UBIQUITOUS PARALLEL COMPUTING FROM BERKELEY, ILLINOIS, AND STANFORD

THE PARLAB AT BERKELEY, UPCRC-ILLINOIS, AND THE PERVASIVE PARALLEL LABORATORY AT STANFORD ARE STUDYING HOW TO MAKE PARALLEL PROGRAMMING SUCCEED GIVEN INDUSTRY'S RECENT SHIFT TO MULTICORE COMPUTING. ALL THREE CENTERS ASSUME THAT FUTURE MICROPROCESSORS WILL HAVE HUNDREDS OF CORES AND ARE WORKING ON APPLICATIONS, PROGRAMMING ENVIRONMENTS, AND ARCHITECTURES THAT WILL MEET THIS CHALLENGE. THIS ARTICLE BRIEFLY SURVEYS THE SIMILARITIES AND DIFFERENCE IN THEIR RESEARCH.

Bryan Catanzaro
Armando Fox
Kurt Keutzer
David Patterson
Bor-Yiing Su
University of California, Berkeley

Marc Snir
University of Illinois at Urbana-Champaign

Kunle Olukotun
Pat Hanrahan
Hassan Chafi
Stanford University

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

This evolution stopped a few years ago, however. Power constraints are resulting in stagnant clock rates, and new microarchitecture designs yield limited IPC improvements. 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 to scale to an increasing number of cores.

This is less of a problem for server applications, which can leverage parallelism by serving a larger number of clients or by consolidating server functions onto fewer systems. Parallelism for client and mobile applications is harder because they are turnaround oriented, and the use of parallelism to reduce response time requires more algorithmic work. Furthermore, mobile systems are power constrained, and improved wireless connectivity enables shifting computations to the server (or the cloud).

Given these conditions, can we identify applications that will run on clients and will require significant added compute power? Can we develop a programming environment that allows a large programmer community to develop parallel codes for such applications? In addition, can multicore architectures and their software scale to the hundreds of cores that hardware will be able to support in a decade? We believe we can answer such questions positively, but a timely solution will require a significant acceleration of the transfer of research ideas into practice.

Research on such topics is ongoing at the Parallel Computing Laboratory (ParLab) at the University of California, Berkeley, the Universal Parallel Computing Research Center at the University of Illinois, Urbana-Champaign (UPCRC-Illinois), and Stanford University's Pervasive Parallelism Laboratory (PPL). All three centers are parallelizing specific applications, 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 also reject a single-solution approach, assuming instead that software is developed by teams of programmers with different specialties requiring different sets of tools. Even though the work at the three centers bears many similarities, they focus on different programming environments, approaches for the production of parallel code, and architectural supports.

ParLab at Berkeley

Recognizing the difficulty of the multicore challenge, Intel and Microsoft invited 25 universities to propose parallel computing centers. They selected the Berkeley ParLab in 2008. Since then, another six companies have joined Intel and Microsoft as affiliates: National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. Embracing open source software and welcoming all collaborators, the ParLab is a team of 50 PhD students and a dozen faculty leaders from many fields who work together toward 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 to enable the productive development of efficient parallel programs by tomorrow's programmers. We believe future programmers will be either domain experts or sophisticated computer scientists because 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 software architecture is the key to designing parallel programs, and the key to these programs' efficient implementation is frameworks. In our approach, the basis of both is design patterns and a pattern language. Borrowed from civil architecture, *design pattern* means solutions to recurring design problems that domain experts learn. A *pattern language* is an organized way of navigating through a collection of design patterns to produce a design. Our pattern language consists of a series of computational patterns drawn largely from 13 motifs.[1,2] We see these as the fundamental software building blocks that are composed using a series of structural patterns drawn from common software architectural styles.

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 the pipe-and-filter style, 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 parallel programming's many annoying problems, such as nondeterminism, races, deadlock, starvation, and so on. To reduce such problems inside these frameworks, we use dynamic testing to execute problematic schedules.

SEJITS

Framework writers are more productive when they write in high-level productivity-layer languages (PLLs) such as Python or Ruby with abstractions that match the application domain; studies have reported factors of three to 10 fewer lines of code and three to five times faster development when using PLLs rather than efficiency-level languages (ELLs) such as C++ or Java. However, PLL performance might be orders of magnitude worse than ELL code, in part due to interpreted execution.

We're developing a new approach to bridge the gap between PLLs and ELLs. Modern scripting languages such as Python and Ruby include facilities to allow late binding of an ELL module to execute a PLL function. In particular, introspection lets a function 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 might 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. This approach preserves portability because it doesn't modify the source PLL program.

Although 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 generate code dynamically for the entire PLL. Also, the introspection and metaprogramming facilities let us embed the specialization machinery in the PLL directly rather than having to modify the PLL interpreter. Hence, we call our approach selective, embedded, just-in-time specialization (SEJITS).[3]

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 lets them concentrate on their respective specialties.

An example is Copperhead, which is a data-parallel Python dialect. 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. Preliminary experiments show encouraging efficiency, while enabling parallel programming at a much higher level of abstraction. Our goal is to specialize an entire image processing computation, such as in Figure 1.

Platforms and applications

We believe future applications will have footholds in both mobile clients and cloud computing. We're investigating parallel browsers because many client applications are downloaded and run inside a browser.[4]

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 various platforms. The application cloud challenge is responsiveness and throughput while minimizing cost in a pay-as-you-go environment.[5] 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 it has identified this person, the client device 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.

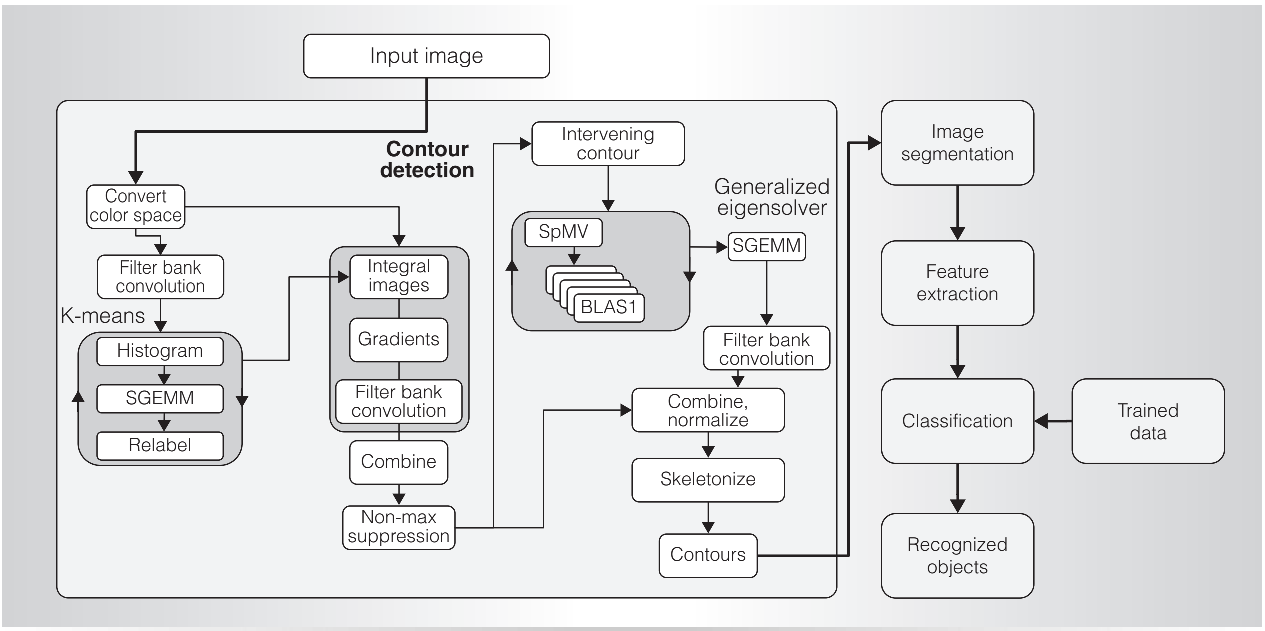


Figure 1. Computation flow of object recognition using regions. The major computational bottlenecks are the localcue computation and the generalized eigensolver.

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,[6] 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 0.15-Mbyte image, which is far too slow for our application.

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. Many algorithms have been proposed for these computations, each with different trade-offs. Each method corresponds to an application pattern. The application framework integrates these patterns. 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 object recognizer's four main computations. We can try out application patterns for different contour detectors, image segmentors, feature extractors, and trainer/classifier and find the most accurate composition. An alternative is to only specify the computation's composition and let the application framework choose the proper application pattern to realize the computation.

We choose application patterns manually. The contour detector, for example, uses 75 percent of the time. Among all contour detectors, our version of the gPb algorithm achieves the highest accuracy (see Figure 1). The major computational bottlenecks are the localcue computation and the generalized eigensolver. For the localcue computation, we replaced the explicit histogram accumulation by integral image and applied 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 architecture, we accelerated contour execution time by 140 times, from 4.2 minutes to 1.8 seconds on a GPU.[7]

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).[8] SHOT enables parallel programming environments to deliver 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.

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.

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

Future architecture and operating system

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 scratchpads.[9] We believe that such mechanisms for mobile clients will also aid servers in the cloud.

Because 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 Tessellation operating system implements scheduling and resource management of partitions.[10] Applications and operating system services (such as file systems) run within partitions. Partitions are lightweight and can be resized or suspended with overhead comparable to a context swap. The operating system kernel is a thin layer responsible for only the coarse-grained scheduling and assignment of resources to partitions and secure restricted communications among partitions. It avoids the performance issues with traditional microkernels by providing operating system services through secure messaging to spatially coresident 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.[11]

ParLab 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 field-programmable gate arrays that we use for architectural experiments, including the first SHOT implementation. FPGAs afforded a 250 times increase in simulation time over software simulators, and the lengthier experiments often led to opposite conclusions.[12]

The initial Tessellation operating system boots on our prototype and can create a partition, and Lithe has demonstrated efficient composition of code written using OpenMP libraries.

UPCRC-Illinois

The Universal Parallel Computing Research Center at the University of Illinois, Urbana-Champaign was established in 2008 as a result of the same competition that led to the ParLab. UPCRC-Illinois involves about 50 faculty and students and is codirected by Wen-mei Hwu and Marc Snir. It is one of several major projects in parallel computing at Illinois (http://parallel.illinois.edu), continuing a tradition that started with the Illiac projects.

Our work on applications focuses on the creation of compelling 3D Web applications and human-centered interfaces. We have a strong focus on programming language, compiler, and runtime technologies aimed at supporting parallel programming models that provide simple, sequential-by-default semantics with parallel performance models and that avoid concurrency bugs. Our architecture work is focused on efficiently supporting shared memory with many hundreds of cores. (The full list of UPCRC-Illinois ongoing projects is available at http://www.upcrc.illinois.edu.)

Applications

The 3D Internet will enable many new compelling applications. For example, the Teeve 3D teleimmersion framework and its descendants have supported remote collaborative dancing, remote Tai-Chi training, manipulation of virtual archeological artifacts, training of wheelchair-bound basketball players, and gaming.[13] 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 orders of magnitude performance improvements in tasks such as the synthesis of 3D models from multiple 2D images; the recognition of faces, facial expressions, objects, and gestures; and the rendering of dynamically created synthetic environments. Such tasks must execute on mobile clients 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, but this will require significant technical progress.

We're 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 74 times. We've implemented parallel versions of algorithms for analyzing video streams, demonstrating speedups in excess of 400 times 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. The complete application achieves speedups in excess of 13 times. We collected the algorithms into the publicly available Vivid library (http://libvivid.sourceforge.net) in the form of parallel functions callable from Python code.[14]

Spatial data structures form the backbone of many computationally intensive 3D applications and data-mining and machine-learning algorithms. To support such applications, we're developing ParKD, a comprehensive framework for parallelized spatial queries and updates through scalable, parallel k-D tree implementations. ParKD 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, to achieve high-quality, real-time graphics in interactive games, we need 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 enabling applications that provide better

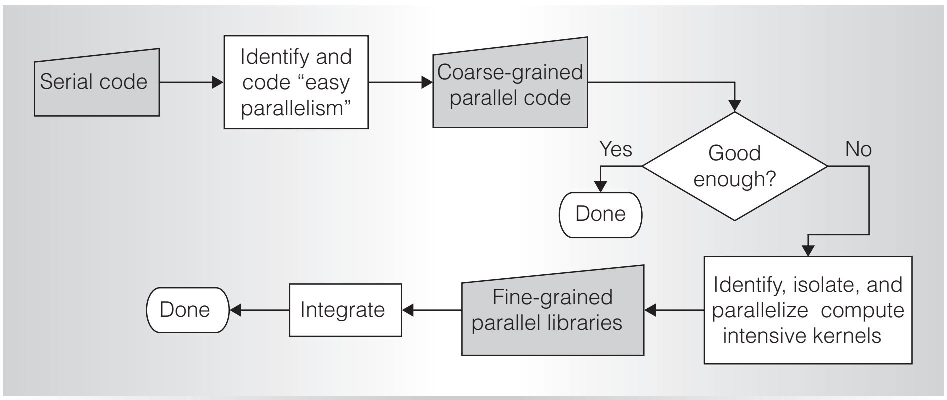

Serial code
Identify and code "easy parallelism"
Coarse-grained parallel code
Good enough?
Yes
No
Done
Identify, isolate, and parallelize compute intensive kernels
Fine-grained parallel libraries
Integrate
Done

Figure 2. Parallel code development workflow. Coarse-grained parallelism is often sufficient and easy to achieve; fine-grained parallelism is the work of experts.

user experience and that can be developed faster.

Parallelism can also help achieve other goals, such as better quality of service and 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 Opus Palladianum (OP) Web browser shows that parallelism both improves browser performance and reduces vulnerabilities by using compartmentalization.[15]

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 a computation's performance. Reactive systems (such as an operating system, GUI, or online transaction-processing system) where computations (or transactions) are triggered by nondeterministic, possibly concurrent, requests or events use concurrent programming. 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 doesn't increase the need for concurrent programming or make it harder; it increases the need for parallel programming. We contend that parallel programming is much easier than concurrent programming; in particular, it is seldom necessary to use nondeterministic code.

Figure 2 presents a schematic view of the parallel software creation process. Developers often start with a sequential code, although starting from a high-level specification is preferred. It's often possible to identify outer-loop parallelism where we can encapsulate all or most of the sequential code logic into a parallel execution framework (pipeline, master-slave, and so on) with little or no change in the sequential code. If this doesn't achieve the desired performance, developers identify compute-intensive kernels, encapsulate them into libraries, and tune these libraries to leverage parallelism at a finer grain, including single-instruction, multiple-data (SIMD) parallelism. Whereas the production of carefully tuned parallel libraries will involve performance coding experts, simple, coarse-level parallelism should be accessible to all.

We propose helping "easy parallelism" by developing refactoring tools that let us convert sequential code into parallel code written using existing parallel frameworks in C# or Java[16] as well as debugging tools that help identify concurrency bugs.[17] More fundamentally, we propose supporting simple parallelism with languages that are deterministic by default and concurrency safe.

Languages such as Java and C# provide safety guarantees (such as type 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 are 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)[18] shows we can satisfy both properties, even for modern object-oriented programming, without undue loss of expressiveness and with good performance. Moreover, we can enforce the race-free guarantee with simple, compile-time type checking. 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.[19] Our tool, DPJizer, can infer much of the added information DPJ requires. 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 fine-tune code and facilitates porting to new platforms. We believe that such work should happen in an environment where compiler analysis and code development interact closely so that the performance programmer works with the compiler, not against it, as is often the case today. We're 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 a portable common parallel code base for both fine-grained engines such as GPUs and coarser grained multicore CPUs, as well as their associated memory hierarchies. Autotuning by manipulating the algorithm or code is another technique we apply to this problem.

Parallel libraries are most easily integrated into sequential code when they follow the SIMD model. Our work on hierarchical tiled arrays (HTAs) confirms earlier results showing that this model can also be used to support a range of complex applications with good performance.[20] The use of tiles as a first-class object gives programmers good control of locality, granularity, and load balancing.

Integrating 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) doesn't violate assumptions made within the framework implementation. Such an approach gives an expert framework implementer freedom to use low-level and/or highly tuned techniques within the framework while enforcing a safety net for 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 lightweight cores, within a decade. Current cache coherence protocols don't scale to such numbers. With current protocols, each shared memory access by a core is considered to be a potential communication or synchronization with any other core. In fact, parallel programs communicate and synchronize in stylized ways. A key to shared memory scaling is adjusting coherence protocols to leverage the prevalent structure of shared memory codes for performance.

We're 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.[21]

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 codesigns the hardware with concurrency-safe programming models, resulting in a much simplified and scalable coherence protocol.[22] The Rigel architecture plans to shift more coherence activities to software.[23] In addition, a higher-level view of communication and synchronization across threads enables the architecture to help program development, for example, by supporting deterministic replay of parallel programs or by tracking races in codes that don't prevent them by design.[24]

Stanford University PPL

In May 2008, Stanford University officially launched the Pervasive Parallelism Laboratory. PPL's goal 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 an open industrial affiliates program. The lab's open nature lets other companies join the effort and doesn't provide any member company with exclusive intellectual property rights to the research results.

PPL approach

A fundamental premise of the PPL is that parallel computing hardware will be heterogeneous. This is already true today; personal computer systems currently shipping consist of a chip multiprocessor and a highly data-parallel GPU coprocessor. 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 must contend with multiple, sometimes incompatible, programming models. Shared memory multiprocessors are generally programmed using threads and locks (such as pthreads and OpenMP), while GPUs are programmed using data-parallel languages (such as CUDA and OpenCL), and communication between the nodes in a cluster is programmed with a message passing library (such as MPI). Heterogeneous hardware systems are driven by the desire to improve hardware productivity, measured by performance per watt and 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 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 application developer knowledge and understanding.

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 bioengineering; 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 that can extract information from huge amounts of data.

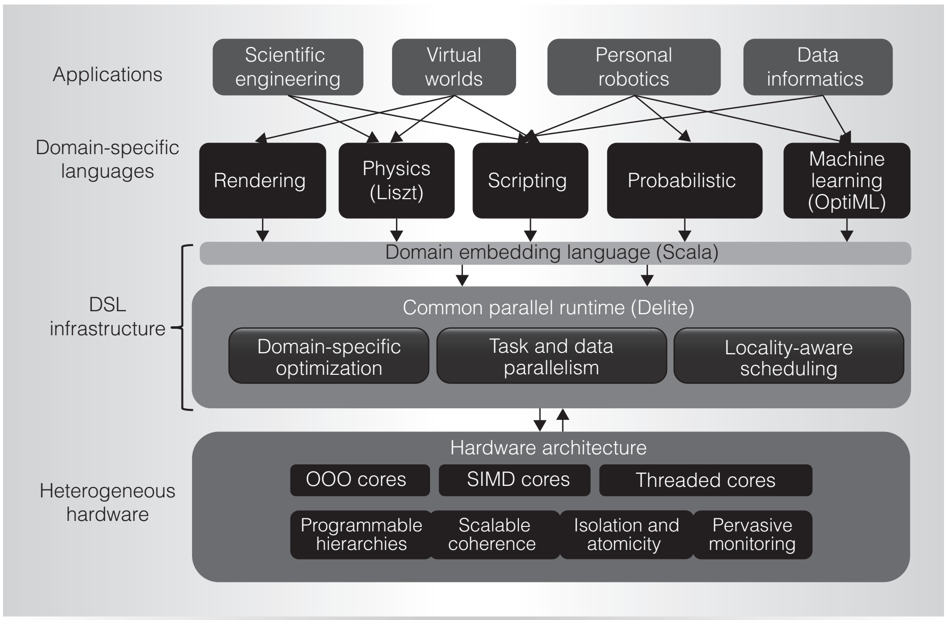


Figure 3. A layered approach to the problem of parallel computing The use of a common embedding language allows the interoperability between DSLs and the DSL infrastructure allows new DSLs to be developed more easily.

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), 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 aren't written from scratch, but rather built by combining existing systems and libraries.[25] 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 can support multiple DSLs. To support science and engineering applications, we're 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're developing a machine-learning DSL called OptiML. Finally, we're using the Scala programming language to serve as the language used for embedding the DSLs.[26] Scala integrates key features of object-oriented and functional languages, and Scala's extensibility makes it possible to define new DSLs naturally.

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- 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, which supports heterogeneous parallelism.

To support the CPR, we're developing a set of architecture mechanisms that provide communication and synchronization with low overhead. Such mechanisms will support both efficient fine-grained exploitation of parallelism to get many processors working together on a fixed-size data set 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).[27] To evaluate these architecture mechanisms with full-size applications at hardware speeds, we're using a prototyping system called the Flexible Architecture Research Machine. FARM tightly couples commodity processors chips for performance with FPGA chips for flexibility, using a cache-coherent shared address space.

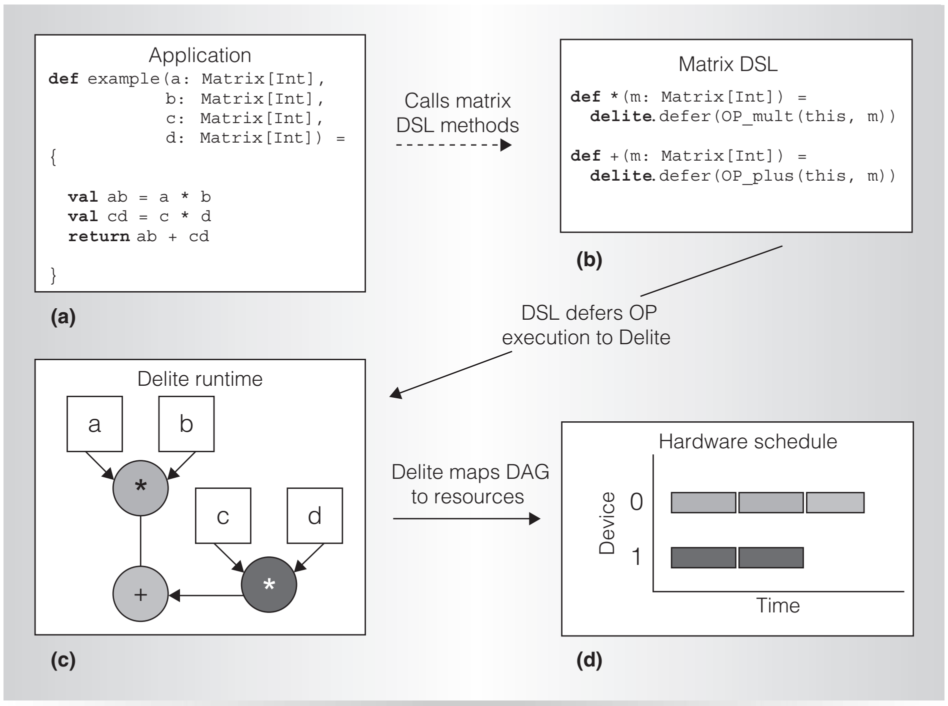
Liszt

Liszt is a domain-specific programming environment developed in Scala for implementing PDE solvers on unstructured meshes for hypersonic fluid simulation. It abstracts the representation of the common objects and operations used in flow simulation. Because 3D vectors are common in physical simulation, Liszt implements them as objects with common methods for dot and cross products. In addition, Liszt completely abstracts the mesh data structure, performing all mesh access through standard interfaces. Field variables are associated with topological elements such as cells, faces, edges, and vertices, but they are accessed through methods so their representation is not exposed. Finally, sparse matrices are indexed by topological elements, not integers. This code appeals to computational scientists because it is written in a form they understand. It also appeals to computer scientists because it hides the details of the machine.

One key 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

Delite's layered approach to parallel heterogeneous programming hides the complexity of the underlying machine behind a collection of DSLs. The use of DSLs creates two types of programmers: application developers, who use DSLs and are shielded from parallel or heterogeneous programming constructs (Figure 4a); and DSL developers, who use the Delite framework to implement a DSL. The DSL developer defines the mapping of DSL methods into domain-specific units of execution called OPs. OPs hold the implementation of a particular domain-specific operation and other information



```
def example(a: Matrix[Int],
            b: Matrix[Int],
            c: Matrix[Int],
            d: Matrix[Int]) =
{

  val ab = a * b
  val cd = c * d
  return ab + cd

}
```

```
def *(m: Matrix[Int]) =
  delite.defer(OP_mult(this, m))

def +(m: Matrix[Int]) =
  delite.defer(OP_plus(this, m))
```

Figure 4. Delite example application execution overview. Delite simplifies parallel programming using implicitly parallel DSLs.

needed by the runtime for efficient parallel execution (operation cost, possible side effects, and so on). OPs also encode any dependencies on data objects or existing OPs.

When an application calls a DSL method, an OP is submitted to the Delite runtime through a `defer` method and receives a proxy in return (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, and unless the DSL interface changes, the application need not even be recompiled.

Delite can be used to extract and optimize both task- and data-level parallelism for several machine-learning kernels written in OptiML. 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[28] (divide and conquer) and GRAMPS[29] (producer/consumer).

The stagnation in uniprocessor performance and the shift to multicore 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. MICRO

Acknowledgments

UC Berkeley ParLab research is sponsored by the Universal Parallel Computing Research Center, which is funded by Intel and Microsoft (award no. 20080469) and by matching funds from UC Discovery (award no. DIG07-10227). Additional support comes from six ParLab affiliate companies: National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. UPCRC-Illinois is funded by Intel and Microsoft and by matching funds from the University of Illinois. Stanford PPL is funded by Sun Microsystems, NVIDIA, IBM, Advanced Micro Devices, Intel, and NEC.

References

1. K. Asanović et al., ''A View of the Parallel Computing Landscape,'' *Comm. ACM,* vol. 52, no. 10, Oct. 2009, pp. 56-67.
2. K. Keutzer and T. Mattson, ''A Design Pattern Language for Engineering (Parallel) Software,'' to appear in *Intel Technical J.,* vol. 13, no. 4, 2010.
3. B. Catanzaro et al., ''SEJITS: Getting Productivity and Performance with Selective Embedded JIT Specialization,'' *Proc. 1st Workshop Programmable Models for Emerging Architecture* (PMEA), EECS Dept., Univ. of California, Berkeley, tech. report UCB/EECS-2010-23, Mar. 2010.
4. C.G. Jones et al., ''Parallelizing the Web Browser,'' *Proc. 1st Usenix Workshop Hot Topics in Parallelism* (HotPar 09), Usenix Assoc., 2009, http://parlab.eecs.berkeley.edu/publication/220.
5. M. Armhurst et al., ''Above the Clouds: A Berkeley View of Cloud Computing,'' *Comm. ACM,* vol. 53, no. 4, Apr. 2010, pp. 50-58.
6. C. Gu et al., ''Recognition Using Regions,'' *Proc. Computer Vision and Pattern Recognition* (CVPR 09), IEEE CS Press, 2009, pp. 1030-1037.
7. B. Catanzaro et al., ''Efficient, High-Quality Image Contour Detection,'' *Proc. IEEE Int'l Conf. Computer Vision* (ICCV 09), 2009; http://www.cs.berkeley.edu/~catanzar/Damascene/iccv2009.pdf.
8. S. Bird et al., ''Software Knows Best: Portable Parallelism Requires Standardized Measurements of Transparent Hardware,'' EECS Dept., Univ. of California, Berkeley, tech. report, Mar. 2010.
9. H. Cook, K. Asanovic, and D.A. Patterson, *Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments,* EECS Dept., Univ. of California, Berkeley, tech. report UCB/EECS-2009-131, Sept. 2009.
10. R. Liu et al., ''Tessellation: Space-Time Partitioning in a Manycore Client OS,'' *Proc. 1st Usenix Workshop Hot Topics in Parallelism* (HotPar 09), Usenix Assoc., 2009; http://parlab.eecs.berkeley.edu/publication/221.
11. H. Pan, B. Hindman, and K. Asanović, ''Lithe: Enabling Efficient Composition of Parallel Libraries,'' *Proc. 1st Usenix Workshop Hot Topics in Parallelism* (HotPar 09), Usenix Assoc., 2009; http://parlab.eecs.berkeley.edu/publication/222.
12. Z. Tan et al., ''A Case for FAME: FPGA Architecture Model Execution,'' to be published in *Proc. Int'l Symp. Computer Architecture,* June 2010.
13. W. Wu et al., ''MobileTI: A Portable Tele-immersive System,'' *Proc. 17th ACM Int'l Conf. Multimedia,* ACM Press, 2009, pp. 877-880.
14. D. Lin et al., ''The Parallelization of Video Processing,'' *Signal Processing,* vol. 26, no. 6, 2009, pp. 103-112.
15. C. Grier, S. Tang, and S.T. King, ''Secure Web Browsing with the OP Web Browser,'' *Proc. 2008 IEEE Symp. Security and Privacy,* IEEE CS Press, 2008, pp. 402-416.
16. D. Dig et al., ''ReLooper: Refactoring for Loop Parallelism in Java,'' *Companion Proc. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 09), ACM Press, 2009, pp. 793-794.
17. A. Farzan, P. Madhusudan, and F. Sorrentino, ''Meta-analysis for Atomicity Violations under Nested Locking,'' *Proc. Int'l Conf. Computer Aided Verification* (CAV), Springer, 2009, pp. 248-262.
18. R. Bocchino et al., ''A Type and Effect System for Deterministic Parallel Java,'' *Proc. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), ACM Press, 2009, pp, 97-116.
19. M. Vakilian et al., ''Inferring Method Effect Summaries for Nested Heap Regions,'' *Proc. 24th IEEE/ACM Conf. on Automated Software Eng.* (ASE 2009), ACM Press, 2009, pp. 421-432.
20. G. Bikshandi et al., ''Programming for Parallelism and Locality with Hierarchically Tiled Arrays,'' *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP), ACM Press, 2006, pp. 48-57.

21. J. Torrellas et al., ''The Bulk Multicore for Improved Programmability,'' *Comm. ACM,* Dec. 2009, pp. 58-65.
22. S. Adve and H.-J. Boehm, ''Memory Models: A Case for Rethinking Parallel Languages and Hardware,'' to appear *Comm. ACM,* http://rsim.cs.uiuc.edu/Pubs/10-cacm-memory-models.pdf.
23. J. Kelm et al., ''Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator,'' *Proc. 36th Int'l Symp. Computer Architecture,* IEEE CS Press, 2009, pp. 140-151.
24. A. Nistor, D. Marinov, and J. Torrellas, ''Light64: Lightweight Hardware Support for Race Detection during Systematic Testing of Parallel Programs,'' *Proc. Int'l Symp. Microarchitecture* (MICRO), IEEE CS Press, 2009, pp. 541-552.
25. N. Bronson et al., ''A Practical Concurrent Binary Search Tree,'' *Proc. 15th Ann. Symp. Principles and Practice of Parallel Programming* (PPoPP), ACM Press, 2010, pp. 257-268.
26. M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide,* Artima Press, 2008.
27. D. Sanchez, R. Yoo, and C. Kozyrakis, ''Flexible Architectural Support for Fine-Grain Scheduling,'' *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS), ACM Press, 2010; http://csl.stanford.edu/~christos/publications/2010.adm.asplos.pdf.
28. K. Fatahalian et al., ''Sequoia: Programming the Memory Hierarchy,'' *Proc. 2006 ACM/IEEE Conf. on Supercomputing,* ACM Press, 2006; http://graphics.stanford.edu/papers/sequoia/sequoia_sc06.pdf.
29. J. Sugerman et al., ''GRAMPS: A Programming Model for Graphics Pipelines,'' *ACM Trans. Graphics,* vol. 28, no. 1, 2009, pp. 1-11.

Bryan Catanzaro is a PhD candidate in the in the electrical engineering and computer science department at the University of California, Berkeley. His research interests center on programming models for many-core computers, with an applications-driven emphasis. Catanzaro has an MS in electrical engineering from Brigham Young University. He is a member of IEEE.

Armando Fox is an adjunct associate professor in the electrical engineering and computer science at the University of California, Berkeley, and active in the Parallel Computing Laboratory and the Reliable Adaptive Distributed Systems laboratory (RAD Lab). His research interests include cloud computing, programming languages and frameworks, and applied machine learning. Fox has a PhD in computer science from the University of California, Berkeley. He is a senior member of the ACM.

Kurt Keutzer is a professor of electrical engineering and computer science at the University of California, Berkeley, and the principal investigator in the Parallel Computing Laboratory. His research interests include patterns and frameworks for efficient parallel programming. Keutzer has a PhD in computer science from Indiana University. He is a fellow of IEEE.

David Patterson is the director of the Parallel Computing Laboratory, director of the Reliable Adaptive Distributed Systems Laboratory, and Pardee Professor of Computer Science at the University of California, Berkeley. His research interests include hardware and software issues in many-core systems and cloud computing. Patterson has a PhD in computer science from the University of California, Los Angeles. He is a fellow of IEEE and the ACM.

Bor-Yiing Su is a PhD candidate in the electrical engineering and computer science at the University of California, Berkeley. His research interests focus on developing programming frameworks for computer vision applications on many-core platforms. Su has a BS in electrical engineering from National Taiwan University.

Marc Snir is the Richard Feiman and Saburo Muroga Professor of Computer Science at the University of Illinois, Urbana-Champaign, codirector of UPCRC-Illinois, and chief software architect for the Petascale Blue Waters system. His research interests include high-performance computing, parallel programming models and programming environments, and computing

education. Snir has a PhD in mathematics from the Hebrew University of Jerusalem. He is a fellow of IEEE, the ACM, and the AAAS.

Kunle Olukotun is a professor of electrical engineering and computer science at Stanford University and director of the Stanford Pervasive Parallelism Lab. His research interests include computer architecture and parallel programming environments. Olukotun has a PhD in computer engineering from the University of Michigan. He is a fellow of IEEE and the ACM.

Pat Hanrahan is the Canon Professor of Computer Science and Electrical Engineering at Stanford University. His research interests include programming environments for parallel computing, graphics systems and architectures, and visualization. Hanrahan has a PhD in biophysics from the University of Wisconsin.

Hassan Chafi is a PhD candidate in electrical engineering at Stanford University. His research interests include transactional memory and parallel programming models. Hassan has an MS in electrical engineering from Stanford University.

Direct questions and comments to Marc Snir, Univ. of Illinois, Siebel Center for Computer Science, 201 N. Goodwin Ave., Urbana, IL, 60801; snir@illinois.edu.