151008.

[66] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4924-8. doi: 10.1109/MICRO.2012.48. URL http://dx.doi.org/10.1109/MICRO.2012.48.

[67] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R.M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 89–100, Dec 2010. doi: 10.1109/MICRO.2010.13.

[68] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145849. URL http://doi.acm.org/10.1145/2145816.2145849.

[69] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349328. URL http://doi.acm.org/10.1145/349299.349328.

[70] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[71] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277725. URL http://doi.acm.org/10.1145/277650.277725.

[72] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[73] David Gay, Joel Galenson, Mayur Naik, and Kathy Yelick. Yada: Straightforward parallel programming. *Parallel Comput.*, 37(9):592–609, September 2011. ISSN 0167-8191. doi: 10.1016/j.parco.2011.02.005. URL http://dx.doi.org/10.1016/j.parco.2011.02.005.

[74] Jeff Gilchrist. Parallel data compression with bzip2. URL http://compression.ca/pbzip2/.

[75] Stanley Gill. Parallel programming. *The Computer Journal*, 1(1):2–10, April 1958. ISSN 0010-4620 (print), 1460-2067 (electronic). doi: http://dx.doi.org/10.1093/comjnl/1.1.2.

[76] Gagan Gupta and Gurindar 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. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155628. URL http://doi.acm.org/10.1145/2155620.2155628.

[77] Gagan Gupta and Gurindar S. Sohi. Does order constrain parallelism? In *Submitted to the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, 2015.

[78] Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi. The road to parallelism leads through sequential programming. In *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*, 2012. URL https://www.usenix.org/conference/hotpar12/road-parallelism-leads-through-sequential-programming.

[79] Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi. Program execution on multicore and heterogeneous systems. CS Technical Report TR1804-2. Department of Computer Sciences, University of Wisconsin-Madison, 2013.

[80] Gagan Gupta, Srinath Sridharan, and Gurindar 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. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594306. URL http://doi.acm.org/10.1145/2594291.2594306.

[81] Meeta S. Gupta, Jude A. Rivers, Pradip Bose, Gu-Yeon Wei, and David Brooks. Tribeca: design for pvt variations with local recovery and fine-grained adaptation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 435–446, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669168. URL http://doi.acm.org/10.1145/1669112.1669168.

[82] M.S. Gupta, K.K. Rangan, M.D. Smith, Gu-Yeon Wei, and D. Brooks. DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 381 –392, feb. 2008. doi: 10.1109/HPCA.2008.4658654.

[83] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924952.

[84] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL http://doi.acm.org/10.1145/289.291.

[85] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190075.

[86] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010. doi: 10.2200/S00272ED1V01Y201006CAC011. URL http://www.morganclaypool.com/doi/abs/10.2200/S00272ED1V01Y201006CAC011.

[87] Tim Harris, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Mark Moir. Constrained data-driven parallelism. In *5th USENIX Workshop on Hot Topics in Parallelism, HotPar'13, San Jose, CA, USA, June 24-25, 2013*, 2013. URL https://www.usenix.org/conference/hotpar13/workshop-program/presentation/harris.

[88] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 3–12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941557. URL http://doi.acm.org/10.1145/1941553.1941557.

[89] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. Kinetic dependence graphs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 457–471, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694363. URL http://doi.acm.org/10.1145/2694344.2694363.

[90] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 417–428, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254114. URL http://doi.acm.org/10.1145/2254064.2254114.

[91] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[92] Stephen T. Heumann, Vikram S. Adve, and Shengjie Wang. The tasks with effects model for safe concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 239–250, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442540. URL http://doi.acm.org/10.1145/2442516.2442540.

[93] Vincent Heuveline, Sven Janko, Wolfgang Karl, Bjorn Rocker, and Martin Schindewolf. Software transactional memory, OpenMP and Pthread implementations of the conjugate gradients method - a preliminary evaluation. In *10th International Meeting on High-Performance Computing for Computational Science*, VECPAR 2012, pages 300–313. Springer Berlin Heidelberg, 2012. URL http://dx.doi.org/10.1007/978-3-642-38718-0_30.

[94] M.D. Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31 (8):28–34, Aug 1998. ISSN 0018-9162. doi: 10.1109/2.707614.

[95] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.53994. URL http://doi.acm.org/10.1145/53990.53994.

[96] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. doi: 10.1109/ISCA.2008.26. URL http://dx.doi.org/10.1109/ISCA.2008.26.

[97] Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood. Calvin: Deterministic or not? free will to choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 333–334, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-9432-3. URL http://dl.acm.org/citation.cfm?id=2014698.2014870.

[98] Jeff Huang, Peng Liu, and Charles Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 207–216, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882323. URL `http://doi.acm.org/10.1145/1882291.1882323`.

[99] ITRS. Semiconductor Industry Association (SIA), Design, International Roadmap for Semiconductors, 2011 edition. `http://public.itrs.net`.

[100] James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. OoOJava: Software out-of-order execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 57–68, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941563. URL `http://doi.acm.org/10.1145/1941553.1941563`.

[101] R.M. Karp and R.E. Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[102] Kurt Keutzer and Tim Mattson. A design pattern language for engineering (parallel) software. `http://parlab.eecs.berkeley.edu/wiki/_media/patterns/`.

[103] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: Merging the plpp and opl projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0127-5. doi: 10.1145/1953611.1953620. URL `http://doi.acm.org/10.1145/1953611.1953620`.

[104] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65 –76. IEEE, april 2009. doi: 10.1109/ISPASS.2009.4919639.

[105] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 155–166, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0038-4. doi: 10.1145/1811039.1811057. URL `http://doi.acm.org/10.1145/1811039.1811057`.

[106] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL `http://doi.acm.org/10.1145/359545.359563`.

[107] James Larus. Programming multicore computers: Technical perspective. *Commun. ACM*, 58 (5):76–76, April 2015. ISSN 0001-0782. doi: 10.1145/2742908. URL `http://doi.acm.org/10.1145/2742908`.

[108] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, April 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.1676929. URL `http://dx.doi.org/10.1109/TC.1987.1676929`.

[109] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.180. URL http://dx.doi.org/10.1109/MC.2006.180.

[110] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 140–151, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1572-2. doi: 10.1145/2486159.2486174. URL http://doi.acm.org/10.1145/2486159.2486174.

[111] Yossi Lev and Mark Moir. Debugging with transactional memory. In *1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT '06, 2006.

[112] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7): 18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940.

[113] Xuanhua Li and Donald Yeung. Exploiting application-level correctness for low-cost fault tolerance. *J. Instruction-Level Parallelism*, 10, 2008.

[114] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[115] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043587. URL http://doi.acm.org/10.1145/2043556.2043587.

[116] Yuheng Long, Yu David Liu, and Hridesh Rajan. Intensional effect polymorphism. In *Proceedings of the 29th European Conference on Object-oriented Programming*, ECOOP'15, July 2015.

[117] A. Lovelace. Notes to the translation of sketch of the analytical engine invented by charles babbage esq. by L. F. menabrea, of turin, officer of the military engineers. *Scientific Memoirs, Selections from The Transactions of Foreign Academies and Learned Societies and from Foreign Journals, edited by Richard Taylor, F.S.A.*, Vol III London: 1843, Article XXIX, 1843.

[118] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346323. URL http://doi.acm.org/10.1145/1346281.1346323.

[119] D. Manivannan and Mukesh Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):703–713, July 1999. ISSN 1045-9219. doi: 10.1109/71.780865. URL http://dx.doi.org/10.1109/71.780865.

[120] Daniel Marques, Greg Bronevetsky, Rohit Fernandes, Keshav Pingali, and Paul Stodghil. Optimizing checkpoint sizes in the c3 system. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10 - Volume 11*, IPDPS '05, pages 226.1–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2312-9. doi: http://dx.doi.org/10.1109/IPDPS.2005.316. URL `http://dx.doi.org/10.1109/IPDPS.2005.316`.

[121] Milo M. K. Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006. URL `http://dblp.uni-trier.de/db/journals/cal/cal5.html#MartinBL06`.

[122] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004. ISBN 0321228111.

[123] Michael D. McCool, Arch D. Robison, and James Reinders. *Structured parallel programming patterns for efficient computation*. Elsevier/Morgan Kaufmann, Waltham, MA, 2012. ISBN 9780124159938 0124159931.

[124] Paul E. McKenney. Selecting locking primitives for parallel programming. *Commun. ACM*, 39(10):75–82, October 1996. ISSN 0001-0782. doi: 10.1145/236156.236174. URL `http://doi.acm.org/10.1145/236156.236174`.

[125] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: 10.1109/MICRO.2007.8. URL `http://dx.doi.org/10.1109/MICRO.2007.8`.

[126] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL `http://doi.acm.org/10.1145/103727.103729`.

[127] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228568. URL `http://doi.acm.org/10.1145/2228360.2228568`.

[128] Samuel P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool, 2012.

[129] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors, *IISWC*, pages 35–46. IEEE, 2008. ISBN 978-1-4244-2778-9. URL `http://dblp.uni-trier.de/db/conf/iiswc/iiswc2008.html#MinhCKO08`.

[130] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992. ISSN 0362-5915. doi: 10.1145/128765.128770. URL http://doi.acm.org/10.1145/128765.128770.

[131] P. Monteiro, M. P. Monteiro, and K. Pingali. Parallelizing Irregular Algorithms: A Pattern Language. In *18th Conference on Pattern Languages of Programs, Portland, USA*, 2011.

[132] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. doi: 10.1109/ISCA.2008.36. URL http://dx.doi.org/10.1109/ISCA.2008.36.

[133] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508254. URL http://doi.acm.org/10.1145/1508244.1508254.

[134] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: efficient handling of i/o in highly-available rollback-recovery servers. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 200–211, Feb 2006. doi: 10.1109/HPCA.2006.1598129.

[135] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic Galois: On-demand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 499–512, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541964. URL http://doi.acm.org/10.1145/2541940.2541964.

[136] Shuou Nomura, Matthew D. Sinclair, Chen-Han Ho, Venkatraman Govindaraju, Marc de Kruijf, and Karthikeyan Sankaralingam. Sampling + DMR: Practical and low-overhead permanent fault detection. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 201–212, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: http://doi.acm.org/10.1145/2000064.2000089. URL http://doi.acm.org/10.1145/2000064.2000089.

[137] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, March 2009. ISSN 0362-1340. doi: 10.1145/1508284.1508256. URL http://doi.acm.org/10.1145/1508284.1508256.

[138] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 325–334, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3. doi: 10.1145/2464996.2465443. URL http://doi.acm.org/10.1145/2464996.2465443.

[139] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629593. URL `http://doi.acm.org/10.1145/1629575.1629593`.

[140] Y. N. Patt, W. M. Hwu, and M. Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, MICRO 18, pages 103–108, New York, NY, USA, 1985. ACM. ISBN 0-89791-172-5. doi: 10.1145/18927.18916. URL `http://doi.acm.org/10.1145/18927.18916`.

[141] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008. doi: 10.1109/CLUSTR.2008.4663765.

[142] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993501. URL `http://doi.acm.org/10.1145/1993498.1993501`.

[143] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 1–11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993500. URL `http://doi.acm.org/10.1145/1993498.1993500`.

[144] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007. ISBN 0521880688, 9780521880688.

[145] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 111–122, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1605-X. URL `http://dl.acm.org/citation.cfm?id=545215.545228`.

[146] Hridesh Rajan. Building scalable software systems in the multicore era. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 293–298, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0427-6. doi: 10.1145/1882362.1882423. URL `http://doi.acm.org/10.1145/1882362.1882423`.

[147] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, March 2000. ISSN 0164-0925. doi: 10.1145/349214.349241. URL `http://doi.acm.org/10.1145/349214.349241`.

[148] Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 249–262, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429100. URL http://doi.acm.org/10.1145/2429069.2429100.

[149] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 1-4244-0804-0. doi: 10.1109/HPCA.2007.346181. URL http://dx.doi.org/10.1109/HPCA.2007.346181.

[150] Lawrence Rauchwerger and David Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. doi: 10.1145/207110.207148. URL http://doi.acm.org/10.1145/207110.207148.

[151] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[152] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 179–188, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: 10.1145/1088149.1088173. URL http://doi.acm.org/10.1145/1088149.1088173.

[153] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level checkpointing for the native posix thread library for linux. In *The International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, Jun 2006.

[154] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, May 1998. ISSN 0164-0925. doi: 10.1145/291889.291893. URL http://doi.acm.org/10.1145/291889.291893.

[155] Phil Rogers, Ben Sander, Yeh-Ching Chung, Ben Gaster, and Wen Mei Hwu. Heterogeneous system architecture (hsa): Architecture and algorithms tutorial. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, Los Alamitos, CA, USA, 2014. IEEE Computer Society Press.

[156] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999. ISSN 0734-2071. doi: 10.1145/312203.312214. URL http://doi.acm.org/10.1145/312203.312214.

[157] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693462. URL http://doi.acm.org/10.1145/1693453.1693462.

[158] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 77–90, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301645. URL http://doi.acm.org/10.1145/301618.301645.

[159] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292552. URL http://doi.acm.org/10.1145/292540.292552.

[160] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993518. URL http://doi.acm.org/10.1145/1993498.1993518.

[161] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Innovations in internetworking. chapter Design and Implementation of the Sun Network Filesystem, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988. ISBN 0-89006-337-0. URL http://dl.acm.org/citation.cfm?id=59309.59338.

[162] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0131387685, 9780131387683.

[163] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 202–211, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319863. URL http://doi.acm.org/10.1145/319838.319863.

[164] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? *Commun. ACM*, 58(5):77–86, April 2015. ISSN 0001-0782. doi: 10.1145/2742910. URL http://doi.acm.org/10.1145/2742910.

[165] Mohit Saxena, Mehul A. Shah, Stavros Harizopoulos, Michael M. Swift, and Arif Merchant. Hathi: durable transactions for memory using flash. In Shimin Chen and Stavros Harizopoulos, editors, *DaMoN*, pages 33–38. ACM, 2012. ISBN 978-1-4503-1445-9. URL http://dblp.uni-trier.de/db/conf/damon/damon2012.html#SaxenaSHSM12.

[166] Jan Schafer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer, 2010. ISBN 978-3-642-14106-5. URL http://dblp.uni-trier.de/db/conf/ecoop/ecoop2010.html#SchaferP10.

[167] Michael L Scott. *Shared-Memory Synchronization*. Morgan and Claypool Publishers, USA, 2013. ISBN 9781608459575.

[168] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008. ISSN 0730-0301. doi: 10.1145/1360612.1360617. URL http://doi.acm.org/10.1145/1360612.1360617.

[169] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *MEMOCODE*, pages 39–48. IEEE Computer Society, 2007. ISBN 1-4244-1050-9. URL http://dblp.uni-trier.de/db/conf/memocode/memocode2007.html#SenS07.

[170] Akhter Shameem and Jason Roberts. *Multi-Core Programming*. Intel Press, USA, 2006.

[171] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.*, 37:562–573, May 1988. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.4607. URL http://dx.doi.org/10.1109/12.4607.

[172] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, Dec 1995. ISSN 0018-9219. doi: 10.1109/5.476078.

[173] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. Comput.*, 39(3):349–359, March 1990. ISSN 0018-9340. doi: 10.1109/12.48865. URL http://dx.doi.org/10.1109/12.48865.

[174] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95:ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[175] Daniel J. Sorin. Fault tolerant computer architecture. *Synthesis Lectures on Computer Architecture*, 4(1):1–104, 2009. doi: 10.2200/S00192ED1V01Y200904CAC005. URL http://www.morganclaypool.com/doi/abs/10.2200/S00192ED1V01Y200904CAC005.

[176] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1605-X. URL http://dl.acm.org/citation.cfm?id=545215.545229.

[177] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 337–348, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3. doi: 10.1145/2464996.2465016. URL http://doi.acm.org/10.1145/2464996.2465016.

[178] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 169–180, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594292. URL http://doi.acm.org/10.1145/2594291.2594292.

[179] Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. AEMinium: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Trans. Program. Lang. Syst.*, 36(1):2:1–2:42, March 2014. ISSN 0164-0925. doi: 10.1145/2543920. URL http://doi.acm.org/10.1145/2543920.

[180] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005. ISSN 1542-7730. doi: 10.1145/1095408.1095421. URL http://doi.acm.org/10.1145/1095408.1095421.

[181] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005. ISSN 1542-7730. doi: 10.1145/1095408.1095421. URL http://doi.acm.org/10.1145/1095408.1095421.

[182] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[183] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:1–11, 2011. ISSN 1089-795X. doi: http://doi.ieeecomputersociety.org/10.1109/PACT.2011.7.

[184] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950370. URL http://doi.acm.org/10.1145/1950365.1950370.

[185] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370836. URL http://doi.acm.org/10.1145/2370816.2370836.

[186] Juraj Wiedermann. Parallel turing machines. Technical Report RUU-CS-84-11. Dept. of Computer Science, University of Utrecht, 1984.

[187] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM. ISBN 0-7695-1945-8. doi: http://doi.acm.org/10.1145/859618.859633. URL `http://doi.acm.org/10.1145/859618.859633`.

[188] Guihai Yan, Xiaoyao Liang, Yinhe Han, and Xiaowei Li. Leveraging the core-level complementary effects of pvt variations to reduce timing emergencies in multi-core processors. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 485–496, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1816025. URL `http://doi.acm.org/10.1145/1815961.1816025`.

[189] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. *System*, (January):236–243, 2010. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5557987`.

[190] Jin Zhou and Brian Demsky. Bamboo: A data-centric, object-oriented approach to many-core software. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 388–399, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806640. URL `http://doi.acm.org/10.1145/1806596.1806640`.