Ubiquitous Parallel Computing from Berkeley, Illinois and Stanford

Bryan Catanzaro, Armando Fox, Kurt Keutzer, David Patterson, and Bor-Yiing Su (UC Berkeley)

Marc Snir (U Illinois at Urbana-Champaign)

Kunle Olukotun, Pat Hanrahan and Hassan Chafi (Stanford University)

Abstract

This paper presents a brief survey of the research on multicore computing that is performed at three research centers: ParLab at Berkeley, UPCRC-Illinois and the Pervasive Parallel Laboratory at Stanford. This research is focused on client computing and covers applications, programming environments and architecture.

Introduction

For more than three decades, the microelectronics industry has followed the trajectory set by Moore's Law of exponentially decreasing feature size and increasing circuit density. The microprocessor industry has leveraged this evolution to increase uniprocessor performance, decreasing cycle time and increasing the average number of executed instructions per cycle (IPC).

This evolution stopped a few years ago. Power constraints are resulting in stagnant clock rates, and new micro-architecture designs yield very limited improvements in IPC. Instead, the industry uses continued increases in transistor counts to populate chips with an increasing number of cores – multiple independent processors. This change has profound implications for the IT industry. In the past, each generation of hardware brought increased performance on existing applications, with no code rewrite, and enabled new, performance hungry applications. This is still true *but only for applications written to run in parallel, and written to scale to an increasing number of cores.*

This is less of a problem for servers. Server applications are usually throughput oriented, and can leverage more parallelism by serving a larger number of clients or by consolidating server functions onto fewer systems. Many server applications spent most of their cycles on a small number of services, such as databases or web servers; increased parallelism in these few services boosts the performance of many server applications.

Parallelism for client and mobile applications is a harder problem. The applications are turnaround oriented, and the use of parallelism to reduce response time requires more algorithmic work. Client applications are more diverse – it is not sufficient to run in parallel a few services.

Furthermore, mobile systems are power constrained; and improved wireless connectivity enables shifting computations to the server (or the cloud).

This suggests the following questions:

1. Will compelling new client applications emerge that require significant added client performance?
2. Can such applications leverage parallelism?
3. Can we develop a programming environment that enables a large programmer community to develop parallel codes?
4. Can multicore architectures and their software scale to the hundreds of cores that hardware will be able to support in a decade?

A negative answer to any of these questions will imply significant upheavals in the business model of many IT companies. We believe that the answer to all four questions is positive; however, a timely solution to these problems requires a significant acceleration of the transfer of research ideas into practice.

In the next three sections, we present the research done at Berkeley, Illinois, and Stanford to address these problems. These centers share common themes. All three have specific applications they are parallelizing, rather than developing technology for undefined future applications as is traditional in research. All focus primarily on client computing, designing technology that can work well up through hundreds of cores. All three reject a single solution approach, assuming instead that software is developed by teams of programmers with different specialties, some with more domain expertise, and some with deeper computer science knowledge; different specialists need different sets of tools.

Although they share assumptions, the diversity of these three centers is seen in:

- The role of parallel programming patterns in shaping parallel software architecture
- A focus on different programming environments (Python+frameworks, safe parallel C++ or Scala+embedded domain specific languages);
- A focus on different approaches for the production of parallel code (static compilation and optimization, run-time compilation and optimization, autotuning);
- Architecture support for parallelism (isolation and measurement vs. communication and synchronization);
- And even how to evaluate parallel architectures (FPGAs and/or software simulation).

ParLab at Berkeley

Recognizing the difficulty of the multicore challenge, Intel and Microsoft invited 25 universities to propose parallel computing centers. We were delighted they selected us in 2008 and that another six companies (so far) joined Intel and Microsoft as affiliates: National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. Embracing open-source software and welcoming all

collaborators, the Berkeley Parallel Computing Laboratory is a team of 50 PhD students and a dozen faculty leaders from many fields who work closely together towards our ambitious goal of making parallel computing productive, performant, energy-efficient, scalable, portable, and at least as correct as sequential programs [1].

Patterns and Frameworks

Many presume the challenge is for today's programmers to become efficient parallel programmers with only modest training. Our goal is quite different. It is to enable the productive development of efficient parallel programs by *tomorrow's programmer*s. As mentioned above, we believe future programmers will be either domain experts or sophisticated computer scientists as few domain experts have the time to develop performance programming skills and few computer scientists have the time to develop domain expertise. The latter group will create frameworks and software stacks that enable domain experts to develop applications without understanding details of the underlying platforms.

We believe that the key to the design of parallel programs is software architecture, and the key to their efficient implementation is frameworks. In our approach, the basis of both is *design patterns* and a *pattern language*. Borrowed from civil architecture, the term *design pattern* means solutions to recurring design problems that domain experts learn. A *pattern language* is simply an organized way of navigating through a collection of design patterns to produce a design. Our Pattern Language [2] consists of a series of *computational patterns* drawn largely from thirteen motifs[1]. We see these as the fundamental software building blocks that are composed using a series of *structural patterns* drawn from common software architectural styles, such as pipe-and-filter.

A software architecture is then the hierarchical composition of computational and structural patterns, which we refine using lower-level design patterns. This software architecture and its refinement, although useful, are entirely conceptual. To implement the software, we rely on frameworks. We define a *pattern-oriented software framework* as an environment built around a software architecture in which customization is *only* allowed in harmony with the framework's architecture. For example, if based on pipe-and-filter, then customization involves only modifying pipes or filters.

We envision a two-layer software stack. In the *productivity layer,* domain experts principally develop applications using *application frameworks*. In the *efficiency layer,* computer scientists develop these high-level application frameworks as well as other supporting software frameworks in the stack. Application frameworks have two advantages: First, the application programmer works within a familiar environment using concepts drawn from the application domain. Second, we prevent expression of many annoying problems of parallel programming: non-determinism, races, deadlock, starvation, and so on. To reduce such problems *inside* these frameworks, we use dynamic testing to execute problematic schedules [3].

SEJITS: Bridging the Productivity-Efficiency Gap

Framework writers are more productive when they write in high level "productivity layer'' languages (PLLs) like Python or Ruby whose abstractions match the application domain; studies

have reported factors of 3-10 fewer lines of code and 3-5 times faster development when using PLLs rather than "efficiency level languages'' (ELLs) like C++ or Java [4,5,6]. However, PLL performance may be orders of magnitude worse than ELL code, in part due to interpreted execution.

We are developing a new approach to bridge the gap between PLLs and ELLs. Modern scripting languages like Python and Ruby include facilities to allow *late binding* of an ELL module to execute a function of the PLL. In particular, *introspection* allows a function to inspect its own abstract syntax tree (AST) when first called, to determine whether the AST can be transformed into one that matches a computation performed by some ELL module. If it can, PLL *metaprogramming* support then *specializes* the function at runtime by generating, compiling, and linking the ELL code to the running PLL interpreter. Indeed, the ELL code generation may include syntax-directed translation of the AST into the ELL. If the AST cannot be matched to an existing ELL module or the module targets the wrong hardware, the PLL interpreter just continues executing as usual. Note that this approach preserves portability since it does not modify the source PLL program.

While just-in-time (JIT) code generation and specialization is well established with Java and Microsoft .NET, our approach *selectively* specializes only those functions that have a matching ELL code generator, rather than having to JIT the entire PLL. Also, the introspection and metaprogramming facilities allow the specialization machinery to be *embedded* in the PLL directly rather than having to modify the PLL interpreter. Hence, we call our approach *SEJITS*: *selective, embedded, just-in-time specialization* [7].

SEJITS helps domain experts use the work of efficiency programmers. Efficiency programmers can "drop" new modules specialized for particular computations into the SEJITS framework, which will make runtime decisions when to use it. By separating the concerns of ELL and PLL programmers, SEJITS frees *both* the PLL and ELL programmers to concentrate on their respective specialties.

An example is Copperhead, which is a data-parallel Python dialect. Following the SEJITS philosophy, every Copperhead program is valid Python, executable by the Python interpreter. However, when it consists of data parallel operations taken from a Python library, the Copperhead runtime specializes the resulting program into efficient parallel code. Examples of data-parallel operations include map, reduce, scatter, gather, sort, scan, zip, collect, split, and join. Parallelism is expressed through data, for example, by splitting a sequence into independent subsequences. Synchronization is also inferred from data access patterns, such as when sequences are joined. When a Copperhead procedure is executed, the Copperhead runtime specializes the program into a series of data parallel kernels for a particular parallel platform, and then dispatches the resulting specialized computation on parallel hardware. Preliminary experiments show encouraging efficiency, while enabling parallel programming at a much higher level of abstraction [7]. Our goal is to specialize an entire image processing computation, such as in Figure 1 below.

Platforms and Applications of 2020

We believe future applications will have footholds in both mobile clients and in cloud computing. Many client applications are downloaded and run inside a browser, so we are investigating parallel browsers [8].

We note that both client and cloud are concerned with responsiveness and power efficiency. The application client challenge is responsiveness while preserving battery life given a variety of platforms. The application cloud challenge is responsiveness and throughput while minimizing cost in a pay-as-you-go environment [9]. In addition, cloud computing *providers* want to lower costs, which are primarily power and cooling.

To illustrate the client/cloud split, suppose a domain expert wants to create an application that suggests the name of a person walking toward you. Once identified, the client device then whispers the information to you. Responsiveness is key, as seconds separate timely from irrelevant. Depending on wireless connectivity and battery state, a "Name Whisperer" could send photos to the cloud to do the search, or the client could search locally from a cache of people you've met.

Object Recognition Systems

Clearly, an object recognition system will be a major component of such an application. Figure 1 shows the computational flow of a local object recognizer [10], which gets good results on vision benchmarks. Although high quality, it is computationally intensive: it takes 5.5 minutes to identify five kinds of objects in a 0.15MB image, which is far too slow for our application.

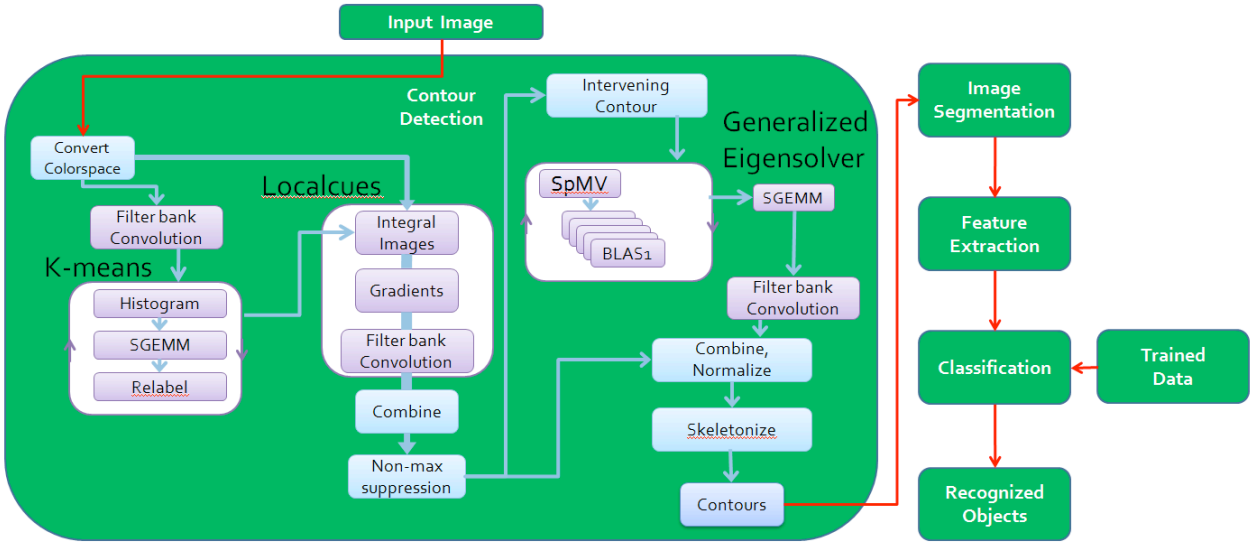


Figure 1: Computation flow of object recognition using regions.

The main goal of an application framework is to help application developers design their applications efficiently. For computer vision, computations include image contour detection, texture extraction, image segmentation, feature extraction, classification, clustering, dimensionality reduction, and so on. There are many algorithms proposed for each computation, with different trade-offs. Each method corresponds to an application pattern. The application framework is used to integrate these patterns together. As a result, the application developers can compose their applications by arranging the computations together, and let the framework figure out which algorithm to use, how to set parameters for the algorithm, and how to communicate between different computations.

Figure 1 shows the four main computations of the object recognizer. We can try out application patterns for different contour detectors, image segmentors, feature extractors, and trainer/classifier and find the composition that achieves the highest accuracy. An alternative is to only specify the composition of the computation, and let the application framework choose the proper application pattern to realize the computation.

We choose application patterns manually. Now we investigate the contour detector, which uses 75% of the time, as an example. Among all contour detectors, the gPb algorithm achieves the highest accuracy. Figure 1 summarizes our version of the gPb algorithm. The major computational bottlenecks are the localcue computation and the generalized eigensolver. For the localcue computation, we replace the explicit histogram accumulation by integral image, and apply parallel scan for realizing integral image. For the generalized eigensolver, we proposed a highly parallel SpMV kernel and investigated appropriate reorthogonal approaches for the Lanczos algorithm. By selecting good patterns to form a proper software architecture that reveals parallelism and then exploring appropriate algorithmic approaches and parallel implementations within that software archtiecture, we accelerated contour execution time by 140X from 4.2 minutes to 1.8 seconds on a GPU by [11].

Performance Measurement and Autotuning

The prevailing hardware trend of dynamically improving performance with little software visibility has become counterproductive; software must adapt if parallel programs are going to be portable, fast, and energy-efficient. Hence, parallel programs must be able to understand and measure any computer so that they can adapt effectively. This perspective suggests architectures with transparent performance and energy consumption and *Standard Hardware Operation Trackers* (SHOT) [12]. SHOT enables parallel programming environments to deliver on the goals of portability, performance, and energy-efficiency. For example, we used SHOT to examine alternative data structures for the image contour detector, by examining realized memory bandwidth versus the data layout.

While SHOT helps evaluate alternatives, what mechanism adapts the software to the hardware? Autotuners produce high-quality code by generating many variants and measuring each variant on the target platform. The search process tirelessly tries many unusual variants of a particular routine. Unlike libraries, autotuners also allow tuning to the particular problem size. Autotuners also preserve clarity and help portability by reducing the temptation to mangle the source code to improve performance for a particular computer. Automatic creation of autotuners is an important research area [13].

The Copperhead specialization machinery has built-in heuristics that guide decisions about data layout, parallelization strategy, execution configuration, and so on. Autotuning allows the Copperhead specializer to gain efficiency portably, however, since the optimal execution strategy differs by platform.

The Architecture and OS of 2020

We expect the client hardware of 2020 will contain hundreds of cores in replicated hardware tiles. Each tile will contain one processor designed for instruction-level parallelism for sequential

code *and* a descendant of vector and GPU architectures for data-level parallelism. Task-level parallelism occurs across the tiles. The number of tiles per chip varies depending on cost-performance goals. Thus, although the tiles will be identical for ease of design and fabrication, the chip supports heterogeneous parallelism. The memory hierarchy will be a hybrid of traditional caches and software-controlled scatchpads [14]. We believe that such mechanisms for mobile clients will also aid servers in the cloud.

Since we rarely run just one program, the hardware will partition to provide performance isolation and security between multiprogrammed applications. Partitioning suggests restructuring systems services as a set of interacting distributed components. The resulting "deconstructed OS" *Tessellation* implements scheduling and resource management of *partitions* [15]. Applications and OS services (like file systems) run within partitions. Partitions are lightweight and can be resized or suspended with overhead comparable to a context swap. The OS kernel is a thin layer responsible for only the coarse-grain scheduling and assignment of resources to partitions and secure restricted communications among partitions. It avoids the performance issues with traditional microkernels by providing OS services through secure messaging to spatially co-resident service *partitions*, rather than context-switching to time-multiplexed service *processes*. User-level schedulers are used within a single partition to schedule application tasks onto processors across potentially multiple different libraries and frameworks, and the Lithe layer offers an interface to schedule independent libraries efficiently [16].

Par Lab Today

We have identified the architectural and software patterns and are using them in the development of applications in vision, music, health, speech, and browsers. These applications drive the rest of the research. We have two SEJITS prototypes and autotuners for several motifs and architectures. We have a 64-core implementation of our tiled architecture in FPGAs that we use for architectural experiments, including the first implementation of SHOT. FPGAs afforded a 250X increase in simulation time over software simulators, and the lengthier experiments often led to opposite conclusions. The initial Tessellation OS boots on our prototype and can create a partition, and Lithe has demonstrated efficient composition of code written using both TBB and OpenMP libraries.

UPCRC-Illinois

The Universal Parallel Computing Research Center at Illinois (UPCRC-Illinois) was established in 2008 as a result of the same competition that led to the establishment of ParLab. It involves about 50 faculty and students and is co-directed by Wen-mei Hwu and Marc Snir. UPCRC-Illinois is one of several major projects in parallel computing at Illinois (see http://parallel.illinois.edu) – continuing on a long tradition of leadership that started with the Illiac projects. The sections below describe work performed by many of the center's researchers; it was carefully edited by Cheri Helregel. The full list of -Illinois research leads and a detailed description of ongoing projects can be found at http://www.upcrc.illinois.edu.

Our work on applications is focused on the creation of compelling 3D web applications and human-centered interfaces. We have a strong focus on programming language, compiler and run-time technologies, aimed at supporting parallel programming models that provide simple, sequential-by-default semantics with parallel performance models, and that avoid concurrency bugs. Our significant architecture work is focused on the efficient support for shared memory with many hundreds of cores.

Applications

The 3D Internet will enable many new compelling applications. For example, the TEEVE 3D teleimmersion framework and its descendants [17] have been used to support remote collaborative dancing, remote Tai-Chi training, manipulation of virtual archeological artifacts, training of wheelchair-bound basketball players and gaming. Applications are limited by a low frame-rate, and inability to handle visually noisy environments. Broad real-time usage in areas such as multiplayer gaming or telemedicine requires several order of magnitude performance improvements in tasks such as the synthesis of 3D models from multiple 2D images; the recognition of faces and face expressions, of objects and gestures; and the rendering of dynamically created synthetic environments. Such tasks must execute on the mobile client, to reduce the impact of Internet latencies on real-time interactions. Many of the same tasks will be at the core of future human-centered ubiquitous computing environments that can understand human behavior and anticipate needs – although significant technical progress is needed [18].

We are developing new parallel algorithms for these tasks that can achieve required performance levels. We have implemented new parallel algorithms for Depth Image Based Rendering – creating a virtual view of a 3D object from a new angle, based on information from a depth camera and multiple optical cameras – and shown speedups of more than $\times 74$. We have implemented parallel versions of algorithms for the analysis of video streams; speedups in excess of $\times 400$ have been demonstrated on a hand tracking task and for B-spline image interpolation. The algorithms have been used for the NIST TREC Video Retrieval Challenge which requires the identification of specific events in surveillance videos. Speedups in excess of $\times 13$ have been achieved for the complete application. The algorithms are being collected into the publicly available Vivid library (http://libvivd.sourceforge.net) in the form of parallel functions callable from Python code 19 [1].

Spatial data structures form the backbone of many computationally-intensive 3D applications and data mining and machine learning algorithms. To support such applications, we are developing ParKD, a comprehensive framework for parallelized spatial queries and updates through scalable, parallel k-D tree implementations. ParkKD algorithms speed up the generation of k-D trees and generate k-D trees that enable efficient parallel rendering. We plan to integrate all these components into an end-to-end teleimmersive application.

Performance constraints of mobile platforms restrict the functionality of mobile application and lengthen their development time. For example, in order to achieve high quality, real-time graphics in interactive games, one needs to precompute data structures used for rendering. This restricts the number of supported scenarios and increases development time and cost. Less constrained

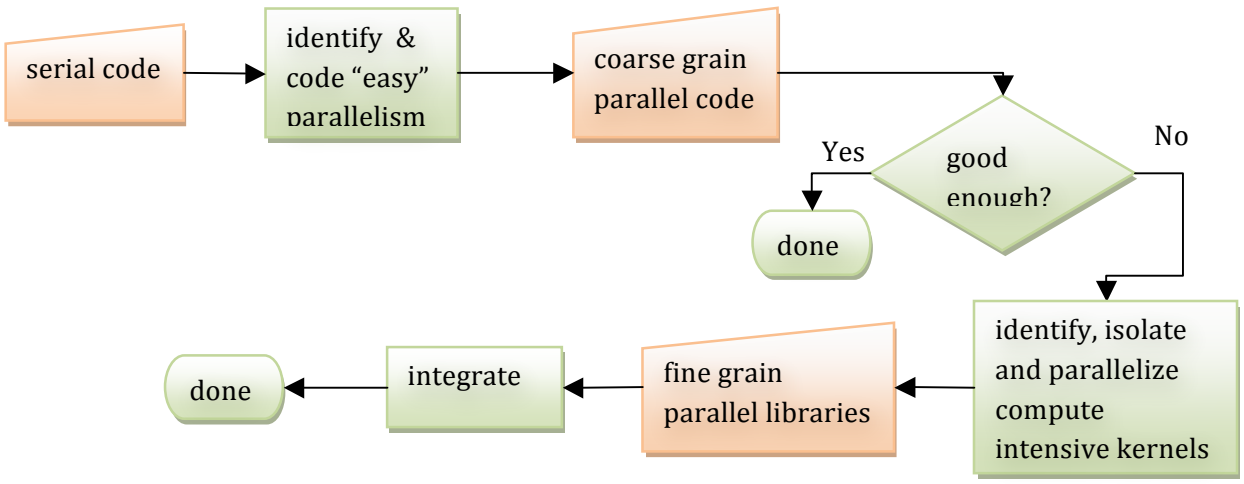
multiplayer games use lower quality graphics. An added advantage of our work will be to enable applications that provide better user experience and that can be developed faster.

Parallelism can also be used to achieve other goals, such as better Quality of Service, or improved security. Our work on Teeve shows how the use of multiple cores can simplify QoS provision for multiple concurrent real-time tasks.. Our work on the OP web browser [20] shows that parallelism can not only improve browser performance, but also reduce vulnerabilities by using compartmentalization.

Programming Environment

We distinguish between *concurrent programming* that focuses on problems where concurrency is part of the specification and *parallel programming* that focuses on problems where concurrent execution is used only for improving the performance of a computation. Concurrent programming is used in *reactive* systems, such as an operating system, graphical user interfaces, or on-line transaction processing system, where computations (or transactions) are triggered by nondeterministic, possibly concurrent, requests or events. Parallel programming is used in *transformational* systems, such as in scientific computing or signal processing, where an initial input (or an input stream) is mapped through a chain of (usually) deterministic transformations into an output (or an output stream). The prevalence of multicore platforms does not increase the need for concurrent programming and does not make it harder; it increases the need for parallel programming. It is our contention that parallel programming is much easier than concurrent programming; in particular, it is seldom necessary to use nondeterministic code.

Error! Reference source not found. presents a very schematic view of the parallel software creation process. One will often start with a sequential code, although starting from a high-level specification is preferred. It is often possible to identify “outer loop” parallelism where all or most of the sequential code logic can be encapsulated into a parallel execution framework (pipeline, master-slave, etc.), with little or no change in the sequential code. If this is not sufficient to achieve the desired performance then one identifies compute intensive kernels, encapsulates them into libraries, and tunes these libraries to leverage parallelism at a finer grain, including SIMD parallelism. While the production of carefully tuned parallel libraries will involve performance coding experts, simple, coarse-level parallelism should be accessible to all.



We propose to help "simple parallelism" by developing refactoring tools that help convert sequential code into parallel code written using existing parallel frameworks in C# or Java [21], and debugging tools that help identify concurrency bugs [22]. More fundamentally, we propose to support simple parallelism with languages that are *deterministic by default* and *concurrency safe* [23].

Languages such as Java and C# provide safety guarantees (e.g., type safety or memory safety) that significantly reduce the opportunities for hard to track bugs and improve programmer productivity. We plan to bring the same benefits to parallel and concurrent programming: a *concurrency safe* language prevents, by design, the occurrence of data races. As a result, the semantics of concurrent execution is well-defined, and hard-to-track bugs are avoided. A *deterministic by default* language will also ensure that, unless explicit nondeterministic constructs are used, any execution of a parallel code will have the same outcome as a sequential execution. This significantly reduces the effort of testing parallel code and facilitates the porting of sequential code. Our work on Deterministic Parallel Java (DPJ) [24] shows that both properties can be satisfied, even for modern object-oriented programming, without undue loss of expressiveness, and good performance. Moreover, the deterministic guarantee can be enforced with simple, compile-time type checking, which has major benefits. Although such a safe language requires some initial programming effort, these efforts are small compared with that of designing and developing a parallel program and can be significantly reduced via interactive porting tools [25]. Our tool, DPJizer, can infer much of the added information required by DPJ. The effort has a large long-term payoff in terms of greatly improved documentation, maintainability, and robustness under future software changes.

The development of high-performance parallel libraries requires a different environment that enables programmers to work more closely to the bare metal, and facilitate porting and tuning for

new platforms. We believe that such work should happen in an environment where compiler analysis and code development interact tightly, so that the performance programmer works *with* the compiler, not *against* it, as is often the case today. We are working on refactoring environments where code changes are mostly annotations that add information not otherwise available in the source code. The annotation information is extended by advanced compiler analysis to enable robust deployment of transformations such as data layout adjustment and loop tiling transformations. One particular goal of our Gluon work is to enable portable parallel code base for both fine-grained engines such as GPUs and coarser grained multi-core CPUs, as well as their associated memory hierarchies. Auto-tuning by manipulating the algorithm or the code is another technique we apply to this problem.

Parallel libraries are most easily integrated into sequential code when they follow the SIMD (Single Instruction stream, Multiple Data stream) model. Our work on Hierarchical Tiled Arrays (HTA)) [26] confirms earlier results showing that the effectiveness of this model is not confined to the traditional array computations and can be used to support a wide range of complex applications with good performance. The use of tiles as a first class object gives programmers good control of locality, granularity, and load balancing.

The integration of parallel libraries and frameworks into a concurrency safe programming environment requires a careful design of interfaces to ensure that the safety guarantees are not violated. We can use simple language features as specialized "contracts" at framework interfaces to ensure that client code using a parallel framework (with internal parallelism) does not violate assumptions made within the framework implementation [27]. Such an approach gives the (presumably) expert framework implementer freedom to use low-level and/or highly tuned techniques within the framework while enforcing a safety net for (presumably) less expert application programmers using the framework. The expert framework implementation can be subject to extensive testing and analysis techniques, including proofs of concurrency safety using manual annotations.

Architecture

The continued scaling of feature sizes will enable systems with hundreds of conventional cores, and possibly thousands of light-weight cores, within a decade. Current cache coherence protocols do not scale to such numbers. A major reason is that, with current protocols, each shared memory access by a core is considered to be a potential communication/synchronization with any other core. In fact, parallel programs communicate and synchronize in stylized ways. A key to shared memory scaling is the adjustment of coherence protocols to leverage the prevalent structure of shared memory codes for performance. We are exploring three approaches to do so. The *Bulk Architecture* is executing coherence operations in bulk, committing large groups of loads and stores at a time [28]. In this architecture, memory accesses appear to interleave in a total order, even in the presence of data races—which helps software debugging and productivity—while the performance is high through aggressive reordering of loads and stores within each group. The DeNovo architecture coherence and communication involves co-designing hardware with concurrency-safe programming models [29], and the Rigel architecture plans to shift more coherence activities to software [30]. In addition, a higher-level view of communication and synchronization across threads enables the architecture to help program development, e.g., by

supporting deterministic replay of parallel programs [31], or by tracking races in codes that do not prevent them by design [32].

The Stanford University Pervasive Parallelism Laboratory

In May 2008 Stanford University officially launched the Pervasive Parallelism Laboratory (PPL). The goal of the PPL is to make parallelism accessible to average software developers so that it can be freely used in all computationally demanding applications. The PPL pools the efforts of many leading Stanford computer scientists and electrical engineers with support from Sun Microsystems, NVIDIA, IBM, Advanced Micro Devices, Intel, NEC, and Hewlett Packard under a completely open industrial affiliates program. The open nature of the lab allows other companies to join the effort and does not provide any member company with exclusive intellectual property rights to the research results.

The PPL Approach to Parallelism

A fundamental premise of the PPL is that parallel computing hardware will be heterogeneous. This is already true today: currently shipping personal computer systems are composed of a chip multiprocessor and a highly data parallel co-processor, the GPU. Large clusters of such nodes have already been deployed and most future high performance computing environments will contain GPUs. To fully leverage the computational capabilities of these systems, an application developer has to contend with multiple, sometimes incompatible, programming models. Shared memory multiprocessors are generally programmed using threads and locks (e.g. pthreads, OpenMP) while GPUs are programmed using data-parallel languages (e.g. CUDA, OpenCL) and communication between the nodes in a cluster is programmed with message passing (e.g. MPI). The existence of heterogeneous hardware systems is driven by the desire to improve hardware productivity, measured by performance per watt and performance per dollar. This desire will continue to drive even greater hardware heterogeneity that will include special purpose processing units. As the degree of hardware heterogeneity increases, developing software for these systems will become even more complex. Our hypothesis is that the only way to radically simplify the process of developing parallel applications and improve programmer productivity, measured by programmer effort required for a given level of performance, is to use very-high level domain-specific programming languages and programming environments. These environments will capture parallelism implicitly and will optimize and map this parallelism to heterogeneous hardware using domain-specific knowledge. Thus, our vision for the future of parallel application programming is to replace a disparate collection of programming models, which require specialized architecture knowledge, with domain-specific languages that match the knowledge and understanding of application developers.

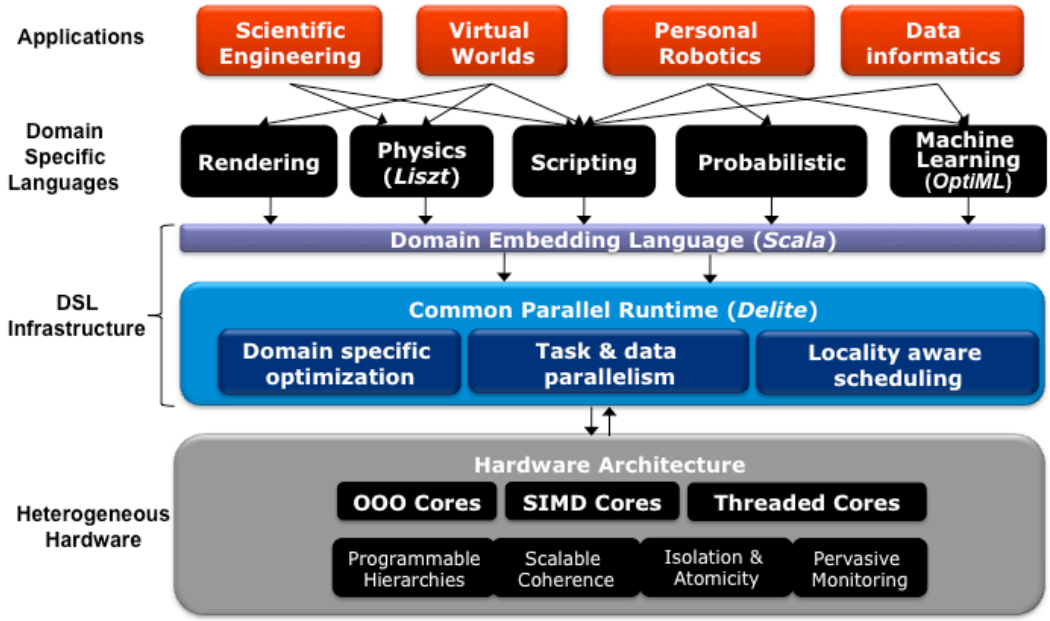


Figure 3: A layered approach to the problem of parallel computing.

The PPL Research Agenda

To drive PPL research we are developing new applications in areas that have the potential to exploit significant amounts of parallelism but also present significant software development challenges. These application areas demand enormous amounts of computing power to process large amounts of information, often in real-time. They include traditional scientific and engineering applications from geosciences, mechanical engineering and bio engineering; a massive virtual world including a client-side game engine and a scalable world server, personal robotics including autonomous driving vehicles and robots that can navigate home and office environments; and sophisticated data analysis applications capable of extracting information from huge amounts of data. These applications will be developed by domain experts in collaboration with PPL researchers.

The core of our research agenda is to allow a domain expert to develop parallel software without becoming an expert in parallel programming. Our approach is to use a layered system (Figure 3) based on implicitly parallel domain-specific languages (DSLs) [33], a domain embedding language, a common parallel runtime system, and a heterogeneous architecture that provides efficient mechanisms for communication, synchronization, and performance monitoring.

We expect that most programming of future parallel systems will be done in DSLs at the abstraction level of Matlab or SQL. DSLs enable the average programmer to be highly productive in writing parallel programs by isolating the programmer from the details of parallelism, synchronization, and locality. The use of DSLs also recognizes that most applications are not written from scratch, but rather built by combining existing systems and libraries [34]. The DSL environment uses its high-level view of the computation and its domain knowledge to direct placement and scheduling to optimize parallel execution. Our goal is to build the underlying technology that makes it easy to create implicitly parallel DSLs, and design and implement at least

three specific languages using this technology to demonstrate that it is capable of supporting multiple DSLs. To support science and engineering applications, we are developing a mesh based PDE DSL called *Liszt* and a physics DSL that is based on the physics simulation library *PhysBAM* which has been extensively used in biomechanics, virtual surgery and visual special effects. To support the many algorithms in robotics and data informatics that are based on machine learning, we are developing a machine learning DSL called *OptiML.* Finally, we are using the Scala programming language [35] to serve as the language used for embedding the DSLs. Scala integrates key features of object-oriented and functional languages and Scala's extensibility makes it possible to define new DSLs in a natural way.

Our common parallel runtime system (CPR), called Delite, supports both implicit task-level parallelism for generality and explicit data-level parallelism for efficiency. The CPR maps the parallelism extracted from a DSL based application to heterogeneous architectures and manages the allocation and scheduling of processing and memory resources. The mapping process begins with a task graph, which exposes task-level and data-level parallelism and retains the high-level domain knowledge expressed by the DSL. This domain knowledge is used to optimize the graph by reducing the total amount of work and by exposing more parallelism. The CPR scheduler uses this task graph to reason about large portions of the program and make allocation and scheduling decisions that reduce communication and improve locality of data access. Each node of the task graph can have multiple implementations that target different architectures, this provides support for heterogeneous parallelism.

To support the CPR, we are developing a set of architecture mechanisms that provide communication and synchronization with very low overheads. Such mechanisms will support both efficient fine-grained exploitation of parallelism to get many processors working together on a fixed-size dataset and coarse-grained parallelism to gain efficiency through bulk operations. A key challenge is the design of a memory hierarchy that simultaneously supports both implicit reactive mechanisms (caching with coherence and transactions) and explicit proactive mechanisms (explicit staging of data to local memory). Our goal is to develop a simple set of mechanisms that is general enough to support execution models ranging from speculative threads (to support legacy codes) to streaming (to support explicitly-scheduled data parallel DSLs) [36]. To evaluate these architecture mechanisms with full-size applications at hardware speeds we are using a prototyping system called the Flexible Architecture Research Machine (FARM). FARM tightly couples commodity processors chips for performance with FPGA chips for flexibility, using a cache coherent shared address space [37].

Liszt

Working with the developers of a hypersonic fluid flow simulation we have designed a DSL called *Liszt.* Liszt is a domain-specific programming environment developed in Scala for implementing PDE solvers on unstructured meshes. Here is a typical fragment of Liszt code:

```
val position = vertexProperty[double3]("pos")
val A = new SparseMatrix[Vertex,Vertex]

for (c <- mesh.cells) {
    val center = average position of c.vertices
    for (f <- c.faces) {
        val face_dx = average position of f.vertices - center
          for (e <- f.edges With c CCW) {
                val v0 = e.tail
                val v1 = e.head
                val v0_dx = position(v0) - center
                val v1_dx = position(v1) - center
                val face_normal = v0_dx cross v1_dx
                // calculate flux for face …
```

Liszt abstracts the representation of the common objects and operations used In flow simulation. First, since 3D vectors are common in physical simulation, they have been implemented as objects with common methods for dot and cross products. Second, the mesh data structure has been completely abstracted. All mesh access is performed through standard interfaces; in the code fragment, `mesh.cells` returns the set of cells in the mesh and "`f.edges With c CCW`" returns a list of edges around a face oriented counter clockwise. Field variables are associated with topological elements such as cells, faces, edges and vertices, but they are accessed through methods so that their representation is not exposed. Finally, sparse matrices (`A` in the example) are indexed by topological elements, not integers. This code appeals to the computational scientists because it is written in a form they understand; it also appeals to the computer scientists because it hides the details of the machine.

One of the keys to good parallel performance is good memory locality, and the choice of data structure representation and data decomposition can have an enormous impact on locality. The use of DSLs and domain-knowledge make it possible to pick the data structure that best fits the characteristics of the architecture. The DSL compiler for Liszt, for example, knows what a mesh, a cell, and a face are. As a result, the compiler has the information needed to select the data structure representations, data decomposition and the layout of field variables that are optimized for a specific architecture. Using this domain-specific approach, it is possible to generate a special version of the code for a given mesh running on a given architecture. No general-purpose compiler could possibly do this type of super-optimization.

Delite

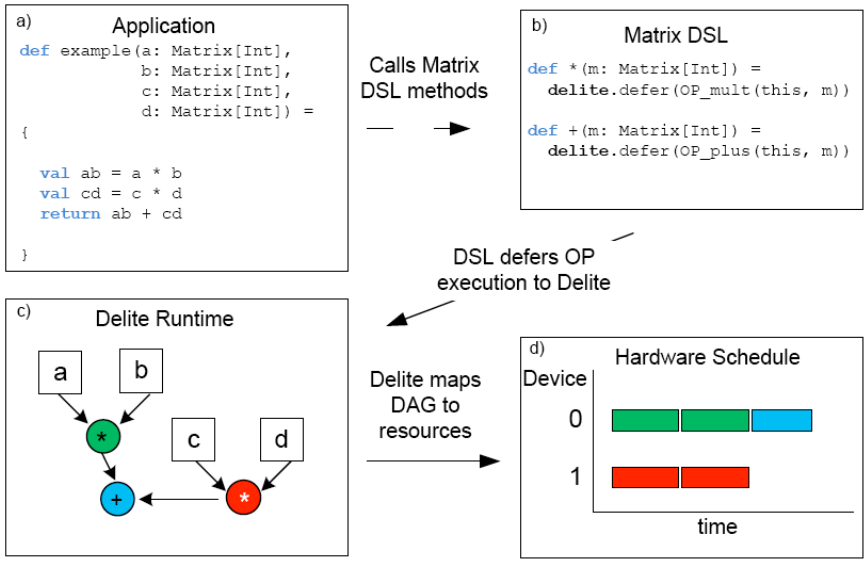


Figure 4: Delite example application execution overview.

Delite is both a framework that helps DSL developers write implicitly parallel DSLs as well as a run-time that handles most aspects of parallel heterogeneous execution. Delite enables a layered approach to parallel heterogeneous programming that hides the complexity of the underlying machine behind a collection of DSLs. The use of DSLs creates two types of programmers. The first type is the application developer who uses DSLs and is shielded from parallel or heterogeneous programming constructs (Figure 4a). The second type of programmer is the DSL developer. The DSL developer uses the Delite framework to implement a DSL. The DSL developer defines the mapping between DSL methods into domain specific units of execution called OPs. OPs are classes that hold the implementation of a particular domain specific operation (i.e. Matrix multiply) and other information needed by the run-time for efficient parallel execution (i.e. cost of this operation, is it side-effect free, etc…). OPs also encode any dependencies on data objects or previously created OPs. When an application calls a DSL method(the * method in Matrix), an OP (OP_mult) is submitted to the Delite run-time through a *defer* method and a proxy is immediately returned as the result of this DSL method invocation (Figure 4b). The application is oblivious to the fact that computation has been deferred and "runs-ahead" allowing more OPs to be submitted. These OPs form a dynamic task graph that can then be scheduled to run in parallel, if independent (Figure 4c). Multiple variants of each OP in the domain specific language can be generated to target different available parallel architectures (Figure 4d). As new architectures emerge, Delite can be enhanced to generate code for these new architectures. The application doesn't need to be rewritten to benefit from these new architectures, which provides forward scalability, and unless the DSL interface changes, the application need not even be recompiled.

Delite employs domain specific knowledge to optimize the task graph before it is scheduled for execution. These optimizations are enabled by domain knowledge supplied to Delite by the DSL developer. Furthermore, DSL OPs tend to be high-level operations (i.e. matrix multiplication, partitioning, mapping, filtering), and many of these OPs contain substantial data-parallelism. Data-

parallel OPs can be partitioned and scheduled independently. This allows Delite to take advantage of both the task-level parallelism by running independent OPs in parallel as well as data-parallelism by splitting data-parallel OPs into independent chunks. The Delite framework will provide a variety of OPs that allow the DSL developer to implement standard data-parallel operations without worrying about granularity and memory locality issues; the run-time will optimize granularity and locality dynamically based on the global information collected from all the OPs submitted to Delite. A DSL developer can also provide domain knowledge that describes how to decompose a data parallel operation for efficient execution.

We have demonstrated that Delite can be used to extract and optimize both task and data level parallelism for a number of machine learning (ML) kernels written in OptiML [38]. Our results show significant potential for this approach to uncover large amounts of parallelism from applications written using a Delite DSL. Future work will demonstrate how the Delite infrastructure can be used to target a heterogeneous parallel architecture composed of multicores and GPUs and interface with more specialized parallel runtimes that we are developing such as Sequoia (divide and conquer), Phoenix [39] (map/reduce) and GRAMPS [40] (producer/consumer).

Summary

The work at the three centers bears many similarities. All three centers are focused on applications and derive their work on programming models and architectures from the needs of applications (this is all the more noticeable given that the lead authors from each center are known for their computer architecture work). All three centers assume that future microprocessors will have hundreds of cores, and will combine different types of cores in one chip. All three centers work to support well a division of labor between efficiency experts and application domain experts. Berkeley expects the domain expert to use application frameworks or to program in a language such as Python, while the efficiency expert programs libraries in a language such as C++; performance is gained by selectively applying just-in-time specializations to the Python programs based on parallel motifs and existing libraries; Illinois expects domain experts to use safe versions of parallel C++ or C#; libraries developed by efficiency experts are incorporated without breaking safety by using suitable contractual interfaces. Stanford develops an infrastructure based on the extensible language Scala that enables efficiency experts to develop domain-specific languages to be used by domain experts; performance is gained by using domain-knowledge to drive compile-time and run-time optimization. Illinois and Stanford are concerned with better architectural support for communication and synchronization while Berkeley is focused on isolation and measurement; finally, Berkeley and Stanford are concerned with novel OS and run-time structures to support parallelism.

The stagnation in uniprocessor performance and the shift to multi-core architectures is a technological discontinuity that will force major changes in the IT industry. The three centers are proud of the opportunity given to them to have a major role in driving this change and humbled by the magnitude of the task.

Acknowledgments

UC Berkeley Par Lab research is sponsored by the Universal Parallel Computing Research Center, which is funded by Intel and Microsoft (Award # 20080469) and by matching funds from U.C. Discovery (Award #DIG07-10227). Additional support comes from six Par Lab Affiliate companies: National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

UPCRC-Illinois is funded by Intel and Microsoft and by matching funds from U. of Illinois.

Stanford PPL is funded by Sun Microsystems, Nvidia, IBM, Advanced Micro Devices, Intel, and NEC.

References

[1] Asanović, K., R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, K. Yelick., "A View of the Parallel Computing Landscape," *Communications of the ACM*, vol. 52, no. 10, 10/2009.

[2] Keutzer, K., T. Mattson, A Design Pattern Language for Engineering (Parallel) Software , *Intel Technical Journal*, Volume 13, issue 4, (to appear), 2010.

[3] Naik , M., C.-S. Park, K. Sen, and D. Gay, ``Effective Static Deadlock Detection,'' in *31st International Conference on Software Engineering*, Vancouver (ICSE 09), Canada, May 2009. ACM SIGSOFT.

[4] Geer, D., Will software developers ride Ruby on Rails to success? *IEEE Computer* 39(2), 18–20. Feb 2006.

[5] P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs... an experiment in software prototyping productivity. Technical Report YALEU/DCS/RR-1049, Yale University Department of Computer Science, New Haven, CT, 1994.

[6] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, Oct 2000.

[7] Catanzaro, B., S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "SEJITS: Getting Productivity and Performance with Selective Embedded JIT Specialization," *1st Workshop on Programmable Models for Emerging Architecture (PMEA)* at the 18th International Conference on Parallel Architectures and Compilation Techniques, Raleigh, North Carolina, 09/2009.

[8] Jones, C G., R. Liu, L. Meyerovich, K. Asanović, and R. Bodik, "Parallelizing the Web Browser," *1st USENIX Workshop on Hot Topics in Parallelism (HotPar09)*, Berkeley, CA, 03/2009.

[9] Armhurst, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M. *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report No UCB/EECS-2009-28. February 10, 2009.

[10] Gu, C., J. Lim, P. Arbelaez, and J. Malik, "Recognition using Regions," *Computer Vision and Pattern Recognition (CVPR09)*, Miami, Florida, 2009.

[11] Catanzaro, B., B. Su, N. Sundaram, Y. Lee, M. Murphy, and K. Keutzer, "Efficient, High-Quality Image Contour Detection," *IEEE International Conference on Computer Vision* (ICCV09), Kyoto Japan, 2009.

[12] Bird, S., A. Ganapathi, K. Datta, K. Fuerlinger, S. Kamil, R. Nishtala, D. Skinner, A. Waterman, S. Williams, K. Asanovic, D. Patterson, "Software Knows Best: Portable Parallelism Requires Standardized Measurements of Transparent Hardware," submitted for publication.

[13] Ganapathi, A., K. Datta, A. Fox, and D. A. Patterson, "A Case for Machine Learning to Optimize Multicore Performance", *1st USENIX Workshop on Hot Topics in Parallelism (HotPar09*), Berkeley, CA, 03/2009.

[14] Cook, H., K. Asanovic, and D. A. Patterson, *Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments*, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-131, Sep. 2009.

[15] Liu, R., K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Tessellation: Space-Time Partitioning in a Manycore Client OS," *1st USENIX Workshop on Hot Topics in Parallelism (HotPar09*), Berkeley, CA, 03/2009.

[16] Pan, H., B. Hindman, and K. Asanović, "Lithe: Enabling Efficient Composition of Parallel Libraries", *1st USENIX Workshop on Hot Topics in Parallelism (HotPar09)*, Berkeley, CA, 03/2009.

[17] Wu, W., Rivas, R., Arefin, A., Shi, S., Sheppart, R. M., Bui, B. D., Nahrstedt, K., "MobileTI: a portable tele-immersive system," presented at the Proceedings of the seventeen ACM international conference on Multimedia, Beijing, China, 2009.

[18] Pantic, M., Pentland, A., Nijholt, A., Huang, T. S., "Human computing and machine understanding of human behavior: A survey," *Lecture Notes in Computer Science,* vol. 4451, p. 47, 2007.

[19] Lin, D., Huang, X., Nguyen, Q., Blackurn, J., Rodrigues, C., Huang, T., Do, M. N., Patel, S. J., Hwu, W.-M. W., "The parallelization of video processing". *Signal Processing Magazine 26(6),* 103-112 (2009).

[20] C. Grier, S. Tang, and S.T. King. *Secure web browsing with the OP web browser.* Proceedings of the 2008 IEEE Symposium on Security and Privacy (Oakland). May 2008

[21] D. Dig, M. Tarce, C. R., M. Minea, R. Johnson. *ReLooper: Refactoring for Loop Parallelism in Java.* To appear in Companion Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09), Orlando FL, Oct 2009.

[22] A. Farzan, P. Madhusudan, F. Sorrentino. *Meta-analysis for Atomicity Violations under Nested Locking.* International Conference on Computer Aided Verification (CAV), Grenoble, France. Springer, 2009, pp. 248-262.

[23] R. Bocchino, V. Adve, S. Adve, and M. Snir*. Parallel Programming Must Be Deterministic by Default. First USENIX Workshop on Hot Topics in Parallelism (HotPar),* 2009.

[24] R. Bocchino, V. Adve, D. Dig., S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. *A Type and Effect System for Deterministic Parallel Java*. OOPSLA 2009.

[25] M. Vakilian, D. Dig. R. Bocchino, J. Overbey, V. Adve, and R. Johnson. *Inferring Method Effect Summaries for Nested Heap Regions*. ASE 2009.

[26] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almási, B.B. Fraguela, M.J. Garzarán, D.A. Padua, C. von Praun. *Programming for parallelism and locality with hierarchically tiled arrays*. PPOPP, 2006, pp. 48-57.

[27] R. Bocchino Jr. and V. Adve. Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks. In preparation.

[28], J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. *The Bulk Multicore for Improved Programmability.* Communications of the ACM (CACM), December 2009.

[29] Sarita Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. Communications of the ACM. To appear.

[30] Kelm, J., Johnson, D., Johnson, M., Crago, N., Tuohy, W., Mahesri, A., Lumetta, S., Frank, M., Patel, S. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator, 36th International Symposium on Computer Architecture, June 2009.

[31] P. Montesinos, M. Hicks, S.T. King, and J. Torrellas. *Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay*. 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2009.

[32] A. Nistor, D. Marinov, and J. Torrellas. *Light64: Lightweight Hardware Support for Race Detection during Systematic Testing of Parallel Programs.* International Symposium on Microarchitecture (MICRO), December 2009.

[33] Mernik, M., Heering, J., Sloane, M., When and How to Develop Domain-Specific Languages, *ACM Comput. Surveys*, 37(4):316–344, 2005.

[34] N. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. In PPoPP '10: Proceedings of the Fifteenth Annual Symposium on Principles and Practice of Parallel Programming, 2010.

[35] Odersky, M., Spoon, L., and Venners, B., Programming in Scala: A Comprehensive Step-by-Step Guide, Artima, 2008.

[36] D. Sanchez, R. Yoo and C. Kozyrakis, Flexible Architectural Support for Fine-Grain Scheduling, *ASPLOS '10.*

[37] J. Casper, T. Oguntebi, S. Hong, N. Bronson, C. Kozyrakis and K. Olukotun, "Hardware Acceleration of Transactional Memory on Commodity Systems," *in preparation.*

[38] Chafi, H., Sujeeth, A., Bronson, N., Makarov, D. and Olukotun, K., Delite: A Domain Specific Approach to Pervasive Parallelism, *in preparation.*

[39] R. Yoo, A. Romano, and C. Kozyrakis Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System, Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, October, 2009.

[40] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley and P. Hanrahan, GRAMPS: A Programming Model for Graphics Pipelines, Siggraph 2009.