Parallel Computing Research at Illinois
The UPCRC Agenda

Department of Computer Science
Department of Electrical and Computer Engineering
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

PARALLEL
@ILLINOIS

PARALLEL @ILLINOIS

Parallel Computing Research at Illinois
The UPCRC Agenda

Department of Computer Science
Department of Electrical and Computer Engineering
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
November 2008

Sarita V. Adve
Vikram S. Adve
Gul Agha
Matthew I. Frank
María Jesús Garzarán
John C. Hart
Wen-mei W. Hwu
Ralph E. Johnson
Laxmikant V. Kale
Rakesh Kumar
Darko Marinov
Klara Nahrstedt
David Padua
Madhusudan Parthasarathy
Sanjay J. Patel
Grigore Rosu
Dan Roth
Marc Snir
Josep Torrellas
Craig Zilles

Parallel@Illinois: Pioneering and Promoting Parallel Computing

Illinois history in parallel computing stretches more than 40 years. From the first academic parallel supercomputer, the ILLIAC IV started in 1964, to today's work to install the first petascale computer, Blue Waters, Illinois has defined the landscape of parallel computing. Contributions from past and current Illinois faculty include:

- ILLIAC
- CEDAR
- Illinois Cache Coherence (MESI) Protocol
- OpenMP
- MPI
- Path Pascal
- Actors
- Java and C++ memory models
- Compilers and auto-parallelization techniques—Analyzer, Polaris, Parafrase, IMPACT, LLVM
- Race detection techniques
- Parallel runtime systems—Chare Kernel, Charm++
- IBM/DARPA PERCS—a precursor to IBM's Power 7
- AVIO to detect atomicity violations
- Parallel programming patterns

Today, parallel computing at Illinois spans mobile and desktop client computing at the Universal Parallel Computing Research Center (UPCRC—**www.upcrc.illinois.edu**), cloud computing, and petascale supercomputing. The work includes long-term research, education, and one-of-a-kind testbed installation, including the world's largest academic supercomputer.

Excellence in parallel computing at Illinois is a collaborative effort that transcends disciplinary boundaries. The Coordinated Science Laboratory, Department of Computer Science, Department of Electrical and Computer Engineering, and the National Center for Supercomputing Applications (NCSA) work together and in conjunction with faculty and researchers across the entire campus, including chemists, biologists, artists, structural engineers, humanities experts, economists, and more.

Parallel@Illinois (**www.parallel.illinois.edu**) is the collective representation of Illinois' current efforts in parallel computing research and education. These include:

- Universal Parallel Computing Research Center
- Blue Waters
- Gigascale Systems Research Center
- Cloud Computing Testbed
- CUDA Center of Excellence
- Institute for Advanced Computing Applications and Technologies
- OpenSPARC Center of Excellence

This paper represents the vision and research agenda of the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The UPCRC gratefully acknowledges the sponsorship of Intel and Microsoft Corporations for much of this research.

We would like to thank Bill Gropp, Mike Heath, and Jim Larus for their comments on this paper.

We welcome your feedback on this paper. Comments and suggestions can be sent to community@upcrc.illinois.edu or posted at www.upcrc.illinios.edu/whitepaper.html. An electronic version of this paper is also available at this URL.

Copyright © 2008 The Board of Trustees of the University of Illinois. All Rights Reserved.

Contents

1 Introduction 6
2 Overview 9
2.1 Applications and Patterns 9
2.2 Disciplined Programming Models 11
2.3 Development and Execution Environments 13
3 Applications and Patterns 16
3.1 Applications 16
3.1.1 Dynamic Virtual Environments 16
3.1.2 Natural Language Processing 17
3.1.3 Tele-immersive Environments 18
3.2 Patterns 18
4 Disciplined Programming Models 21
4.1 Disciplined Shared Memory 21
4.2 Parallel Operators 25
4.3 Metaprogramming and Autotuning 25
4.4 Domain-Specific Environments 27
4.5 Actors 28
5 Development and Execution Environments 31
5.1 Translation Environment 31
5.2 Runtime System 32
5.3 Hardware Architecture 34
5.3.1 The Bulk Multicore: High-Performance, Programmable Shared Memory 35
5.3.2 DeNovo: Rethinking Hardware for Disciplined Parallelism 36
5.4 Formal Methods and Tools to Check Correctness 38
6 Conclusions 42
References 43

1 Introduction

For many decades, Moore's law has bestowed a wealth of transistors that hardware designers and compiler writers have converted to usable performance, without changing the sequential programming interface. The main techniques for these performance benefits—increased clock frequency and smarter but increasingly complex architectures—are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster. This historic shift to multicore processors changes the programming interface by exposing parallelism to the programmer, after decades of sequential computing.

Parallelism has been successfully used in many domains such as high performance computing (HPC), servers, graphics accelerators, and many embedded systems. The multicore inflection point, however, affects the entire market, particularly the client space, where parallelism has not been previously widespread. Programs with millions of lines of code must be converted or rewritten to take advantage of parallelism; yet, as practiced today, parallel programming for the client is a difficult task performed by few programmers. Commonly used programming models are prone to subtle, hard to reproduce bugs, and parallel programs are notoriously hard to test due to data races, non-deterministic interleavings, and complex memory models. Mapping a parallel application to parallel hardware is also difficult given the large number of degrees of freedom (how many cores to use, whether to use special instructions or accelerators, etc.), and traditional parallel environments have done a poor job virtualizing the hardware for the programmer. As a result, only the highest performance seeking and skilled programmers have been exposed to parallel computing, resulting in little investment in development environments and a lack of trained manpower. There is a risk that while hardware races ahead to ever-larger numbers of cores, software will lag behind and few applications will leverage the potential hardware performance.

Moving forward, if every computer will be a parallel computer, most programs must execute in parallel and most programming teams must be able to develop parallel programs, a daunting goal given the above problems. Illinois has a rich history in parallel computing starting from the genesis of the field and continues a broad research program in parallel computing today [1]. This program includes the Universal Parallel Computing Research Center (UPCRC), established at Illinois by Intel and Microsoft, together with a sibling center established at Berkeley. These two centers are focused on the problems of multicore computing, especially in the client and mobile domains.

This paper describes the research vision and agenda for client and mobile computing research at Illinois, focusing on the activities at UPCRC (some of which preceded UPCRC).

Given the long history of parallel computing, it is natural to ask whether the challenges we face today differ from those of the past. Compared to the HPC and server markets, the traditional focus of parallel computing research, the client market brings new difficulties, but it also brings opportunities. Table 1 summarizes some of the key differences.

HPC	Server	Client
Few applications	Few subsystems	All applications
Very skilled programmers	Skilled programmers	All programmers
Latency	Throughput	Quality of experience
Deadline always missed	Time to market important	Time to market critical
Small volume	Medium volume	Large volume

TABLE 1 • Client parallel computing versus high performance and server computing.

The parallel client system must enable most programming teams to write "good" parallel code. In contrast to the HPC and server markets, the metrics for "goodness" are more diverse. Client programmers care about many *performance* metrics, including execution time, power consumption, audio and video quality, output accuracy, and availability, all which contribute to the ultimate metric—the quality of user experience. The programmers and system vendors also care about *scalability*—using more cores should improve quality of experience, with no software rewrite. Time to market, and hence software development *productivity*, is paramount. Fortunately, the volume is much larger than the HPC and server markets, affording investments on a scale not previously possible.

A key impediment to all three goals of performance, scalability, and productivity is the lack of high-level parallel programming models. To make parallelism truly universal, we must move beyond current bug-prone parallel programming models to an ecosystem that reduces opportunities for errors, while exploiting the full performance potential of parallelism. We are optimistic that this is possible for two reasons. First, although current practices make parallel programming hard, our view is that this is not a fundamental property of all parallelism. For example, at Illinois, middle school students routinely write parallel programs using the Smalltalk-based graphic language, Squeak, without being aware that parallelism is supposed to be hard [2]. The lesson here is that we need to provide the right form of parallelism for the right problem and the right set of tools to support it. While some parallel codes may inherently require complex, bug-prone interactions among threads, many parallel programs will have interactions with a simple logic and should be provided a programming notation that expresses this logic in a clear way. Our second reason for optimism arises from the observation that the high market volumes on the client side can support multiple programming solutions that can shed the burdens of a one-size-fits-all solution; further, these markets can afford investments in sophisticated development tools and execution environments that can be translated to simpler programming models for the programmer.

Our research agenda therefore centers on the following three themes.

- ***Focus on Disciplined Parallel Programming.***

  Modern sequential programming languages have evolved from unstructured programs with goto statements to structured procedural languages to object-oriented languages. Notions of encapsulation and

modularity make sequential programming easier by promoting a separation of concerns, both in terms of correctness and performance. Further, modern notions of safety simplify programming by restricting the set of possible dynamic interactions. In comparison to these advances in sequential languages, threads programming is still in the "go-to" age [3]. We therefore aim to develop *disciplined parallel programming models and languages, supported by sophisticated development and execution environments,* that will offer the analog of modern sequential programming principles of safety, structure, and separation of concerns.

- ***Multi-Front Attack on Multicore Programming.***

  Illinois has a long history of work on parallelism at all levels from applications and programming abstractions down to hardware architecture, and across a broad spectrum of approaches from explicit high-performance programming to automatic tools that hide the details of parallelism. Our focus on disciplined parallelism is backed up by a broad-based attack that uses every weapon in our arsenal to address this problem. We investigate disciplined explicitly parallel languages, metaprogramming and autotuners, and domain-specific environments. Leveraging our strength in compilers, we aim for a powerful translation environment to exploit information from multiple sources (language level annotations, compiler analyses, runtime, and hardware) at different times in the life of a program. Following decades of parallel architecture research, we propose novel multicore hardware substrates that not only enable performance scalability but also support programmability. Finally, our work with refactoring tools will help move existing code to our new environments, and formal methods based techniques and tools will help ensure correctness.

- ***Human-Centric Vision of Future Consumer Applications.***

  Our work is driven by a foreseeable future where all client applications will be parallel, and the primary consumer feature that will drive the economics of future client software development will be the quality of the human interaction. We are targeting applications that rely on enabling technologies for computer support of social interaction through quantum-leaps in immersive visual realism, reliable natural-language processing, and robust telepresence. Investigating these applications reveals new approaches to parallel pattern analysis, inspires new multicore versions of domain-specific environments, and serves as a testbed for evaluating, refining and ultimately proving our ideas on multicore programming.

2 Overview

This section gives an overview of our agenda for parallel applications and patterns, disciplined parallel programming models, and development and execution environments, summarized in Figure 1. Subsequent sections describe our projects in more detail.

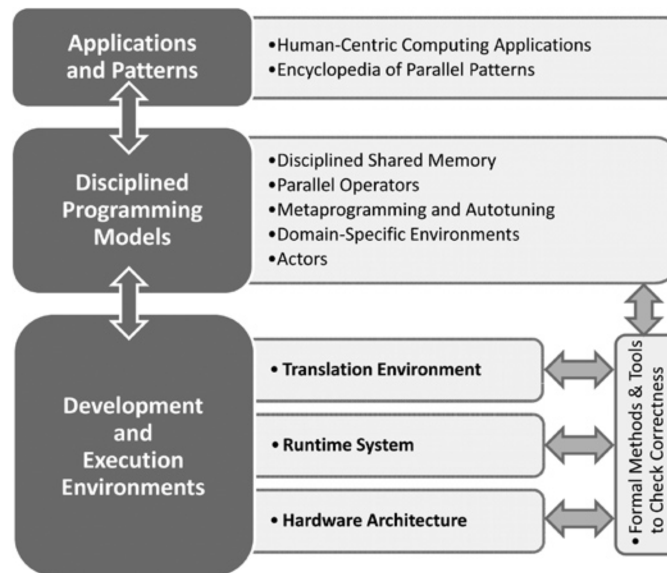

FIGURE 1 • Overview of the UPCRC/Illinois research agenda.

2.1 Applications and Patterns

The long history of parallel computing at Illinois is as much a history of parallel applications as of parallel systems. As we move to parallelism in the client, we continue our previously successful research model of integrated teams of applications and systems researchers. Specifically, the following experiences and observations drive our work on future client applications.

- ***Integrating applications and systems research:*** Systems with new capabilities drive new applications and new applications drive new systems. Parallel computing research at Illinois has repeatedly bootstrapped this cycle with integrated applications and systems research, in areas ranging from conventional high performance computing to more recent efforts in the general purpose GPU (GPGPU) domain. For example, 15% and 20% of the cycles in one year at NCSA and the Pittsburgh Supercomputing Center respectively ran applications (including NAMD [4, 5]) using Charm++ [6], co-developed by one of us. The evolution of Charm++ is a clear demonstration of how application research can drive the language/runtime research agenda, leading to techniques that in turn are useful for a broader set of applications.

Illinois is also recognized as an early innovator and leading center in exploiting the 100+ cores on a GPU for diverse applications ranging from graphics to simulation to more general-purpose domains [7, 8, 9, 10]. Our client computing research agenda continues this tradition. A group of applications researchers from representative client domains are working closely with systems researchers to produce parallelized frameworks that represent key computations in these domains. This process will provide a testbed for and influence our systems research ideas. As the systems ideas mature, the application frameworks will evolve to use them to add new value for the applications and will be disseminated for a wider community of application developers.

- ***Human-centric computing will drive multicore client applications:*** For our research to be relevant, it is imperative that it be driven by the client computing workloads of the future. These workloads will be CPU intensive and will require computing at a fast client rather than in the "cloud." We believe that such workloads abound—many of them will have a central theme of understanding humans and facilitating their interactions with other humans and with large amounts of information. Understanding and reacting to the (human) client's needs in real-time, with enough realism, within the current context, and without sacrificing privacy has the potential to absorb foreseeable client-side performance increases. Society will determine the next generation of killer applications. We aim to accelerate their development by focusing on a few broad domains that we believe encompass the "killer technologies" for them, and for which we have leading in-house expertise.
- ***Parallel patterns as a vocabulary for developers:*** Before programmers start using our programming models and tools, they must think about the parallel algorithms, data structures, and more specifically, parallel *patterns* to use for their applications. We believe a vocabulary of such patterns is essential to educate parallel programmers, both directly and by enabling sharing of experiences. One of us previously co-authored a book on design patterns for object-oriented programming that many believe to be the engine that popularized sequential object-oriented programming [11]. That effort focused on very specific patterns that programmers could directly use to translate to code. We aim to duplicate that success for parallel programming. For a few years, we have been working to catalog parallel patterns through a community effort, previously in the context of high performance computing [12] and now broadened to include client computing. Like the sequential patterns book, we believe that the most value in a parallel patterns catalog will come from cataloging the rich diversity of specific patterns that programmers can directly use in their code. This is in contrast to identifying a few motifs, which is useful as a characterization of important computations but has unclear value in guiding code development.

Motivated by the above observations and experiences, we are engaged in two types of activities:

1) ***Domain-specific frameworks and APIs:*** We have identified three important domains for future killer client applications:

    a. Dynamic virtual environments

    b. Natural language processing

    c. Tele-immersive environments

    These domains cover technologies in graphics, computer vision, and image, video, text, and speech processing. Leading researchers from these areas are collaborating with systems researchers to define and

implement parallelized, high-performance domain-specific frameworks and APIs for key computations in the above domains, using the technologies developed by our systems researchers.

2) ***Parallel Patterns:*** We are engaged in an effort to create an encyclopedia of parallel patterns that encompasses a broad set of programmer experiences. This is necessarily a community effort. It involves collaboration with programmers inside and outside our university and other researchers also working on parallel patterns. To facilitate this collaboration, we plan a workshop series on parallel patterns, based on similar workshops we have organized in the past [12].

2.2 Disciplined Programming Models

Three observations drive our work on disciplined parallel programming models:

- ***Disciplined coordination:*** In a seminal paper [3], Dijkstra argued that goto's make programs hard to comprehend and debug since, when they are used, there is no easy way to describe the mapping from the static structure of the code to the dynamic evolution of the execution. Arbitrarily placed task spawns, unsynchronized conflicting memory accesses or data races, and low-level synchronization operations such as locks, are the equivalent of goto's in the parallel world: it is hard to describe the mapping from the individual executions of threads to the global execution. We seek models that impose structure on parallel control flow and on synchronization. Current language specifications already discourage the use of data races [13, 14], but do not aid the programmer in achieving this goal. A stronger guarantee is *determinism*, which guarantees that for a given input, the program will always produce the same output. This output is the result of an equivalent sequential execution, providing a simple semantic model. This model facilitates code development and debugging, while still exposing to the programmer a parallel performance model. Effectively, deterministic languages can ride on the advances in sequential programming, including safety, modularity, and composability. Many programs, especially a large class of transformative programs, are deterministic; however, current languages do not aid in expressing them in provably deterministic terms. We wish to explore the extent to which language support can be used to guarantee data-race-freedom, determinism, and other higher level coordination structures, in the context of modern sequential programming practices and client applications.
- ***Encapsulation:*** Not all code will be deterministic—reactive programs and some classes of transformative programs require various forms of non-determinism for efficiency. For a disciplined environment, we propose that when non-deterministic behavior is unavoidable, it be made explicit and encapsulated in a way that limits its impact on the rest of the program. Actor models, where the interaction between sequential threads is always explicit, are an example of such an approach that works well for reactive programs. How to best encapsulate non-determinism for transformative programs and how to specify alternate disciplines for non-determinism (other than data-race-free) are research questions we are currently exploring.
- ***Enabling performance through a separation of concerns:*** As mentioned previously, discipline is not an end in itself, but a means to exploiting the potential of parallelism. In another influential paper [15], Dijkstra articulated the notion of separation of concerns that allows reasoning about one property of a program (e.g., correctness) without having to simultaneously reason about another property (e.g., performance). In today's multicore environments, however, reasoning about correctness is intricately tied to reasoning about performance. The complexity of modern hardware means that much of this performance tuning

requires reasoning about complex (and often machine-specific) effects and subtle interactions. A key aspect of our work on disciplined models is to enable programmers to use higher levels of abstraction to specify functionality, while the system exploits this information to use its full flexibility to provide performance. Ideally, we would like to reduce the burden on programmers so they worry only about correctness while the system automatically tunes for performance.

The above observations motivate the following five specific research directions we are currently exploring.

1) ***Languages for disciplined shared memory:*** We are working on a general-purpose disciplined shared-memory language that builds upon modern object-oriented languages with strong safety properties, providing the programmer with all the familiar ease-of-use facilities of modern sequential languages in conjunction with disciplined parallelism. The general philosophy is to use language constructs, potentially backed up by the compiler, formal correctness checking tools, and hardware, to (i) *guarantee* no data races, (ii) *guarantee* determinism where possible, and (iii) where non-determinism is unavoidable, make it explicit and encapsulated with limited impact on the rest of the program [16, 17, 18]. This language would support both task-parallel and flexible data-parallel programming idioms, within a familiar object-oriented language. We aim to separate, to the extent possible, concerns about the semantics of programs from concerns about performance such as optimizations for locality. Such a language would support a progressive refinement of codes for performance tuning that does not affect the program semantics.

2) ***Parallel operators:*** Parallelism is most easily managed when it is encapsulated in operators defined on data aggregates (data parallelism) or simple iteration domains. Parallelism is hidden, thus facilitating program development and mapping to the target machine. Such operators also help raise the level of abstraction and thus have the potential of improving the quality of automatic optimization. Array operations have been used to represent parallel computations practically from the beginning of parallel computing, but modern client applications demand a richer set of array operations and extensions to irregular data structures such as sets and graphs; e.g., [19, 20, 21]. We are working to identify operators that would best serve our application domains. These operators can be added to existing programming languages or to newly developed disciplined shared memory languages. This work bridges the gap between general-purpose programming languages and domain-specific languages.

3) ***Metaprogramming and autotuning:*** The idea with metaprogramming and autotuning is for the programmer to provide options for achieving the required functionality (e.g., different sorting algorithms). The system then automatically finds the best option for the current platform and (possibly) the current input, and generates this (close to) optimal program [22, 23, 24]. This line of work is further facilitated by identifying commonly used high-level data parallel primitive operations (e.g., [19, 20],) and *codelets* that can then be easily autotuned.

4) ***Domain-specific environments (DSEs):*** Domain-specific environments (including languages, libraries, and frameworks) can hide or significantly simplify the use of parallelism through the knowledge of domain information, and offer a high level of abstraction for the domain expert. Our research goal is not to create specific DSEs for each domain per se, but to provide the techniques and tools that domain experts for a wide range of domains can use to build effective parallel DSEs. Two examples we are working on are techniques to translate performance information into domain-specific terms and techniques to simplify the porting and evolution of applications to parallel domain-specific libraries and frameworks.

5) ***Actors:*** In many cases, shared memory and deterministic sequential reasoning are unnatural. Such cases include *programming in the large* such as cloud computing and sensor networks, and reactive programs, where concurrency is part of the problem specification. For such programs, we are exploring the actor model of computation [25]. Fine-grained actor languages have previously been shown to be efficiently implementable on distributed memory architectures [26]. Our focus here will be again driven by the separation of concerns philosophy—how to separate the specification of functional behavior of individual actors and the specification of aggregate multi-actor constraints such as scheduling, synchronization, quality of service, reliability, etc. [27, 28].

In summary, we hypothesize that to achieve the benefits of parallel computing, we will need disciplined programming models. Our research agenda involves an exploration of the appropriate disciplines such as data race freedom, determinism, controlled non-determinism, high-level operators, domain-specific encapsulation, metaprogramming, and actors. Although our eventual goal is for low-level programming to be needed only as a matter of last resort for the most performance-critical operations, realistically, we expect that software using current models and low-level programming will continue to exist for a number of years—our research agenda includes how to support such software with the best performance and with limited impact on correctness of the rest of the code. The fact that our agenda includes multiple approaches should not be surprising—it is analogous to the sequential world where programmers program at different levels of abstraction, including high-level domain-specific languages such as Mathematica, scripting languages such as Python, Java, C#, C++, C, and assembly.

2.3 Development and Execution Environments

The following observations drive our research in development and execution environments.

- ***Supporting discipline:*** New programming models require support from the system. For example, disciplined use of shared memory could benefit from hardware support for race detection [29] and for speculation [30, 31]. To truly separate performance concerns from correctness, and to support portability and scalability, it is likely that the runtime will need to expose a common virtual parallel interface as a target for the compiler and then map that interface to the current hardware at execution time. Autotuning and efficient run-time resource management require better performance sensors and actuators; e.g., for memory subsystem behavior.

- ***Rewarding discipline:*** Our research goes further than providing only the support required to enable our disciplined models. Too often, information available at higher levels of the system stack is lost at the lower levels, forgoing significant opportunities. All of our disciplined programming models naturally capture a rich amount of information about the program. Our research seeks to exploit this information to the fullest so we can reward discipline with robust performance, and foster the adoption of such models. For example, how can compilers use domain information from domain-specific frameworks and library components of a larger full application? How can high-level QoS specifications be used to drive runtime resource management? How can hardware exploit the fact that a program is written in a deterministic language to eliminate unnecessary traffic and complexity implied by current cache coherence protocols? More generally, if most code follows disciplined models, what concurrency model should hardware support?

- ***Facilitating transition to disciplined programs:*** Change is always disruptive, even change for the better. We can accelerate change by lowering its cost. Tools such as refactoring tools and correctness checking tools can make it easier to adopt our techniques; e.g., by enabling *interactive* (semi-automated) conversion of old code into the new models and by making it possible to use the new models with the old. Moreover, architectural support for deterministic replay of parallel programs [32, 33], data-race detection [29] and pervasive program monitoring [34, 35] will facilitate the transition of users to the new environments.
- ***Supporting co-existence:*** New programming models and new runtime environments will need to co-exist with currently available software for many years. Although we want to reward discipline, we cannot punish current software stacks. We therefore need to continue to provide performance and aid programmability with current programming models.

Driven by the above, we are engaged in the following research.

1) ***Translation environment:*** We are working on an ambitious compiler infrastructure to enable our vision of disciplined parallel programming. Our compiler aims to bring together a number of transformations and analyses that exploit novel sources of information at various times in the life of an application, with the following goals: (i) to support explicitly parallel deterministic languages, high-level parallel operators, and domain-specific languages, exploiting the rich information provided by the programmer in these environments; (ii) to support parallelism discovery and speculation when the above information is insufficient or requires runtime speculation for validation; (iii) to appropriately interface with autotuners; and (iv) to support user-driven refactoring tools to enable porting existing sequential and parallel code to the new models.

2) ***Runtime system:*** The runtime system is responsible for virtualizing the (potentially heterogeneous) hardware for the rest of the software stack. Consequently, it is responsible for providing transparent resource management for potentially heterogeneous platforms, achieving required QoS within given physical constraints. Although we present the runtime and hardware as separate entities here, our research views the boundary between these two layers as fuzzy—techniques such as virtual instruction set computing [36, 37] and binary translation can replace a hardware interface with a software interface. We expect that virtualization, portability, and efficient handling of heterogeneity will increasingly require the kind of hardware-software co-design represented by such methods.

3) ***Hardware architecture:*** Our research in hardware architecture focuses not just on performance but on programmability as a first order design objective. We are working on two broad projects. The first project focuses on supporting a flexible substrate with scalable cache coherence, high-performance sequential memory consistency, and an easy-to-use development and debugging environment. The second project rethinks concurrent hardware as a co-designed component of our disciplined programming strategy, both for enforcing the discipline and for exploiting it for simpler and more efficient hardware.

4) ***Formal methods and tools to check correctness:*** We are working on formal methods and tools for assuring correctness in concurrent programs. Our work encompasses the following research directions: (i) to understand high-level synchronization intentions that programmers use to manage concurrency, through the study of concurrency bug databases and user studies, (ii) utilizing the high-level intentions as specifications for testing concurrent programs, in particular to build effective algorithms that avoid enumerating and testing all interleavings, but find errors by testing only those interleavings that violate

these intentions, and (iii) to build mechanisms to sandbox the parts of a program that use undisciplined concurrency mechanisms and interface it with other portions that adhere to disciplined principles, and realize ways to summarize effects of these portions to infer correctness properties of the entire program.

3 Applications and Patterns

3.1 Applications

Our applications thrust has two objectives: (1) to serve as a driver for our work in parallel programming models and environments, and (2) to apply state-of-the-art parallelization techniques to key application domains to accelerate the development of killer applications for multicore client systems. Our focus is on applications that enhance human/human and human/computer interactions, and particularly on the underlying compute intensive algorithms that improve interface intelligence and interactivity. Our current focus is in three broad domains: dynamic virtual environments, natural language processing, and tele-immersive environments.

Broadly, our approach within each domain is to develop a set of key parallelized frameworks that can be used by application developers to develop applications on a larger scale. Such frameworks will include APIs for frequently-used, compute-intensive functionality with extensibility for user-supplied kernels and data structures. The frameworks themselves will utilize technology developed in our other thrusts, such as autotuning and metaprogramming, and benefit from the parallel models and environments we are investigating. The subsequent subsections describe the individual application domains.

3.1.1 Dynamic Virtual Environments

Real-time online virtual worlds (e.g. World of Warcraft and Second Life) have transformed the internet into an immersive space for social interaction, but the visuals and simulations of these online collaborative environments lag well behind those of stand-alone video games. Modern video games (like Halo 3) achieve cinematic photorealism and fluid motion by limiting the set of viewpoints and configurations of the game environment, and precomputing its lighting and animation over this limited set (via, e.g., the GPU-assisted precomputed radiance transfer used in the Xbox 360 [38]), essentially encoding the visuals into a massive high-dimensional multiple-viewpoint movie. This precomputation requires careful gameplay coordination (which makes game development expensive and time consuming), and does not support user interaction (e.g., building in an online virtual environment). The increased computational power of future multicore processors can overcome the need for precomputation, making videogames cheaper and faster to produce and online virtual environments more realistic.

The performance of the rendering and simulation of online virtual worlds and video game environments relies on spatial data structures (SDS) to efficiently manage interactions between objects and neighborhood queries (as demonstrated in Figure 2a), and the fundamental challenge for multicore processing of these visual applications will be the development of efficient parallel algorithms for processing a shared spatial data structure. We and others have developed hierarchical SDS traversal algorithms that avoid conditional program flow for efficient streaming SIMD execution, as demonstrated in Figure 2b [39, 40]. We and others have also utilized parallel "scan" primitives to construct a balanced hierarchical SDS [41, 42]. But to achieve a goal of realistic rendering and animation of fully dynamic user-reconfigurable virtual worlds, we will need new multicore algorithms to efficiently maintain a kinetic SDS with efficient insertion, deletion, motion, and rebalancing.

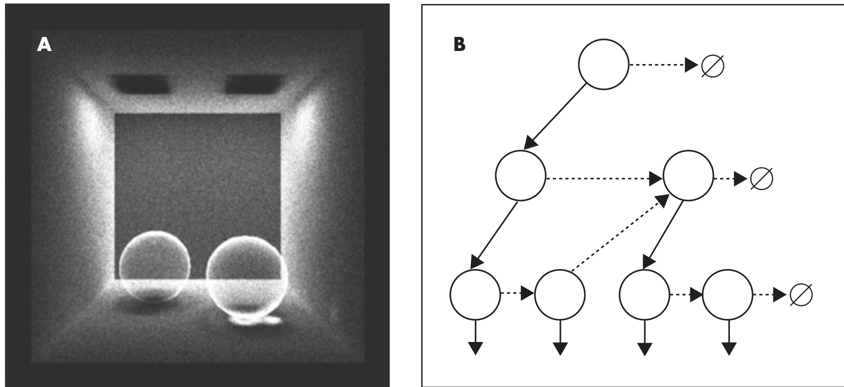

FIGURE 2 • (a) The performance of many visual applications relies on efficient nearest-neighbor queries, e.g. the photon-mapping rendering method determines the light reflected by a surface point by finding the k nearest photons. (b) A binary tree where each node contains two pointers: a "pass" pointer to the first child and a "fail" pointer to the next sibling (or uncle). This binary tree avoids conditional program flow since a node's traversal always follows one of its two nodes, supporting efficient SIMD parallel queries.

A natural application driver for client multicores therefore is a parallel framework for such spatial data structures that provides the applications developer with highly scalable SDS functionality for their applications. The interactivity demands of visual applications have resulted in a wide variety of detailed algorithms and data structures for managing spatial queries. Depending on the spatial data distribution, application, query type and processing architecture, the spatial data structure can be a 3-D grid, either simple or hashed, or a hierarchy, organized as an oct- tree, k-d tree, or BSP tree, and their processing can be organized iteratively or recursively, using SIMD operators, *short stacks*, or tree threading. As such, these domain-dependent application demands will be used to close-the-loop between our work on higher-level parallel pattern analysis and lower-level multicore implementation details.

3.1.2 Natural Language Processing

Recent studies have shown that over 85% of the information that corporations handle is *unstructured,* the vast majority of which is textual. A multitude of techniques has to be used in order to enable intelligent access to this information and to support transforming it to forms that allow sensible use of the information. The fundamental issues that all these techniques have to address is that of semantics—if one wants to access text based on its content (rather than just *words* mentioned in it), there is a need to move toward *understanding* the text at an appropriate level. This may include understanding the topic of the text, the events described in it, the sentiments expressed in it, understanding "who is doing what to whom?", etc.

While there has been huge progress in research in these directions over the last few years, all commercial applications (e.g., Google, Yahoo, Microsoft, and others) make use of very shallow techniques—key words that are present in the text. One of the key reasons for that is computational—supporting semantic analysis of text, even at the level of a single sentence or a single paragraph, may take *seconds* on a single core machine.

Analyzing huge amounts of data, as is required to support better search, information extraction, and other deeper analyses of text, is therefore infeasible. Over the last few years, we have developed several algorithms and systems that support deeper analysis of natural language [43, 44, 45, 46, 47, 48, 49] that can serve as a basis for the current research.

Our work focuses on techniques for parallelizing natural language processing, and constructing a parallelized Natural Language Processing (NLP) framework for developers to create large-scale NLP-based applications. This involves creating parallelized functionality for optimization based machine learning (very high dimensional vector operations), constrained optimization based inference (including integer linear programming), and a large number of sub-graph isomorphism-like computations. Parallelizing can be done at multiple levels—from data parallel, to decomposing processes to independent threads to algorithmic innovations that would facilitate more efficient learning and constrained optimization algorithms in very high dimensions.

We foresee NLP-based applications that are server-based and those that are client-centric. Client-side applications include smart language translation in mobile devices, intelligent application user interfaces, and human-like interfaces to virtual characters. Client-side applications demand real-time and interactive performance, thus placing stringent performance demands on the underlying implementations, creating an excellent driver application for our parallelization technologies.

3.1.3 Tele-immersive Environments

Future human-human communications can be dramatically enhanced with video environments that enable people to virtually interact together [50, 51, 52]. Our efforts in tele-immersive environments tap into long-standing world-class expertise at Illinois in video and image processing, computer vision, and multimedia systems. We are applying our parallelism technologies to state-of-the-art algorithms for 3D reconstruction, view synthesis, depth calculation, and super-resolution, all of which are computationally intensive and need to hit real-time performance thresholds.

One such example involves parallelization of 3D reconstruction algorithms, which are well known in the computer vision domain. Most 3D algorithms were designed for correctness to derive one 3D image from multiple 2D images and less importance was given to the real-time execution of the 3D algorithms. With the advent of 3D displays and 3D video cameras, 3D reconstruction algorithms must work in real-time; if one wants to achieve high-definition quality of video in spatial (HDTV quality) and temporal dimensions (30-60 frames per second), the 3D video algorithms must employ parallelism. Currently, most of the 3D cameras yield unsatisfactory results. They provide either 15-20 frames per second with very small frames (320-240) pixels or 8-10 frames per second with larger frames (640x480) and are not even close to the performance of HDTV video streaming with frame size of (1920x1080) pixels and 60 frames per second currently available in the 2D video world. Hence, our efforts are focused on parallel algorithms for the 3D reconstruction processes to achieve 2D performance limits but with 3D content.

3.2 Patterns

To make parallel programming easier, we must improve the way we think about parallel programming. Since "clear language breeds clear thinking," we need better ways of describing and teaching parallel programming

activities. Thus, in addition to teaching language features and algorithms, we should also teach patterns used in parallel programming. Expert programmers think about programs at a higher level than beginners. They see relationships between different parts of a program and understand the problems that these relationships solve. Courses and books on parallel programming have tended to focus either on parallel algorithms or on programming languages or lower-level programming models that are used to write parallel programs, but do not teach about software structure and higher-level patterns. Documenting and teaching these patterns should enable more programmers to become experts faster. The patterns can also become a vocabulary for documenting programs and for letting programmers talk about the programs they are designing. Programming languages evolve to better support the most common patterns, and sometimes libraries are designed to automate some patterns. So better understanding the patterns of parallel programming can make parallel programming easier.

The most widely used book on (sequential) software patterns is *Design Patterns* [11], co-authored by one of us, which focuses on object-oriented design. These are relatively low-level patterns that are easy to map down to code. There are a lot of higher-level patterns common to object-oriented programming that are not described in the book, but the patterns in the book have been useful on their own and have been good enough to teach people the style of object-oriented programming.

Work on parallel programming patterns has been going on for some years (papers on parallel patterns are regularly presented at Pattern Language of Programs (PLoP) conferences). Unfortunately, the work has not been as successful as the work of patterns for sequential programming. Some noteworthy efforts are the book by Doug Lea [53] covering low-level parallel programming patterns in the context of the Java threading model, the book by Mattson et. al [54] focusing on high-level patterns that are technology-independent, and Phil Colella's original identification of seven *dwarfs* or categories of applications that cover much of scientific computing [55] (these were later extended to thirteen *motifs* by the UPCRC—Berkeley team [55]). However, much work still needs to be done. In general, the conversion of sequential code to a parallel one may require the development of new parallel algorithms—the programmer may need to discover opportunities for parallelism, make high-level decisions on strategies for the distribution of computation and data, and then decide how to implement the parallelism and synchronization. To be effective, the patterns effort must aid in the navigation of both the high- and low-level patterns that this process encompasses.

As a concrete example, consider one of the high-level seven *dwarfs*, namely *particle codes*. All solutions to this problem share the fact that there is a set of particles in a current state, and the next state is determined by computing and integrating the force on each particle. The solutions differ in how to compute the interaction forces between pairs of particles. The naïve approach is all-pairs but is often the best choice when the number of particles is small. In case of a uniform distribution of particles, one can use a grid to store the nearest neighboring particles for each region of space. If collision is not important, the Particle-Mesh method uses an FFT to solve the problem. For systems with a more clustered distribution of particles, such as astronomical systems, it is faster to group particles and to compute the force produced by a group. Particles can be organized into a hierarchy so that the particles affecting a given particle can be found in logarithmic time, and the Fast Multipole Method can compute an approximated force on a given particle in constant time if the right hierarchy is used. Depending on the particle distribution and application, this hierarchy can be a position tree, a region tree, a kD-tree, or a binary-space-partition tree. These are not only different algorithms; they have completely different patterns of parallelism. The naïve approach uses *task parallelism,* while the others use

divide and conquer, and there are other algorithms that use *geometric decomposition*. Finally, the implementation of these algorithms will differ, according to the target architecture and the selected programming language and run-time (different shared memory models vs. distributed memory vs. SIMD, fixed number of processes vs. run-time load balancing, etc.). Analogous choices arise in the spatial data structures functionality for our dynamic virtual environments applications, as described in Section 3.1.1). A good pattern language must guide a programmer through all these choices.

Although there will be a lot of patterns for parallel programming, most projects will use only a small subset since they will use a particular programming model and require only a few categories of algorithms. A key research challenge is determining how to organize the pattern catalog in a way that is meaningful to programmers and that naturally leads them to the patterns they need.

We believe that the work of documenting patterns is too big for any small group to finish. Success requires a community effort that involves not only academia but also practitioners in industry and national labs and students: pattern languages evolve through a continued interaction between writers documenting patterns, experts reviewing their documentation, programmers using the patterns and students learning patterns. Writers must learn how readers misinterpret and misuse the patterns, and rewrite them to prevent these problems. It is our goal to foster such interaction and create a broad community involved in the development of parallel pattern languages through workshops, courses, and online fora.

We are working to create a body of parallel programming patterns in several ways. First, we are documenting patterns. We are focusing on patterns that are important to us and that are not well documented already. This will include low-level patterns of the technologies we use (e.g. deterministic shared memory and actors) and high-level motifs used by our applications. Second, we are working with other authors to help improve their patterns. Third, we plan to use patterns to teach parallel programming. Experience using our patterns and the patterns of other people will help us to improve the patterns. Fourth, we plan to organize workshops for authors of parallel programming patterns, similar in style to the Pattern Languages of Programming conferences held for years at Illinois. This will bring authors together, reduce duplicated effort, and help concentrate work on the most needed patterns. Fifth, we will organize events for practitioners to both teach them patterns and to get feedback on how they use the patterns. It is easier to try this out on students, but the reaction from practitioners will not be the same, so it is important for pattern authors to hear directly from practitioners.

Documenting parallel programming patterns can have a big impact on parallel programming. Language designers will use them to test the expressiveness of their language. Tool developers will try to automate particular patterns. However, our main interest is in using them to teach the next generation of parallel programmers, whether they are in school or are already professional programmers. Thus, our main goal for the next few years is on discovering the patterns that are most important in parallel programming and in ensuring that they are documented in a way that is easy to understand and to use correctly.

4 Disciplined Programming Models

4.1 Disciplined Shared Memory

Shared memory parallel programming languages have been around for decades. Recent examples include C++ with threads or with the OpenMP parallel constructs [56], Java [57] and C# [58]– these languages support object oriented programming, which is essential for their broad use. The use of a shared memory programming model facilitates the expression of many algorithms, but in its current incarnations, it does not offer adequate protection from program defects. Our goal is to develop disciplined shared memory models and languages that, to the extent possible, minimize concurrency related defects by design, without giving up the conveniences of modern object-oriented languages and the expressivity and performance of low-level programming.

Data-race-free—a fundamental discipline: There is broad consensus that *data races* are a fundamental source of concurrency related bugs in shared memory programs. Data races involve unsynchronized, conflicting accesses to memory, which result in unpredictable, schedule-dependent, and platform-dependent results. There is extensive literature on the difficulties introduced by data races, their classification and formalization, static and dynamic strategies for their detection, and their relation to memory models, including much of our own early work [59, 60, 61, 62, 63, 64, 65, 66]. Many have argued that programs should avoid data races so that the outcome is determined by the order in which synchronization operations execute. In fact, recent efforts on memory models for shared-memory languages have shown that it is extraordinarily difficult even to formalize acceptable semantics for programs with data races. Thus, Java [13] and the upcoming C++ memory model [14] both discourage the use of data races. Unfortunately, neither language prevents the programmer from introducing a data race. C++ leaves the semantics of a program with a data race undefined, an unacceptable strategy for safe languages such as Java. Java provides semantics with data races, but they are too complex for most programmers or system designers.

We believe that a fundamental requirement for safe shared memory programming is that races be avoided or detected [29], resulting in well-defined exceptions. Race avoidance or detection are the only practical ways of enforcing the safety of shared memory parallel code and of providing a feasible shared memory model. Debugging and testing of race-free programs is simpler since there are fewer interleaving sequences to consider—only the (possibly non-deterministic) interleaving of synchronization operations.

Determinism—a stronger discipline: A stronger requirement is that programs be *deterministic* so that, for each input, there is a unique output. While such a program may execute in parallel, the outcome is equivalent to that obtained in a sequential execution, and the relative timing of the executing threads cannot possibly affect the outcome. Programs satisfying this requirement have *sequential semantics*, which yields several major benefits. Such a program can be understood without concern for execution interleavings, data races, or complex memory models. Programmers can *reason* about programs, *debug* them during development, and *diagnose* error reports after deployment using familiar development patterns and tools used for sequential programs. ISVs can *test* codes more easily, without being concerned about the need to cover multiple possible executions for each input. Programmers can use an incremental parallelization and tuning strategy, progressively replacing sequential constructs with parallel constructs, while preserving program behavior. Finally,

two separately developed but deterministic parallel components should be far easier to compose than more general parallel code because, with a deterministic language, a deterministic component should have the same behavior regardless of the external context within which it is executed (with some constraints on the behavior of that external context [18]).

We believe that many transformative programs (where parallelism is used only for performance and is not part of the problem specification) can be expressed in deterministic terms. Nevertheless, non-deterministic behavior may be needed for the efficient implementation of some parallel algorithms. For example, parallel associative reductions are often scheduled in an order that depends on the number of executing threads, resulting in schedule-dependent rounding differences that users may be willing to tolerate. A parallel connected components algorithm will always return the same connected components, but may return different spanning trees for each component. Branch-and-bound search and clustering algorithms are other relevant examples.

Our fundamental thesis is that *client-side parallel programming must be deterministic by default.* Non-deterministic behavior should occur only when explicitly non-deterministic constructs are used. All shared memory parallel code, whether deterministic or non-deterministic, must be *data-race-free*: unsynchronized conflicting accesses to shared memory must result in compile-time errors or run-time exceptions.

Where possible, non-deterministic behavior should be encapsulated behind interfaces with well-defined contracts such that the rest of the program can be guaranteed deterministic as long as those contracts are satisfied. Such operations are often written by expert programmers as libraries or frameworks, and widely reused. Such code may be written *to the meta* to achieve the highest performance and can be difficult to *guarantee* deterministic, but is generally well designed and thoroughly tested. Encapsulating such code helps to localize the places where debugging and testing has to cope with non-deterministic behavior. Thereafter, *trusting* such code is a practical compromise so that application programmers can obtain most of the benefits of determinism as well as the performance of highly tuned library code. Alternately, where possible, we can also use static analyses and formal methods to verify such contracts, as described in Section 5.4.

Providing a parallel performance model: While we have argued for a sequential semantic model (by default) for easier reasoning, we believe that for robust performance, it is important for programmers to be exposed to a *parallel performance model*. For example, consider a language where the only explicitly parallel construct is a parallel loop, and where loop iterates are required to be independent. Although the loop has serial semantics, the user can analyze performance assuming that such a loop does execute in parallel. If the iterates are not independent, then an error will occur at compile time or an exception or warning will be raised at run-time (perhaps reverting to a sequential execution). In any case, the programmer is warned if the parallelism is not achieved. We can think of the explicitly parallel loop construct as an *annotated* loop where, in addition to specifying loop semantics, the programmer provides an indication of the expected execution model and the language implementation tries to deliver that model. Languages such as OpenMP already provide such annotations, as pragmas or directives. However, incorrect directives in OpenMP may cause data races; furthermore, the expressiveness of annotations that are, syntactically, comments, is limited. We believe that we need new programming constructs in the form of *executable annotations*.

Research questions and our approach: To achieve the goals described in this section, we propose to add new programming constructs to existing popular object-oriented languages, such as Java, C# and, possibly, C++. We believe that using such extended languages will provide an easier path to parallelism than using

current languages or totally new languages. If parallelism is pursued using current languages, then it is achieved either by using explicit threading, or by adding directives and writing code in idioms that can be parallelized by the compiler. The former approach is bug-prone; the latter is platform-dependent. A totally new language presents an obvious porting barrier. Our preferred alternative is to add to a popular language *performance annotations*—i.e. syntax extensions that affect the performance model but not the program semantics. With this approach, refactoring tools can support the porting of existing codes as a progressive annotation process and development tools can display a semantically equivalent program in the original language (Java, C# or C++), for debugging. The annotations are not specific to a platform; further, the investment in using them has a long-term payoff because they improve programmability, clarify design decisions, and reduce maintenance costs.

Designing such an extended language that meets the diverse goals described in this section raises many important research questions:

- ***Disciplined Control and Synchronization:*** An important means to achieve the discipline we recommend is the use of well-structured parallel control flow—the equivalent of "goto-less" programming for the parallel world. As Dijkstra argued [3], structured programming results in a simple mapping from the static program to the dynamic execution state so that a program execution is easier to comprehend and debug; at the same time, the banning of goto's causes little loss of performance. Well-behaved parallel constructs such as nested parallel loops or static dataflow programs [67, 68]) (vs. arbitrary fork-join constructs) lead to a simple view of execution state, simplifying reasoning for programmers and compilers. For non-deterministic programs, high level constructs, such as atomic sections, will be preferable to explicit locks. It will be important to provide high-level parallel constructs (such as deterministic and non-deterministic parallel iterators) in order to express prevalent parallel tasking patterns in a structured way. Research is needed to evaluate the expressiveness of such constructs and ensure that no significant performance is lost by imposing such a programming discipline.
- ***Race detection and prevention mechanism:*** We expect that race avoidance or detection can be implemented efficiently through a judicious combination of language features, as described below; of compiler techniques, as described in Section 5.1; and of hardware support for race detection, as described in Section 5.3.1 Whenever possible, it is preferable to avoid data races by design, i.e., having them detected at compile time. Data races can be avoided by providing in the source code sufficient information about memory access patterns so that the compiler can analyze memory accesses and determine that no *conflicting* memory accesses may occur. Our Deterministic Parallel Java (DPJ) language [17] achieves this goal by adding region types and effect annotations to Java. The region types partition the shared variables into disjoint sets, according to their type; the effect annotations specify which sets can be accessed or updated by a particular statement. The program is legal only if two concurrent threads never have conflicting effects on the same set. The annotations enable the programmer to express the memory sharing patterns in the code, and the reasons why the code is believed to be data-race-free. The compiler verifies the annotations (no conflicting concurrent accesses). Flaws in the programmer reasoning are exposed as compile-time errors, thus facilitating debugging; and the compiler and run-time can optimize the code using the information on data sharing.

  Research is needed to determine how expressive such a type and effect system can be, and how it is best supported. At a minimum, we expect that some applications will require dynamic (executable) annotation

mechanisms, relegating some checks to run-time, but still allowing for more efficient checks and more informative run-time exceptions. We would like a language design that encourages and supports *early binding* of type and effect information, but does not prevent *late binding*; and that infers as much as possible information on access patterns, thus reducing the need for explicit annotations.

- ***Definition of conflicting effects:*** The DPJ effect system currently supports two effects: *read* and *write*, with writes conflicting with any other effect. This effect system can be extended to support more types of operations on shared variables, with a suitable table of conflict rules. For example, associative reductions or commutative operations on concurrent data structures can be expressed via a *commutative* effect on a method; this would conflict with any other effect and would indicate that the method can be invoked in any order within a parallel phase and the final result will be unique (up to rounding errors, for floating-point reductions). As another example, the Multi-phased Shared Arrays (MSA) language [69, 70] supports three mutually exclusive effects of read, write and accumulate, and allows the access mode of an array to be changed during execution.

- ***Encapsulation:*** Encapsulation of non-deterministic code, as described above, could be handled using suitably defined and validated interface contracts. For example, users of a generic reduction or parallel prefix library operation should be able to define pure, associative operators, as in languages like Fortress [71]: the *pure and associative* requirement is a contract that can often be checked by the compiler, possibly relying on effect annotations. A more complex case occurs when the result is deterministic in some essential sense, but may vary in representation or other characteristics. For example, a parallel connected components algorithm will always return the same connected components, but may be non-deterministic in terms of the representations of these components. A floating point parallel prefix computation may suffer from different rounding errors, according to the order operations are applied. A final example is the mesh refinement discussed in [72] where the outcome is guaranteed to satisfy the Delaunay property although its exact form may differ across executions. Proper interfaces to such libraries raise non-trivial theoretical and pragmatic problems.

- ***Virtualization and Locality:*** Traditional thread-based shared memory programming models such as *pthreads* or Java threads use relatively heavy-weight threads and therefore generally work efficiently when the number of threads match the number of physical cores. This results in programs that are either unable to adapt to changes in the underlying processor resources or requires complex code to perform non-trivial load balancing techniques. Higher-level programming models, such as OpenMP [56], TBB [73], and Charm++ [6] virtualize CPU resources: the degree of program parallelism (concurrent tasks) need not match the number of physical cores. Such virtualization greatly improves load-balancing but may also hurt locality, and therefore performance, if the underlying run-time system, which schedules program tasks onto the physical cores, does not have the knowledge of communication patterns. The Partitioned Global Address Space (PGAS) languages such as Titanium [74], UPC [75] or CAF [76], provide some mechanisms for handling locality, but match a model of a distributed memory system with a fixed number of nodes. Neither approach is satisfactory. We plan to explore mechanisms to provide better control of locality while virtualizing physical resources, building on our experience with object-based decomposition in Charm++ [77].

4.2 Parallel Operators

A particularly useful form of disciplined control is provided by data parallel operators: in data-based parallelism, tasks perform the same operation on different components of a pre-existing data ensemble or a simple iteration domain. In the simple case, such as a vector sum, the operations on distinct elements are independent and perform the same amount of work. In more complex cases, the operations may take different amounts of time and may not be all independent: one needs to find and repeatedly schedule an independent subset of operations [59, 64, 69].

Data parallelism can be encapsulated inside operands defined on aggregates, such as arrays or sets and, hence, can be incorporated into traditional languages in a transparent manner. Second, because parallelism is encapsulated in the data parallel operators, it is possible to modify the implementation of these operators. Portability is achieved by providing optimized versions of the data parallel operators for different platforms including shared and distributed memory multiprocessors, SIMD processors, and combinations of these. The result is that the same code can be executed on different platforms with minimal loss of performance. Finally, data parallel operators are an effective mechanism to encapsulate non-determinism. The possible results of these non-deterministic operations could be identical or just be guaranteed to satisfy some property while differing in significant ways as discussed above.

We have extended the traditional array operators of array languages such as APL, Fortran 90, or Matlab by adding tile abstraction and well-defined parallel semantics. We thus obtained a new data type that we call Hierarchically Tiled Arrays (HTAs) which enables the direct manipulation of tiles sequentially or in parallel [19, 20]. This extension facilitates the development of array-based parallel codes that have a high degree of locality and gives programmers the ability to directly control data layout, scheduling strategy, affinity, data distribution, and communication. We are currently developing abstractions that extend data parallel operations to other classes of aggregates such as sets, graphs, or trees to enable the development of highly readable parallel non-numeric programs.

4.3 Metaprogramming and Autotuning

When the target machine is a multicore, productivity suffers not only due to the increased likelihood of defects, but also because of the need to make these programs efficient and scalable. Targeting multicores complicates optimization since programmers must deal with issues that do not arise in the sequential world such as load balancing and communication [78]. The natural way to address this problem is automation. Optimization tools have always been important, but their importance is now even greater since they are our only hope to compensate for the increased difficulty brought on by parallelism. In the spirit of separation of concerns and following tradition, we are developing tools whose only objective is performance optimization. The most important such tool is of course the *program optimization passes of the compiler*, but our experience indicates that compilers at least with today's technology are not sufficient to address the productivity problem and even with the support of the best compilers the development of efficient and scalable programs remains laborious. The *library generators* implemented using autotuning techniques constitute a promising new class of tools. These produce codes that achieve impressive efficiency across a wide range of machines. Some of these generators, including ATLAS, FFTW, and SPIRAL [79, 80, 22]; have gone beyond the experimental

stage and are now routinely used by library developers or as a component of commercial systems [81]. Most library generators use empirical search to find a near-optimal version. The main idea is to generate several versions of the routine being automatically implemented by replacing algorithms and changing the way they are implemented. Variations in the implementation of an algorithm include loop unrolling, vectorization, reordering statements to improve scheduling, and tiling the computation to enhance locality. The generator executes these versions on the target machine in order to evaluate them and select the one with the best performance. Although in some cases it is feasible to conduct an exhaustive search, often the space of possibilities is too large and a subset must be selected for generation and evaluation. Typically, performance is execution time, but power consumption could also be taken into account. Library generators can be conceived as *metaprograms* which embody in a single code all the versions that are to be empirically evaluated. Although the vast majority of widely known autotuning metaprograms are library routine generators, metaprograms implementing complete applications are also of great importance when the bulk of the computation cannot be implemented in terms of existing libraries.

The autotuning approach has the advantage over compilers that it can make use of semantic information that typically would not be available to compilers, but this information must in some cases be provided by the programmer and this means extra work. However, the effectiveness with which these autotuning systems enable portability across machines and machine generations has made the extra effort worthwhile in the past because although the initial effort is higher than that required to develop a highly tuned version for a single machine, porting to new machines becomes much simpler. We expect the impact of these systems on productivity to be even greater when dealing with parallelism. Furthermore, we believe that many applications can benefit from autotuning without the intervention of programmers if the application is written in terms of routines for which there exist a generator or in terms of other primitives such as codelets or data parallel operations (see below).

There are several important issues that must be addressed. In fact, today's developers of autotuning software must do all the work from scratch with practically no software support and therefore we must study techniques to increase the applicability of autotuning. Therefore, building on our earlier work on library generators, we are working to make metaprogramming for autotuning a more useful and effective methodology so that it can become one of the foundations of productivity for multicores. In particular, we are studying and developing abstractions and tools to facilitate the implementation of parallel self-tuning codes, such as the ones outlined next:

- Continue advancing our understanding of data-dependent autotuning. In many cases, the best version of a computation depends on the input data. For example, many sorting algorithms perform differently for different input data sets and the choice of the best algorithm is therefore data dependent. We need runtime selection to enable data-dependent optimization. One possibility is to generate code that contains multiple semantically equivalent versions of the computation, one of which is selected at execution time. Alternatively, autotuning could be done at execution time. The impact of runtime selection on performance is demonstrated by our studies on the autotuning of sorting [23, 24].
- Develop languages for metaprogramming and autotuning. This metalanguage should describe collections of valid implementations, ways to combine code components, and possible values of implementation parameters. This metalanguage should also make it possible to specify strategies for the identification of optimal points in the space of implementations for each target machine. The metaprograms could implement

libraries or complete applications. One of our goals is to study the complexity of these metaprograms. We believe that, with the appropriate design, metaprogramming will dramatically facilitate the initial tuning and subsequent porting to new generations of machines. This work will build on our experience with a prototype metalanguage called X-language [82].

- Implement autotuning versions of the parallel operators discussed in the previous section. Autotuning versions of these operators will facilitate portability across classes of parallel machines, including muticores, multicomputers, and SIMD processors, and will enable powerful optimizations.
- Design and development of a codelet-based optimization strategy. Codelets are computational blocks that occur sufficiently often in programs that it is worth developing manual and/or automatic techniques to (i) isolate them within the context of large programs, and (ii) optimize their performance, possibly using customized approaches. We plan to search for codelets by analyzing source programs statically and binaries dynamically. Our strategy is to optimize programs by recognizing and replacing codelets with improved versions. As mentioned above, with autotuning versions of codelets and data parallel operators, it will be possible to bring the benefit of autotuning to numerous programs without the need for programmer intervention.
- Develop search strategies and implement them to support autotuning. Given the astronomical number of versions that are typically possible, effective search strategies that are generally applicable are of great importance. We have recently investigated effective strategies for pruning the search space for massively parallel GPUs [83]. Analytical models to prune the search or avoid it altogether, statistical techniques, empirical models, as well as advanced search techniques must be studied.

4.4 Domain-Specific Environments

One way to make parallel programming be "just programming" is to hide the parallelism inside domain-specific languages or libraries. We call a collection of domain-specific tools *domain-specific environments* (DSEs). Many existing DSEs are being ported to take advantage of multicores, including languages like Matlab [84] or Labview, libraries like Apple's Core Audio [85], and frameworks like PureMVC [86], a user interface framework for ActionScript. There are also DSEs designed from the start to hide parallelism, such as Google's MapReduce and the parallelized application frameworks we are developing, mentioned in Section 3.1.

Ideally, the goal of many such DSEs is for application programmers (especially those who are already using these environments) to not have to learn parallel programming—using these environments should allow their programs to automatically take advantage of more cores. In practice, however, much can go wrong, and application programmers are often disappointed by the performance of their programs using a DSE that hides parallelism. Unfortunately, the development tools they can use to diagnose and fix these problems usually do not have any knowledge of their domain, which means that the benefit of domain-specific programming is lost; worse still, it can actually be harder to tune a program if performance behavior can be understood only at the level of a base language and low-level run-time instead of the programmer's source code.

There are too many possible domains for us to develop new tools or specialize existing tools for each domain. Instead, we must develop *generic* techniques and tools that enable domain experts to build effective, rich parallel DSEs. We plan to implement these for the parallelized frameworks being implemented for our applications,

as well as help design these frameworks. We expect to discover common themes and recurring problems that will form the basis for such generic techniques as described next.

A true "programming environment" for a domain should include domain-specific development tools such as tools for code optimization (i.e., domain-aware compilers), performance analysis, debugging, refactoring, and interactive development. All of these tools would present information to the programmer in the concepts and terminology of the domain, rather than simply in those of a base language or run-time system.

One of the tools that often needs more information about a DSE is the compiler. Compilers miss opportunities to optimize applications that use domain-specific libraries and frameworks (and often even domain-specific languages) because they do not understand the domain. A compiler can sometimes produce a more efficient program if it knows that two vectors drawn from the same matrix do not overlap, or that two operations on a stream commute. There need to be ways that library designers can describe properties of their library to compilers so that the compilers can better compile applications that use the library. The challenge, of course, is to make the underlying compiler domain-independent, *including* the description language used for specifying such properties. We believe this is feasible because *the properties useful for performance optimization are often domain-independent*, e.g., properties like data reuse, algebraic equivalence of operations, commutativity, variable privatization, etc., even though the semantics that lead to those properties are not.

Ideally, the interface of a DSE should not change when it starts to support parallelism. However, there are limits to the performance gains that can be achieved with complete backward compatibility, so often the DSE will introduce new features. For example, the first version of Matlab that supported multiprocessors had the same interface as the previous version, but recently Matlab has introduced constructs for explicitly introducing parallelism into a program. The Java FJTask library has added multiple mechanisms to perform the same operation; programmers porting their code to use FJTask often miss opportunities to use the most efficient mechanism [87]. Sometimes libraries add non-blocking operations to help speedup applications. We shall study tools and methodologies to help application programmers cope with such interface changes.

4.5 Actors

So far, we have largely focused on shared memory programming. For many reactive programs (where concurrency is part of the problem specification), non-determinism is inherent to the problem and alternatives to pure shared memory are attractive. In this context, we explore actor based models.

We draw again on object-oriented programming concepts, where objects encapsulate data and behavior, and separate interface (what) from the representation (how), enabling modular reasoning and evolution. In contrast to introducing concurrency through threads where control remains external to the objects, it is natural to extend the concept of objects by modeling each object as an autonomous agent or *actor*, operating potentially in parallel with others.

The Actor model of programming [25] enables programs to be decomposed as self-contained, autonomous, interactive, asynchronously operating components that, by default, communicate using asynchronous message-passing. The asynchrony in the model allows us to model non-determinism inherent in reactive systems, cloud computing, sensor networks and modules requiring history-sensitive behavior. Asynchronous operation allows flexibility in placement and scheduling actors on different cores, as well as facilitating

mobility [26]. This allows the runtime to preserve locality, balance loads, and manipulate schedules in order to enforce synchronization constraints [27], meet deadlines, minimize energy consumption, and preserve quality of service [28]. Not surprisingly, because of the need to program peer-to-peer systems, web services and applications [88], and now multi-core processor architectures, there has been a growth of interest in languages based on the actor model (e.g., E, Erlang, Salsa, Scala [89, 90, 91, 92]) as well as actor libraries and frameworks. Figure 3 illustrates the components of an acor:

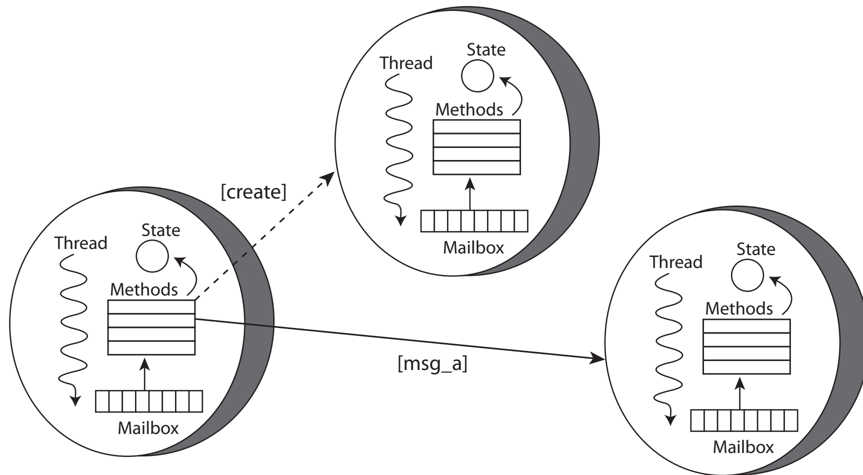

FIGURE 3 • An Actor represents a concurrent object: like objects, it encapsulates state and methods, but it also has its own thread of control. Each actor's thread steps through the actor's behavior, and when the actor is idle, picks a new message from its pending queue of messages for processing. Note that this figure represents a semantics view of an actor. In implementations, actors may share a mail queue, or message sending may be transformed into method calls. Actors may send messages (asynchronously invoke methods) to other actors, or create new actors.

Formally, actors can be defined as objects augmented with a thread of control and a mailbox. The lifecycle of an actor involves creation, processing messages one after the other and destruction. During the processing of a message, an actor may create other actors, send messages and change its internal state. However, there are some notable research challenges. An important aspect of actor programs is the specification of synchronization constraints (when) among the concurrently executing actors. Modular specification and efficient enforcement of constraints is an interesting direction. Hardware variability (due to power management and production defects) makes the difficult problem of concurrent programming even harder. Run-time architecture that supports efficient migration of actors can enable dynamic load-balancing algorithms.

Low-level concurrency models require the programmers to specify resource allocation at task level. But since actor programs may comprise of a very large number of light-weight tasks (actors), such allocations need to be specified at a higher level of abstraction. However, the heterogeneity of resources and possibility of task-specific accelerators may encourage programmers to specify nature of resource suited to a particular

task. Moreover, programmers can specify application-specific QoS requirements at a high-level e.g. a video application can set bounds on the delay between its audio and video feed. Satisfying such QoS constraints in addition to the above-mentioned constraints at run-time requires introspection capabilities to query the resources as well as clever scheduling algorithms.

Our focus here will be again driven by the separation of concerns philosophy—how to separate the specification of individual actors and the specification of aggregate multi-actor constraints such as synchronization, resource allocation, quality of service, reliability, energy consumption, etc., and how to efficiently implement these constraints in the context of an actor-oriented run-time.

5 Development and Execution Environments

5.1 Translation Environment

The Illinois UPCRC compiler infrastructure sits between the disciplined parallel programming languages and frameworks described in Section 4 and the runtime and hardware described in Sections 5.2 and 5.3. The requirements and systems views presented by these two interfaces differ, and it is the responsibility of the compiler to translate between the two interfaces. The challenge, and opportunity, is to make use of the new information available from these two interfaces to find more parallelism, improve cache locality, reduce runtime overheads, and assist programmers in porting existing sequential or parallel codes to the new programming models. The runtime system and front-end type system can provide information about code reachability and aliasing relationships that static compiler analyses cannot otherwise derive.

We are basing the new dynamic compilation infrastructure on the existing open-source LLVM compiler infrastructure [93]. We are designing a new program dependence graph (PDG) based internal representation to support static and dynamic analysis and transformations. On this internal representation, we are implementing a variety of traditional PDG-based loop transformations (strip-mining, interchange, unswitching, scalar expansion, distribution, reassociation, fusion), together with a range of interprocedural program analyses (pointer analysis, escape analysis, dependence analysis). These techniques are useful for a variety of purposes: extracting parallelism from loops that are not already fully parallel, extracting sharing information between tasks to optimize parallel performance, and restructuring loops for better memory locality.

New sources of information

We believe that it will be necessary for the transformation engine to leverage information from a wider set of sources than traditional pointer and array alias analyses.

- The disciplined shared-memory languages described in Section 4.1 include annotations that programmers use to indicate which loops and procedures they want run in parallel. This allows the compiler to focus its efforts only on sections of code that really matter to the programmer, while avoiding added overheads on code that the programmer knows not to be (efficiently) parallelizable.
- Safe deterministic parallel languages may include type systems that allow the programmer to create code with provably unaliased pointers. The additional aliasing information allows the type system to guarantee race-freedom. A key research question is how to transmit the information from the type system to the transformation engine. Our first prototype will do this by having the front-end system generate assert calls that the transformation engine can then use to generate a sparser program dependence graph.
- Data-types and frameworks often have invariants that a compiler can leverage. For example, the DPJ language mentioned in Section 4.1 and the Hierarchically Tiled Arrays described in Section 4.2 have mechanisms to create sub-arrays that are guaranteed to be disjoint. The compiler can use this property to prove that work on different sub-arrays is independent.
- A conservative static compiler analysis must assume that all paths through the program can be taken. At runtime, however, we have specific information about the values of variables that are invariant (in, for

example, a loop that we would like to parallelize). At runtime we can use these values to prove that some paths through the program cannot possibly be exercised in the currently running instance. Those paths can be eliminated, potentially reducing inter-iteration dependences, and thus providing more opportunities for optimization.

Runtime Efficiency

In addition to transforming programs to expose more parallelism and to improve locality, the compiler will be responsible for passing information to the runtime software (Section 5.2) and hardware (Section 5.3) that can be used for more efficient decisions. For example, the compiler can pass information about variable liveness to the Bulk Multicore architecture (Section 5.3.1) so that it can eliminate checkpointing overheads on data structures that are not live. Similarly, the compiler can transmit information on disciplined sharing characteristics (for example through the use of DPJ's type system or through traditional dependence analysis) to the runtime and the hardware for various purposes—information on code regions that are already proved data-race-free will allow the Bulk Multicore architecture to focus on other regions for its runtime data race detection; the DeNovo architecture (Section 5.3.2) will use such information to provide a more efficient software-driven coherence fabric; and the runtime will use the information to improve scheduling decisions based on locality.

5.2 Runtime System

A major challenge in developing software for client platforms is hardware diversity. It is untenable to ask software vendors to adapt or optimize their programs for each of these platforms. Instead, we believe it is important to provide an execution environment that attempts to meet the applications goals the best it can given the available resources on the platform. Two key concepts in this statement are worth emphasizing:

1) ***Application goals:*** We believe that quality of service (QoS) will be increasingly important on client systems to provide a good user experience. Many performance-hungry applications can be written so as to provide the best answer that can be computed by a given deadline, and will be written this way to be responsive without jitter and long pauses. We expect applications to be annotated and organized such that a level of output quality can be selected based on the available resources.

2) ***Available resources:*** We expect heterogeneity in client platforms. Not only will there be heterogeneity between platforms (different design/price points within a process generation and across process generations), but also within a platform. We expect future platforms to include a variety of cores (a few large cores, optimized for latency, for sequential performance and many small cores, optimized for throughput, for parallel workloads); even when designed to be similar, process variation will endow them with different performance characteristics. Furthermore, the resources that can be applied to each program's execution may vary over time as applications are launched or complete and due to adaptation of the hardware to physical constraints (e.g., power, temperature, battery life, and aging).

The process of trying to maximize utility (the sum of the user benefits of all running programs) given the available resources is an optimization problem. Drawing on our previous work [77, 94, 95], we use a combination of task over-decomposition and a hierarchical adaptive resource allocation strategy for an efficient solution to this problem. Figure 4 illustrates our approach.

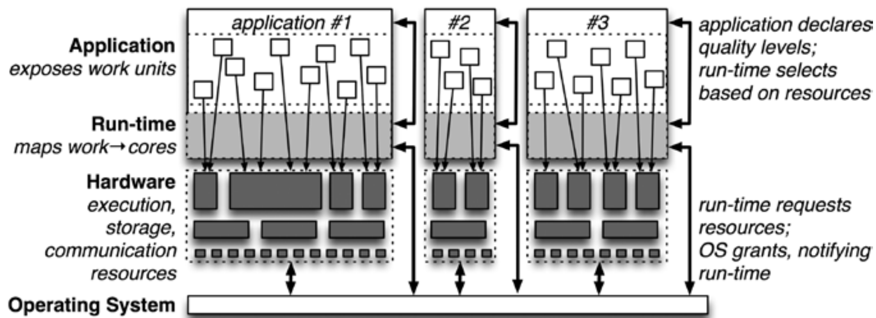

FIGURE 4 • The anatomy of a managed execution of a heterogeneous multi-core machine running three parallel applications.

Each application runs inside a run-time system. The run-time system has three key goals. First, it virtualizes the hardware platform for the application. That is, application writers are charged with exposing the concurrency (in the form of work units), but the binding of work units to cores is handled transparently by the run-time system. Second, the run-time system communicates to the operating system a set of utility/resource-requirement tuples for different possible configurations of the application. Third, once the OS makes a resource allocation (as described below), the runtime ensures that the application uses these resources in the most efficient way, appropriately reacting to any local changes in the application behavior and system environment.

The job of the operating system is to make global resource allocation decisions, arbitrating between all of the applications. These decisions attempt to allocate resources so as to optimize the global utility. Resource allocations are performed as *block grants* of resources to applications; as much as possible, the operating system attempts to space multiplex parallel applications by assigning uninterrupted use of a subset of the processors. Such block grants only need to be recalculated when the set of active applications changes or an application substantial changes its utility/resource tuples. When the operating system changes an application's allocation, it notifies the application's run-time.

Our proposed run-time system derives a number of important ideas from our previous work on the Charm++ run-time [77]. By having the programmer over-decompose into significantly more work units than the expected number of hardware threads, the Charm++ run-time can both achieve good performance across a large range of core counts and tolerate heterogeneity in the core capabilities. Key to achieving these are its adaptive load balancing which optimizes communication by co-locating communicating work units.

The hierarchical structure of this system (global optimization by the O/S, optimization within an allocation by the run-time) and optimization at multiple time scales was motivated by our previous work in the GRACE project [94, 95]. That work demonstrated these techniques to build a cross-layer adaptive system which could adapt hardware, network, O/S, and application algorithms to minimize power consumption while still meeting QoS requirements.

5.3 Hardware Architecture

The arrival of multiprocessors on a chip fundamentally changes many of the parameters for hardware architecture research. We make the following key observations that impact our work:

- ***Hardware must be used for programmability.*** Until a few years ago, the only goal of much research in hardware architecture was to provide the best possible cost/performance ratio. Most recently, power-efficient (or energy-efficient) performance has been the dominant goal. The advent of multicore systems forces us to fundamentally rethink our priorities. Since performance is now achieved by parallel software, it becomes important for the hardware to facilitate the development of such software. As we argue in Section 5.3.1, silicon can be increasingly leveraged to support ease of programming.
- ***Hardware must scale in performance, not complexity.*** As the primary means for performance will be from an increased number of cores (doubling every 18 months), hardware must be designed to scale in performance but not in complexity. The *memory wall* will be a major impediment to performance scalability [96] and we must focus attention to memory hierarchy design, communication protocols, and synchronization protocols. Complexity must be reigned in—this is easiest to achieve with simple and modular designs. This includes design for the cores, cache hierarchy, coherence protocols, and network fabric, and provision of standard interfaces to manage core-to-memory and core-to-core communication that can accommodate heterogeneous cores.
- ***Multicore requires new cache protocols.*** Coherent shared-memory systems either use snooping protocols that scale to a few tens of cores, or use directory protocols that significantly increase memory latency. In addition, directory protocols have been designed for NUMA machines, where a node integrates processor and memory, and access to a remote cache requires multiple chip crossings; multicore chips will have hundreds of cores, all accessible with no chip crossings, and all nearly equally afar from the memory DIMMs. This requires new designs, such as embedded-ring protocols [97]. Furthermore, the synchronization mechanisms and the data sharing patterns exhibited by high-level languages and parallel algorithms may benefit from novel protocol designs.
- ***An opportunity for hardware-software co-design***. We argue in Section 5.3.2 that a large part of the complexity of current hardware concurrency mechanisms arises from a software-oblivious approach to hardware design. We now have a unique opportunity to rethink the entire system stack and develop hardware that is better aligned with the needs of modern software. For example, coherence protocols can be greatly simplified by focusing on the support for disciplined programs. Over time, we need a fundamental rethinking of concurrent hardware, including how to express and manage concurrent work units, communication, synchronization, and the memory consistency model, in tandem with our rethinking of the best practices for concurrent software.

We describe below two architecture projects we are pursuing in UPCRC. The *Bulk Multicore* project focuses on supporting a flexible substrate with scalable cache coherence, high-performance sequential memory consistency, and an easy-to-use development and debugging environment. The *DeNovo* project rethinks concurrent hardware as a co-designed component of our disciplined programming strategy, both for enforcing the discipline and for exploiting it for simpler and more efficient hardware.

5.3.1 The Bulk Multicore: High-Performance, Programmable Shared Memory

The goal of the Bulk Multicore architecture is to enable a scalable shared-memory substrate that provides a *highly programmable* environment, while delivering *high performance* and keeping the *hardware simple*. The Bulk Multicore advances usability and programmability by efficiently supporting both the disciplined software described earlier in this document as well as currently-existing software stacks—including those for which performance is paramount. It provides novel hooks and support for a sophisticated development and debugging environment. Such environment will have an overhead low enough to be *on* all the time—including when the user runs production codes. In addition, by providing hardware cache coherence, it removes the burden of managing data sharing from the programmer or run-time system software. Finally, by supporting high-performance sequential consistency, it provides a more usable platform for the software.

The Bulk Multicore builds on the recently-proposed BulkSC architecture fabric [98]. It is also inspired on work on shared-memory multiprocessor architectures at the University of Illinois. In particular, it builds on mechanisms for state buffering and undo, as in thread-level speculation designs [30, 31, 99, 100], and on scalable coherence protocol designs [101, 97].

Providing Scalable Shared Memory for General-Purpose Computing: The idea behind the Bulk Multicore is to eliminate the need to commit one instruction at a time, which is an important source of design complexity in a multiprocessor environment. In the Bulk Multicore, the default execution mode of a processor is to commit only *chunks* of instructions at a time. A chunk is a *dynamically defined* group of consecutively-executing instructions, for example 2,000 consecutive instructions. Such chunked mode of execution and commit is invisible to the software running on the processor.

Each chunk executes on a processor *atomically* and *in isolation*. Atomic execution means that none of the actions of the chunk are made visible to the rest of the processors until when the chunk completes and commits. Execution in isolation means that if the chunk reads a location and, before it commits, a second chunk in another processor modifies the location and then commits, then the local chunk gets squashed and has to restart. Finally, the commit of chunks is globally serialized in hardware.

To execute chunks atomically and in isolation inexpensively, the architecture relies on *hardware address signatures* [102]. A signature register is approximately 1-Kbit long and contains an accumulation of hash-encoded addresses through a Bloom filter. The hardware automatically accumulates the addresses read and written by a chunk into a Read (R) and Write (W) signature, respectively.

Since chunks execute atomically and in isolation, commit in program order in each processor, and there is a global commit order, the architecture supports Sequential Consistency (SC) at the chunk level. As a consequence, it also supports it at the instruction level. Importantly, it supports SC with a low hardware implementation complexity while delivering high performance. The hardware implementation complexity is low because memory consistency enforcement is largely decoupled from processor structures. There is no need to support global snoops on critical processor structures like the load queue; detection of consistency violations is performed with simple signature operations outside the processor core. At the same time, performance is high because the processor is allowed any reordering and overlapping of memory accesses within a chunk—even *across* synchronization operations.

The proposed architecture supports cache-coherent shared memory in a scalable way. Cache coherence is maintained with signatures, without the need to send individual cache-line invalidation messages. Cache coherence combines high performance with ease-of-programming.

Providing a Highly Programmable Environment: A key goal of the project is to develop novel architectural designs to provide a highly programmable environment for general-purpose shared memory. We are building on work at the University of Illinois on such issues [34, 35, 103, 104]. The Bulk Multicore provides a highly programmable environment in two main ways.

The first one stems from its support for SC, *even* for programs with data races. Providing SC is beneficial for a variety of reasons. Chief among them is the fact that existing software correctness tools almost always assume SC. Using them in combination with hardware that provides SC will make them most effective. In addition, debugging concurrent programs is easier in a machine that provides SC, since the possible outcomes of memory accesses are more intuitive. Indeed, subtle data races can be very hard to debug under relaxed memory models. Finally, supporting SC also simplifies support for safe parallel programming languages, such as Java.

The second way in which chunked execution provides a more programmable environment is by enabling very low-overhead debugging techniques—opening the door to a sophisticated, always-on debugging framework for production runs. The key insight is that development and debugging tools do not need to record or be concerned with individual loads and stores—only with chunks. This can reduce the amount of bookkeeping or state required substantially, for example, for deterministic replay of parallel programs [32, 33] and for data race detection [29].

Leveraging the Bulk Multicore: Our goal is to integrate this flexible architectural fabric with the other layers in UPCRC, and develop fundamentally new capabilities to support such layers. In particular, to support the disciplined shared-memory models described in Section 4.1, we will develop low-overhead hardware primitives for data-race detection. Such primitives will also be usable by the compiler described in Section 5.1, to detect data races in code regions that the compiler has not proved data-race free. Moreover, the compiler will also be able to drive low-overhead hardware check-pointing and undo primitives to generate higher-performing code. Finally, we will develop better hardware primitives for deterministic replay, software testing, and generation of alternate interleavings to complement and to interface to the software correctness tools of Section 5.4.

5.3.2 DeNovo: Rethinking Hardware for Disciplined Parallelism

A large part of the complexity and inefficiency in current hardware concurrency mechanisms arguably arises from a software-oblivious approach to hardware design. This paper has argued that software dependability is paramount, and drives the use of disciplined models. The DeNovo project asks the question that if dependability drives disciplined software, then *how can we design hardware from the ground up to exploit this discipline (e.g., for higher performance) and how can hardware best support and foster this discipline (for higher dependability).* We are particularly inspired by two previous lines of work.

First, the history of the work on memory consistency models exemplifies well the pitfalls of software-oblivious hardware design. Many hardware models were proposed that gave better performance than sequential consistency, but were complex to program, difficult to understand, and promoted questionable programming practices. The software-centric data-race-free model observed that for programs that do not contain data races, the hardware and compiler can easily provide sequential consistency with high performance [65, 66].

The model therefore guarantees sequential consistency only to data-race-free programs and is the basis for most commercial consistency models today [13, 14]. *The focus on a discipline (data-race-free) both simplified system specification and design and allowed higher performance.*

Second, we are also inspired by recent projects seeking safe operating systems. The Singularity project [105] rethinks the operating system from the ground up with dependability as the primary criterion. It took into account the large advances in languages, static analysis, and verification tools. It showed that rethinking requirements can lead to solutions that are much better than current designs, and it is indeed possible to design in safety even for software as complex as an OS. Another approach for OS safety is typified by the Secure Virtual Architecture (SVA) project which uses a typed virtual instruction set to provide safety for commodity OS code [106].

DeNovo seeks a fundamental rethinking of concurrent hardware, given the assumption that most future software will be disciplined for better dependability. The following discusses directions we are pursuing, including a disciplined hardware concurrency (or consistency) model, techniques to exploit the discipline for better simplicity and efficiency, techniques to enforce discipline for better system dependability, and mechanisms for implementing such a hardware/software interface.

Disciplined hardware concurrency model: Like the data-race-free model, we take the approach that hardware should provide easy to reason semantics (sequential consistency or stronger) only for disciplined programs. For arguments made in Section 4.1, we go a step further—programs that violate the discipline should either not be compiled or should raise runtime exceptions. This approach clearly requires a careful definition of "discipline" at the hardware level, and lies at the heart of our hardware/software co-design approach. Based on Section 4.1, we expect to optimize for software that is deterministic by default, with non-determinism that is requested explicitly and well-encapsulated. Unlike a language-level model, however, hardware must balance the requirements of all expected applications, the operating system, and legacy code. To what extent the semantics we develop for our disciplined shared-memory language can be translated into a hardware concurrency model is a research question we are exploring. We are also exploring how hardware should support messaging, as motivated in Section 4.5 as a language-level mechanism, and how to integrate it with shared-memory.

Rewarding discipline: We believe that a disciplined model can be exploited to simplify and optimize many aspects of the system. For example, current systems communicate implicitly through a hardware cache coherence protocol. While hardware coherence makes communication transparent to software, it is also one of the most complex and difficult to scale aspects of concurrent hardware design. We ask if the complexity of hardware coherence is warranted for disciplined models. Or is it possible to exploit data sharing and synchronization information often available naturally with disciplined models to drive software-controlled communication? Shared-memory programs may provide this information through type (region) and effect annotations; actor programs provide this information explicitly. We are investigating how far we can exploit such information to both simplify and optimize the hardware communication fabric.

Disciplined models also impact task management and scheduling. A key feature of disciplined models is task over-decomposition (to virtualize the number of cores), with scheduling mediated through a runtime system. Current runtime systems are mostly hardware oblivious (e.g., Cilk, [107] TBB [108]), missing significant opportunities. Many open questions need to be investigated; e.g., how best to represent a task

and a continuation at the hardware level, how to virtualize heterogeneous hardware for the runtime, how to perform locality- and synchronization-aware scheduling in the context of the hierarchical scheduling algorithm described in Section 5.2, and hardware support for common scheduling tasks (e.g., [109] proposes hardware queues). Again, information on data sharing and synchronization available from disciplined software can drive more optimal design choices.

Enforcing discipline: There will invariably be untrusted and unverified code that potentially does not obey the required discipline. Hardware can provide support for sandboxing such code and not letting it affect other portions of the application by allocating an explicit protection domain with a disjoint shared memory partition. An observed violation of default contracts (e.g., data-race-free) would flag an illegal application and result in an exception. An analogous situation arises for trusted but unverified code. Such code will come with a contract or expectation, and hardware can use additional runtime information to verify that the contract is not violated. An example contract is to ensure there are no conflicting accesses between such code and the disciplined sections. This can be done using variants of data race detection techniques in hardware; however, we are considering rather stylized situations which can potentially be handled more effectively than the general case.

Interface mechanism—a typed virtual instruction set: Virtualization is perhaps the only viable means for supporting the expected variety of heterogeneous architectures as well as implementation-specific mechanisms (e.g., communication management instructions) that vendors may be reluctant to make part of their software exposed ISA. A virtual instruction set computer (VISC) [36] provides a separate low-level but rich and machine independent ISA for software, which is then translated to a hardware ISA. The latter is implementation-specific and never exposed to the software (other than the dynamic translator). There are several advantages to such an approach as described in [36, 37].

DeNovo will use a virtual ISA. The design of such an ISA and the integrated runtime is part of our research. Given our emphasis on safety and dependability, it is natural to consider a *typed* instruction set as in [36]. Having a typed virtual ISA allows expressing rich high level information in a structured way to the hardware and also makes it easier to check safety properties at install or runtime. For example, type information can indicate which method calls are tasks, or even more sophisticated structures like a parallel array of tasks, for better coordination of scheduling. The type information about tasks can also be used to prove at runtime that the code (including any dynamically loaded components) obeys the disciplined paradigm which can then be exploited in many ways. Types also provide a natural mechanism to provide information on the regions and effects discussed in Section 4.1 to the hardware, which can be exploited as described above.

5.4 Formal Methods and Tools to Check Correctness

The disciplined programming methodologies we propose will go a long way in managing the complexity of writing a concurrent program and arguing its correctness. However, multicore programs will still be prone to errors due to concurrency, both in the new components written for our disciplined models and in components that continue to use current models.

There are two serious problems related to correctness that we wish to address through a combination of formal methods and correctness checking tools. First, there is a pressing problem in the engineering of tools that assure correctness of concurrent code. Testing, which is the most useful mechanism employed in

the industry to assure some degree of correctness, is fundamentally challenged in the multicore domain. A test for a sequential program generally consists of a test harness that interacts with the program under test, feeds it a series of inputs, and checks whether the output is the expected outcome. Tests can be easily run on sequential programs, and are routinely used to check correctness, usually every time the code is changed. However, in the multicore domain, testing is a challenging task. Given a program and a test harness, there are numerous interleavings of the program, often exponentially increasing with the length of the runs, making even testing the program for a single input an extremely intractable problem. While deterministic languages alleviate this problem, as mentioned earlier, there will continue to be codes that require non-determinism or are written in non-deterministic languages for various reasons. For such codes, an important problem will be to effectively search the space of interleavings, testing perhaps only a fraction of them, but yet assuring high likelihood of finding errors. For instance, the Chess tool from Microsoft Research searches only those interleavings obtained using k context switches, for a small value of k [110].

Second, as previously described, even as we transition to disciplined languages, programs will likely have components that adhere to different levels of this discipline; e.g., deterministic components, non-deterministic but data-race-free components with locks, and even low-level shared memory legacy components. A rational composition of such modules into a single program that can be analyzed for correctness will be crucial in furthering the applicability and adoption of the disciplined programming languages we propose.

We advocate addressing the above programs by first building an understanding of the *high-level synchronization intentions* that help programmers co-ordinate the parallelism and simplify reasoning about the correctness of their code. We believe that this understanding, inferred by detailed bug and code analysis, will prove useful in addressing both the testing and the composition problems described above. In testing a concurrent program, we plan to use the high-level synchronization intentions to effectively prune the space of interleavings, concentrating only on those that violate these intentions as they are more likely to have errors. The high-level synchronization intentions will also help build a language to summarize the high-level synchronizations for the less disciplined code so that we can encapsulate them, verify them, and compose them with newly written disciplined code, while assuring correctness of the entire program.

Understanding High-level Synchronization Intentions:

Data-race freedom is, of course, one of the primary aspects for a correct concurrent program. However, data-races are not the only problems for correctness of concurrent interaction. There are several higher-level synchronization mechanisms that programmers intend to enforce, the failure of which lead to serious and hard-to-detect concurrency errors.

Consider the snippet of a concurrent (shared-memory) program given in Figure 5. First, notice that if the lock acquisitions and releases are not present, there would be data-races in the program. In fact, we have found that programmers often protect accesses to every shared variable using a lock for that variable, making sure they possess the lock before they access the variable. While this does remove data-races, it does not remove the high-level errors that may exist in the code. For instance, for the program in Figure 5, there is a high-level *atomicity* violation, where the procedure executed by Thread 1 is being non-trivially interrupted by Thread 2, causing a common error. In fact, a recent survey on concurrency errors [111] shows that a majority of errors in concurrent programs (~65% of them) are caused by such high-level atomicity violations.

The error in Figure 5 occurs because the programmer intended to have the procedure in each thread work *atomically* (at least semantically) from other threads. However, due to the lack of high-level synchronization constructs to state such a mechanism, the programmer implements this mechanism using low-level locking routines and makes an error doing so.

There are several such high-level synchronization intentions. This paper has discussed determinism, the intention to write code that is parallel but produces the same effect no matter how the scheduling happens. Causality constraints are another—for instance, a piece of code in a thread may get scheduled only when another method in another thread finishes its execution. Programmers model this using wait-loops that wait on a variable to turn true signaling that the other method has finished, and often make mistakes (in fact, this contributes about 30% of the concurrency errors in the study in [111]).

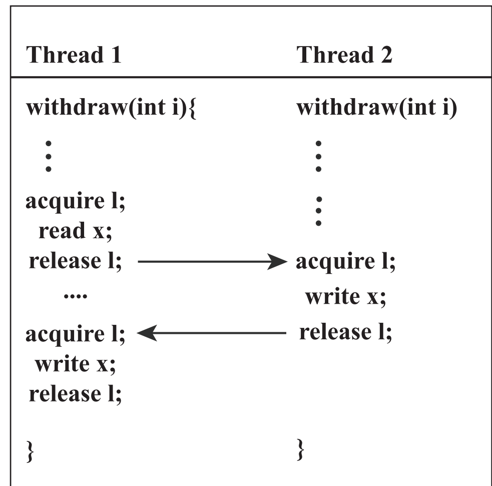

FIGURE 5 • A typical atomicity violation.

We are studying such high-level synchronization intentions. We propose to find these intentions in two ways. First, we are studying bug databases for concurrency errors and tracking their root cause to find the intentions that failed. Secondly, we think there is a need to conduct user-studies where programmers are given tasks to implement in parallel, and study and classify the high-level mechanisms they use to manage concurrency. We suggest that an understanding of these high-level mechanisms will be crucial in understanding the way programmers manage concurrency, will enable documentation, and will enable effective testing mechanisms and encapsulation techniques.

Effective Testing of Concurrent Programs:

Building effective testing for concurrent programs against test inputs is crucial to ensure reliable software. We propose the following two approaches to build effective testing techniques that avoid enumeration of all interleavings in order to get full coverage on a particular test input.

a) First, we advocate using the high-level synchronization intentions as a *generic specification* for testing. Given a test input, monitoring a run and checking whether it meets the high-level synchronization intentions of the programmer yields an effective way of finding concurrency errors. For example, by marking boundaries of methods and procedures, we can check for atomicity violations, and report them to the user [112, 113, 114]. In fact, we think that we can considerably reduce the number of interleavings that are checked. We suggest that effectively finding interleavings that violate synchronization intentions can be a powerful and scalable approach to testing concurrent programs. For instance, given a particular run on a test, we advocate predicting alternate executions from this run that will violate synchronization intentions (like atomicity) and execute these tests to force errors.

b) Disciplined programming models that we have proposed also require effective testing technology. We propose to utilize the discipline enforced in our models to reduce the cost of testing. For example, programs

where portions of the code are statically assured to be deterministic need to be run only once, as other interleavings will yield the same test results. Data-race freedom can also be used to reduce interleavings (for instance, interleavings for race-free programs need to switch context only at synchronization points to achieve full interleaving coverage). We believe that other forms of disciplined interaction of concurrent threads can be fruitfully exploited to build testing tools that scale to large programs.

Insulating and Composing Programs:

An important aspect of transitioning to disciplined concurrent code is that there will be parts of legacy code that we may want to reuse, rewriting only certain parts of the program using disciplined programming. Also, even when programming afresh with disciplined concurrency, we may want to allow certain libraries and code written by experts, which may use low-level and complex synchronization mechanisms (such as wait-free algorithms) that do not strictly adhere to the application-code level discipline. We believe that it will be very important to *compose* code written at different disciplinary levels, but at the same time be able to encapsulate their effects and reason about the correctness of the whole program.

A usable scheme for implementing such a compositional mechanism will need a framework with several components: (a) a mechanism, both at the programming language level and at the runtime level, to effectively sandbox and encapsulate modules and libraries, (b) a language for expressing contracts that capture the effect of these modules and libraries, in particular the way they access global data and the assurance they give on their consistency, atomicity guarantees of their methods, etc., and (c) ways to formally reason about these contracts to show that the entire code, some portions consisting of disciplined code and some consisting of encapsulated libraries with undisciplined programming, is correct. For example, defining such a framework for deterministic programs will expand the deterministic programming paradigm to include modules and libraries that have encapsulated non-determinism within them. In general, such techniques for annotating and composing modules will greatly enhance the appeal of the programming paradigms that we propose in this paper.

6 Conclusions

We have described an ambitious research agenda that aims to make client parallel programming synonymous with programming. Our key themes are a transformative change from current low-level bug-prone programming models to a *disciplined parallel programming* ecosystem, and a broad-based attack on parallelism at *all levels of the stack* that focuses on enabling performance, scalability, and support for programmability. We believe that the breadth and depth of research expertise at Illinois is ideally suited for such an agenda—we bring together research in programming languages, compilers, autotuners, runtime systems, hardware architecture, refactoring tools, and formal methods, along with research in programming patterns and application domains that are expected to trigger the killer applications of the future.

Our deterministic-by-default object-oriented shared memory languages, parallel operators, metaprogramming and autotuning systems, domain-specific environments, actor based languages, and correctness checking tools aim to establish a disciplined parallel programming paradigm that will make it easy to program a variety of future client applications. Our work in compilers, autotuners, runtime systems, and hardware will support and reward this discipline, while ensuring that we continue to support performance, scalability, and productivity with current software systems.

Driving the above agenda is a human-centric vision of future consumer applications, backed up by research on application technologies to enable quantum-leaps in immersive visual realism, reliable natural-language processing, and robust teleprescence. Our strategy of integrated applications and systems research will ensure we have the right testbed for evaluating, refining and ultimately proving our ideas on client parallel programming.

References

1 Parallel@Illinois. University of Illinois at Urbana-Champaign. http://www.parallel.illinois.edu/

2 SqueakCMI. University of Illinois at Urbana-Champaign. http://www.squeakcmi.org/

3 E.W. Dijkstra, “Letters to the editor: go to statement considered harmful,” *Communications of the ACM*, vol. 11, no.3, 1968, pp. 147-148.

4 J.C. Phillips, G. Zheng, S. Kumar, and L.V. Kale, “NAMD: Biomolecular Simulation on Thousands of Processors,” *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, September 2002, pp. 1-18.

5 A. Bhatele, S. Kumar, C. Mei. J.C. Phillips, G. Zheng, and L.V. Kale, “Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms,” *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

6 L.V. Kale and S. Krishnan, “Charm++: Parallel Programming with Message-Driven Objects,” *Parallel Programming using C++*, Gregory V. Wilson and Paul Lu, Eds, MIT Press, 1996, pp. 175-213.

7 “NVIDIA Appoints First CUDA Center of Excellence,” NVIDIA Corporation, June 30, 2008. http://www.nvidia.com/object/io_1214807636303.html

8 S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu , “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA,” *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

9 C. I. Rodrigues, J. Stone, D. Hardy, W. W. Hwu, “GPU Acceleration of Cutoff-Based Potential Summation,” *ACM Computing Frontier Conference 2008*, Italy, May 2008.

10 S.S. Stone, J.P. Haldar, S.C. Tsao, W.W. Hwu, Z.P. Liang, B.P. Sutton, “Accelerating Advanced MRI Reconstruction using GPUs,” *ACM Computing Frontier Conference 2008*, Italy, May 2008.

11 E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Reading: Addison-Wesley, 1995.

12 Workshop on Patterns in High-Performance Computing, University of Illinois at Urbana-Champaign, May 4-6, 2005. http://charm.cs.uiuc.edu/patHPC/

13 H-J. Boehm and S.V. Adve, “Foundations of the C++ Concurrency Memory Model,” to appear in the *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

14 J. Manson, W. Pugh, and S.V. Adve, “The Java Memory Model,” *Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL)*, January 2005.

15 E.W. Dijkstra, “On the role of scientific thought,” Selected writings on Computing: A Personal Perspective, New York: Springer-Verlag, 1982, pp. 60-66.

16 M. Snir, “Parallel Programming Language 1, V0.9 (Draft),” Technical Report No. UIUCDCS-R-2006-2696, University of Illinois at Urbana-Champaign, 2006.

17 R. Bocchino and V. Adve, “Formal Definition and Proof of Soundness for Core Deterministic Parallel Java,” Technical Report No. UIUCDCS-R-2008-2980, University of Illinois at Urbana-Champaign, 2008.

18 R. Bocchino, V. Adve, S. Adve, and M. Snir, “Parallel Programming Must Be Deterministic By Default,” Computer Science Technical Report #UIUCDCS-R-2008-3012, University of Illinois at Urbana-Champaign, October 2008.

19 J. Guo, G. Bikshandi, B. Fraguela, M. Garzarán, and D. Padua, “Programming with Tiles,” *Proc. of the 2008 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008, pp. 111-122.

20 G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. Fraguela, M. Garzarán, D. Padua, and C. von Praun, “Programming for Parallelism and Locality with Hierarchically Tiled,” *Proc. of the of the 2006 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006, pp. 48-57.

21 J.C. Brodman, B.B. Fraguela, M.J. Garzaran, and D. Padua, “New Abstractions For Data Parallel Programming,” University of Illinois at Urbana-Champaign, Department of Computer Science, Technical Report No. UIUCDCS-R-2008-3014, November 2008.

22 M. Puschel, P.J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo, “SPIRAL: Code Generation for DSP Transforms,” *Proceedings of the IEEE special issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, February 2005, pp. 232-275.

23 B.A. Garber, D. Hoeflinger, X. Li, M. Garzarán, and D. Padua, “Generation of a Parallel Sorting Algorithm,” in the *Next Generation Software Workshop*, in conjunction with *IPDPS*, April 2008.

24 X. Li, M. Garzarán, and D. Padua, “Optimizing Sorting with Genetic Algorithms,” in *Proc. of the International Symposium on Code Generation and Optimization*, March 2005, pp. 99-110,.

25 G. Agha, “Concurrent object-oriented programming,” *Communications of the ACM*, vol. 33, no. 9, 1990, pp. 125-141.

26 W. Kim and G. Agha, “Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages,” *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, 1995.

27 M. Astley, D.C. Sturman, and G. Agha, “Customizable middleware for modular distributed software,” *Communications of the ACM*, vol. 44, no. 5, 2001, pp. 99-107.

28 N. Venkatasubramanian, C.L. Talcott, and G. Agha, “A Formal Model for Reasoning about Adaptive Qos-Enabled Middleware,” *ACM Transactions Software Engineering Methodology*, vol. 13, no. 1, 2004, pp. 86-147.

29 Prvulovic and J. Torrellas, “ReEnact: Using Thread-Level Speculation to Debug Data Races in Multithreaded Codes,” *30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.

30 M. Cintra, J.F. Martínez, and J. Torrellas, “Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems,” *27th Annual International Symposium on Computer Architecture (ISCA)*, June 2000.

31 J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas, “Energy-Efficient Thread-Level Speculation on a CMP,” *IEEE Micro Magazine, Special Issue: Micro’s Top Picks from Computer Architecture Conferences*, January-February 2006.

32 P. Montesinos, L. Ceze, and J. Torrellas, “DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently,” *35th Annual International Symposium on Computer Architecture (ISCA)*, June 2008.

33 P. Montesinos, M. Hicks, S. King, and J. Torrellas, “Capo: Abstractions and Software-Hardware Interface for Hardware-Assisted Deterministic Multiprocessor Replay”, *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009.

34 P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, “iWatcher: Efficient Architectural Support for Software Debugging,” *31th Annual International Symposium on Computer Architecture (ISCA)*, June 2004.

35 J. Tuck, W. Ahn, L. Ceze, and J. Torrellas, “SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization,” *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

36 V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, “LLVA: A Low-level Virtual Instruction Set Architecture,” *Proceedings of the 36th annual ACM/IEEE International Symposium on Microarchitecture (MICRO-36)*, San Diego, California, December 2003.

37 J.E. Smith, T. Heil, S. Sastry, T. Bezenek, “Achieving High Performance via Co-designed Virtual Machines,” *Proc. Int’l Workshop on Innovative Architecture (IWIA)*, 1999.

38 P-P Sloan, J.D. Hall, J.C. Hart, and J. Snyder, “Clustered Principle Components for Precomputed Radiance Transfer,” *Proc. SIGGRAPH 2003, ACM Transactions on Graphics*, vol. 22, no. 3, July 2003, pp. 382-391.

39 T. Foley and J. Sugerman, "kD-tree Acceleration Structures for a GPU Raytracer," *Proc. Graphics Hardware*, 2005, pp. 15-22.

40 N.A. Carr, J. Hoberock, K. Crane, and J.C. Hart, "Fast GPU Ray Tracing of Dynamic Meshes Using Geometry Images," *Proc. Graphics Interface*, 2006, pp. 203-209.

41 A. Godiyal, J. Hoberock, M. Garland, and J.C. Hart, "Rapid Multipole Graph Drawing on the GPU," *Proc. Graph Drawing*, September 2008.

42 K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-Time KD-Tree Construction on Graphics Hardware," to appear in *Proc. SIGGRAPH Asia, ACM Transactions on Graphics*, vol. 27, no. 5, Dec 2008.

43 V. Punyakanok, D. Roth, and W. Yih, "The Necessity of Syntactic Parsing for Semantic Role Labeling," *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, (2005.

44 V. Punyakanok, D. Roth, and W. Yih, "The Importance of Syntactic Parsing and Inference in Semantic Role Labeling," *Computational Linguistics*, 2008.

45 X. Li, P. Morie, and D. Roth, "Semantic Integration in Text: From Ambiguous Names to Identifiable Entities," *AI Magazine Special Issue on Semantic Integration*, 2005.

46 E. Bengtson and D. Roth, "Understanding the Value of Features for Coreference Resolution," *EMNLP, Empirical Method in NLP*, 2008.

47 A. Klementiev and D. Roth, "Weakly Supervised Named Entity Transliteration and Discovery from Multilingual Comparable Corpora," *Proc. of the Annual Meeting of the ACL*, 2006) .

48 R. Braz, R. Girju, V. Punyakanok, D. Roth, and M. Sammons, "An Inference Model for Semantic Entailment in Natural Language," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2005.

49 D. Roth and W. Yih, "Global Inference for Entity and Relation Identification via a Linear Programming Formulation," *Introduction to Statistical Relational Learning*, 2007.

50 Z. Yang, K. Nahrstedt, and R. Bajcsy "TEEVE: The Next Generation Architecture for Tele-Immersive Environments," *IEEE Int'l Symp. on Multimedia (ISM)*, UC Irvine, CA, December 2005.

51 Z. Yang, W. Wu, K. Nahrstedt, G. Kurrilo, and R. Bajcsy, "ViewCast: View Dissemination and Management for Multiparty 3D Tele-immersive Environments," *ACM Multimedia*, September 2007, Augsburg, Germany.

52 W. Wu, Z. Yang, K. Nahrstedt, and I. Gupta, "Towards Multisite Collaboration in 3D Tele-immersive Environments," *IEEE International Conference on Distributed Computer Systems (ICDCS)*, Beijing, China, June 2008.

53 D. Lea, *Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition)*, Addison-Wesley Professional, 2006.

54 T.G. Mattson, B.A. Sanders and B.L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2004.

55 K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W. Lester P.J. Shalf, S.W. Williams, and K.A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2006-183, December 18, 2006.

56 L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, 1998, pp. 46-55.

57 J. Gosling, *The Java Language Specification*, Addison-Wesley Professional, 2000.

58 S. Wiltamuth and A. Hejlsberg, "C# language specification," *MSDN Library*, Microsoft Corporation, Dec 2002. Accessed April 2, 2003. http://msdn.microsoft.com/en-us/library/ms228593.aspx

59 P.A. Emrath, S. Ghosh, and D.A. Padua, "Detecting Nondeterminacy in Parallel Programs," *IEEE Software*, vol. 9, no. 1, January 1992.

60 T.R. Allen and D.A. Padua, "Debugging Fortran on a Shared Memory Machine," *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987, pp. 721-727.

61 P.A. Emrath and D.A. Padua, “Automatic detection of nondeterminacy in parallel programs,” in *PADD ‘88: Proceedings of the 1988 ACM SIGPLAN* and *SIGOPS Workshop on Parallel and Distributed Debugging*, November 1988.

62 P.A. Emrath, S. Ghosh, and D.A. Padua, “Event synchronization analysis for debugging parallel programs,” *Supercomputing ‘89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, August 1989.

63 S.V. Adve, M.D. Hill, B.P. Miller, and R.H.B. Netzer, “Detecting Data Races on Weak Memory Systems,” *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 234-243.

64 D. Shasha and M. Snir, “Efficient and correct execution of parallel programs that share memory,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 2, April 1988.

65 S.V. Adve and M.D. Hill, “Weak ordering—a new definition,” *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, May 1990, pp. 2-14.

66 S.V. Adve, *Designing Memory Consistency Models for Shared-Memory Multiprocessors*, Ph.D. Thesis, available as Computer Sciences Technical Report #1198, University of Wisconsin, Madison, December 1993.

67 C. Huang and L.V. Kale, “Charisma: Orchestrating Migratable Parallel Objects,” *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC),* July 2007.

68 L.V. Kale, M. Hills, and C. Huang, “An Orchestration Language for Parallel Objects,” *Proceedings of Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 04)*, Houston, Texas, October 2004.

69 J. DeSouza and L.V. Kale, “MSA: Multiphase Specifically Shared Arrays,” *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

70 S. Chakravorty, A. Becker, T. Wilmarth, and L.V. Kale, “A Case Study in Tightly Coupled Multi-Paradigm Parallel Programming,” *Proceedings of Languages and Compilers for Parallel Computing (LCPC ‘08)*, 2008.

71 The Fortress Language Specification, Version 1.0, Sun Microsystems Inc., 2008.

72 M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew. “Optimistic parallelism requires abstractions,” *PLDI ‘07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, June 2007.

73 Phillip Colella, “Defining Software Requirements for Scientific Computing,” 2004.

74 P.N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, *Titanium Language Reference Manual, Version 1.16.8*, Report 2004, Computer Science, UC Berkeley, http://www.cs.berkeley.edu/projects/titanium/doc/lang-ref.pdf

75 T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*, Wiley, 2005.

76 R.W. Numrich and J. Reid, “Co-array Fortran for parallel programming, *SIGPLAN Fortran Forum*, vol. 17, no. 2, 1998, pp. 1-31.

77 L.V. Kale, “Performance and Productivity in Parallel Programming via Processor Virtualization,” *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, February 14, 2004.

78 W. Hwu, K. Keutzer, T. Mattons, “The Concurrency Challenge,” *IEEE Design and Test of Computers,* July/August 2008, pp. 312-320.

79 R.C. Whaley, A. Petitet, and J. Dongarra, “Automated Empirical Optimization of Software and the ATLAS project,” *Parallel Computing*, vol. 27, nos. 1-2, 2001, pp. 3-35.

80 M. Frigo and S.G. Johnson, “The Design and Implementation of FFTW3,” *Proceedings of the IEEE special issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, February 2005, pp. 216-231.

81 Matlab FFTW documentation.
http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/ref/fftw.html

82 S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzarán, D. Padua and K. Pingali, "A Language for the Compact Representation of Multiples Program Versions," *Proc. of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, October 2005. Lecture Notes in Computer Science—Springer Series #4339, 2005.

83 S. Ryoo, C. Rodrigues, S. Stone, S.S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu, "Program Optimization Space Pruning for a Multithreaded GPU," *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

84 Matlab Parallel Computing Toolbox 4.0. http://www.mathworks.com/products/parallel-computing/index.html?s_cid=HP_FP_ML_parallelcomptbx

85 Apple Core Audio Preliminary Documentation, May 2001. http://developer.apple.com/audio/pdf/coreaudio.pdf

86 PureMVC Framework for AS3 (MultiCore Version), http://puremvc.org/component/option,com_wrapper/Itemid,161/

87 D. Dig, J. Marrero, and M.D. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," MIT Technical Report #MIT-CSAIL-TR-2008-057, September 30, 2008. Available at http://hdl.handle.net/1721.1/42841

88 P-H. Chang and G. Agha, "Towards Context-Aware Web Applications," *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2007)*, Lecture Notes in Computer Science—Springer Series #4531, 2007, pp. 239-252.

89 P. Haller and M. Odersky, "Actors that Unify Threads and Events," *The 9th International Conference on Coordination Models and Languages (COORDINATION 2007)*, Lecture Notes in Computer Science—Springer Series #4467, 2007, pp. 171-190.

90 C.A. Varela and G. Agha, "Programming Dynamically Reconfigurable Open Systems with SALSA," *16th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications: Intriguing Technology Track*, ACM SIGPLAN Notices #36, 2001, pp. 20-34.

91 J. Armstrong, "The development of Erlang," *International Conference on Functional Programming*, ACM, 1997, pp. 196-203.

92 Web link for E language. http://erights.org/

93 C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*, Palo Alto, California, March 2004.

94 V. Vardhan, D. Sachs, W. Yuan, A.F. Harris III, S. Adve, D. Jones, R. Kravets, and K. Nahrstedt," GRACE-2: Integrating Fine-Grained Application Adaptations with Global Adaptation for Saving Energy," to appear in the *International Journal of Embedded Systems (IJES), Special Issue on "Low-Power Real-Time Embedded Computing."* Invited paper. Extended version of a paper in the *Workshop on Power Aware Real-Time Computing (PARC)*, 2005.

95 W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets, "Design and Evaluation of A Cross-Layer Adaptation Framework for Mobile Multimedia Systems," *Proceedings of the SPIE/ACM Multimedia Computing and Networking Conference (MMCN'03)*, Santa Clara, California, January 2003, pp. 1-13.

96 W.A. Wulf and S.A. McKee, *Hitting the Memory Wall: Implications of the Obvious*, December 1994.

97 K. Strauss, X. Shen, and J. Torrellas, "Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors," *40th International Symposium on Microarchitecture (MICRO)*, December 2007.

98 Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," *34th Annual International Symposium on Computer Architecture (ISCA)*, June 2007.

99 V. Krishnan and J. Torrellas, "Hardware and Software Support for Speculative Execution of Sequential Binaries On a Chip-Multiprocessor," *International Conference on Supercomputing (ICS)*, July 1998.

100 J.F. Martínez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

101 K. Strauss, X. Shen, and J. Torrellas, "Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessor," *33rd Annual International Symposium on Computer Architecture (ISCA)*, June 2006.

102 L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," *33rd Annual International Symposium on Computer Architecture (ISCA)*, June 2006.

103 M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *29th Annual International Symposium on Computer Architecture (ISCA)*, May 2002.

104 S.R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou, "ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing," *38th International Symposium on Microarchitecture (MICRO)*, November 2005.

105 G.C. Hunt, and J.R. Larus, "Singularity: rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, April 2007, pp. 37-49.

106 J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure Virtual Architecture: A Safe Execution for Commodity Operating Systems," *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.

107 M. Frigo, C.E. Leiserson, and K.H. Randall, "The implementation of the Cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, May 1998, pp. 212-223.

108 J. Reinder, *Intel Threading Building Blocks—Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly, 2007.

109 S. Kumar, C.J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, Jun. 2007, pp. 162-173.

110 M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, ACM, 2007, pp. 446-455,

111 S. Lu, S. Park. E. Seo, Y. Zhou, "Learning from Mistakes—A Comprehensive Study on Real World Concurrency Bug Characteristics," *Proceedings of the 13th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'08)*, March 2008.

112 A. Farzan and P. Madhusudan, "Causal Atomicity," *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, Lecture Notes in Computer Science—Springer Series #4144, 2006, pp. 315-328.

113 A. Farzan and P. Madhusudan, "Monitoring Atomicity in Concurrent Programs," *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*, Lecture Notes in Computer Science—Springer Series #5123, 2008, pp. 52-65.

114 F. Chen, T.F. Serbanuta, and G. Rosu, "jPredictor: a predictive runtime analysis tool for Java," Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM, 2008, pp. 221-230.

www.upcrc.illinois.edu

PARALLEL@ILLINOIS

www.parallel.illinois.edu