The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View

Krste Asanovic
Ras Bodik
James Demmel
Tony Keaveny
Kurt Keutzer
John D. Kubiatowicz
Edward A. Lee
Nelson Morgan
George Necula
David A. Patterson
Koushik Sen
John Wawrzynek
David Wessel
Katherine A. Yelick

Electrical Engineering and Computer Sciences
University of California at Berkeley

March 21, 2008

The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View

Krste Asanovic, Rastilav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Edward Lee, Nelson Morgan, George Necula, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick

March 17, 2008

Abstract

In December 2006, we published a broad survey of the issues for the whole field concerning the multicore/manycore sea change (see view.eecs.berkeley.edu). (Asanovic, Bodik et al. 2006) We view the ultimate goal as the ability to create efficient and correct software productively that scales smoothly as the number of cores per chip doubles biennially. This much shorter report covers the specific research agenda that a large group of us at Berkeley is going to follow.

This report is based on a proposal for creating a Universal Parallel Computing Research Center (UPCRC) that a technical committee from Intel and Microsoft unanimously selected as the top proposal in a competition with the top 25 computer science departments. The five-year, $10M, UPCRC forms the foundation for the U.C. Berkeley Parallel Computing Laboratory, or Par Lab, a multidisciplinary research project exploring the future of parallel processing (see parlab.eecs.berkeley.edu)

To take a fresh approach to the longstanding parallel computing problem, our research agenda will be driven by compelling applications developed by domain experts. Historically, past efforts to resolve these challenges have often been driven "bottom-up" from the hardware, with applications an afterthought. We will focus on exciting new applications that need much more computing horsepower to run well, rather than on legacy programs that already run well on today's computers. Our applications are in the areas of personal health, image retrieval, music, speech understanding, and web browsers.

The development of parallel software is the heart of our research agenda. The task will be divided into two layers: an efficiency layer that aims at low overhead for 10 percent of the best programmers, and a productivity layer for the rest of the programming community--including domain experts--that reuses the parallel software developed at the efficiency layer. Key to this approach is a layer of libraries and programming frameworks centered on the 13 computational bottlenecks ("motifs") that we identified in the original Berkeley View report. (Asanovic, Bodik et al. 2006) We will also create a Composition and Coordination Language to make it easier to compose these components. Finally, we will rely on autotuning to map the software efficiently to a particular parallel computer. Past attempts have often relied on a single programming abstraction and language for all programmers and on automatically parallelizing compilers.

The role of the operating system and the architecture in this project is to support software and applications in achieving the ultimate goal, rather than the conventional approach of fixing the environment in which parallel software must survive. Example innovations include very thin hypervisors, which allow user-level control of processor scheduling, and hardware support for partitioning and fast barrier synchronization.

We will prototype the hardware of the future using field-programmable gate arrays (FPGAs), which we believe are fast enough to be interesting to parallel software researchers, yet flexible enough to "tape out" new designs every day, while being cheap enough that university researchers can afford to construct systems containing hundreds of processors. This prototyping infrastructure is called RAMP (Research Accelerator for Multiple Processors), and is being developed by a consortium of universities and companies (see ramp.eecs.berkeley.edu).

1. Introduction to Our Specific Aims, Relevance, and Research Plan
Preserving conventional sequential programming models while increasing performance led to general-purpose microprocessors that were fast but inefficient. The microprocessor industry is now at a crossroads due to hitting the limit of the power that a single integrated circuit can dissipate. To continue the pattern of increasing performance, semiconductor companies have been forced to replace the single large power-inefficient processor with several smaller power-efficient processors operating in parallel.

At the same time, because of the need to amortize rapidly increasing design costs, the desire to minimize end-product design risk, and the value in providing flexible platforms, many segments of embedded computing are moving away from per-product ASIC designs to families of programmable platforms. On-chip multiprocessing appears to be the most promising approach to deliver competitive high performance on these programmable platforms.

As a result, both general-purpose computing and embedded computing are being served by on-chip multiprocessor systems. The term *multicore* was coined to describe a microprocessor with multiple processors or cores on a single chip. The new "Moore's Law" is to double the number of processors per chip with each new technology generation, with individual processors going no faster. As we said in a recent report (Asanovic, Bodik et al. 2006):

> *"This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional single processor architectures."*

This switch to multicore breaks the 50-year tradition of how to program computers. Since the dawn of computing, many have tried to replace a single large processor with many small processors, but with little success. For example, there has been a nearly 100% failure rate of parallel computing startups: Convex, Encore, MasPar, nCUBE, Kendall Square Research, Sequent, Inmos, and Thinking Machines all bet on parallelism and are no longer in the marketplace.

The problem of effectively using increasingly parallel chips is primarily of making software applications run efficiently, yet the software industry largely isn't ready to use parallelism. Quoting from an interview (O'Hanlon 2006) with Stanford President John Hennessy:

> *"...when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced. … I would be panicked if I were in industry. … you've got a very difficult situation."*
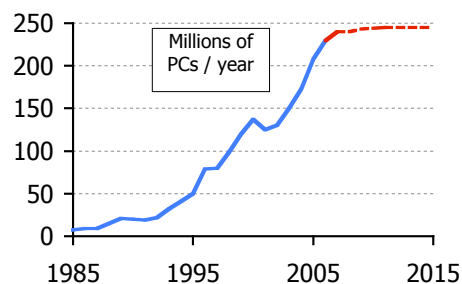

Figure 1. PC sales history (blue line) through 2006 and then hypothetical sales from 2007 (dashed red line) if PCs shift from a growth industry to replacement industry.

If software can't effectively use 32, 64, 128 ... cores per chip, then programs will run no faster on new computers, and so users may only buy a new computer when it wears out. Figure 1 shows the actual history of PC sales, and then projects sales if the IT industry degenerates from a growth industry to a replacement industry. Such a shift would be unprecedented and devastating to Silicon Valley. Hence, the current bet by the hardware industry may well fail.

Because of such concerns, on March 2, 2007 Intel and Microsoft announced it would fund a $2M per year Universal Parallel Computing Research Center (UPCRC). They invited the 25 top Computer Science departments to submit pre-proposals by April 6, 2007. On May 5, Intel and Microsoft invited Berkeley, Illinois, MIT, and Stanford to submit full proposals by June 8. The full proposal was submitted on June 8. Intel and Microsoft made site visits in July. On August 30, 2007, Intel and Microsoft announced that their technical committee had unanimously selected Berkeley as the top choice of the competition with Illinois ranked second.

2. Structure of the Universal Parallel Computing Research Center at Berkeley

It is comparatively easy to propose a project with a big group of faculty, and much harder to put together a plan that will actually generate strong collaborations among the participants. This is especially difficult with very large projects with many faculty as there is less accountability, so teams often only meet just before site visits to become organized. Berkeley, however, has a 20-year tradition of doing genuinely integrated system projects with many faculty tackling a common goal, with each faculty member being an expert in a discipline of interest to the research project. Often other faculty in related disciplines have some expertise, so many consult on a topic, but we defer to the appropriate experts.

We have formalized this model for the UPCRC by adopting the ACM structure for Special Interest Groups (SIGs). Figure 2 shows this organization and the faculty affiliations. The UPCRC SIGs are Applications (SIGAPP), Software Engineering (SIGSOFT), Programming Languages (SIGPLAN), Operating Systems (SIGOPS), and Architecture (SIGARCH). The "Chairs" of each SIG form the SIGBOARD, with the UPCRC Director acting as the Chair of SIGBOARD. SIGBOARD meets monthly over dinner and the other SIGs meet weekly. We plan to invite people from industry and nearby research labs to join our SIGs.

	David Patterson	James Demmel	Kurt Keutzer	Katherine Yelick	John Kubiatowicz	Krste Asanovic	Ras Bodik	Koushik Sen	Tony Keaveny	Nelson Morgan	David Wessel	George Necula	John Wawrzynek	Edward Lee
SIGBOARD	Chair	Member	Member	Member	Member									
SIGAPP		Chair	Member	Member			Member		Member	Member		Member	Member	
SIGSOFT			Chair			Member		Member			Member	Member		
SIGPLAN		Member		Chair							Member			Member
SIGOPS	Member				Chair				Member	Member	Member			
SIGARCH	Member			Member		Chair	Member						Member	
PI / Co-PI?	PI	PI	PI	PI	PI	PI	PI	PI	Co	Co	Co	Co	Co	Co

Legend: ■ SIG Chair   ■ SIG Member

Figure 2. UPCRC Organization and faculty leaders and members of SIGs.

3. Introduction to Research Themes

Two years before the Intel/Microsoft call for proposals, a multidisciplinary group of Berkeley researchers met weekly to discuss the implications of the parallel revolution for applications, programming models, architecture, and operating systems (Asanovic, Bodik et al. 2006). These "Berkeley View" discussions created a synergistic environment that led to the UPCRC. We decided on a fresh approach: to start top-down from applications; to innovate across disciplinary boundaries by creating a culture that encourages interaction and cooperation; and to create prototypes that can be quickly adapted to reflect multidisciplinary innovation.

To choose our applications, we first considered the two likely dominant future platforms:

- *The datacenter*. Google, Microsoft, and others are racing to construct buildings with 10,000 servers to run software as a service. We expect this trend to accelerate.
- *The mobile client (laptop/handheld).* In 2007, the largest maker of PCs shipped more laptops than desktops. Millions of cell phones are shipped each day and they are increasing in functionality. We expect these trends to accelerate as well.

In this re-imagining of client-server computing, the UPCRC is aimed at the mobile client, since it is the harder target. At home or in the office, people will use large displays with their mobile device. For economic and technical reasons, clients will not be continuously connected to the Internet, and so they must have substantial computation and storage while running on batteries.

When we explored our candidate large-scale client applications with our domain experts, we made several critical observations. First, the applications are based on frameworks that support components written in multiple languages, and use large libraries of reusable components drawn from multiple sources. Second, all of our domain experts require very efficient execution (i.e., optimized C or assembler) of at least some modules of their code even though many others are written in scripting languages. Using a sequential programming model, composition of such a wide variety of types of code is conceptually straightforward, if occasionally cumbersome in practice. Using today's parallel programming models, it is conceptually difficult and impossible in practice. *We believe composition lies at the heart of the parallel computing challenge.* Without sane composition, there can be no reuse. Without reuse, there is no software industry.

These observations also caused us to pause and reconsider our own pet language, compiler, and architecture projects. Solving individual concurrency problems with a clever language feature, compiler pass, or hardware widget will not lead to a pervasive parallel software industry (although it is a proven path to research conference publication). Rather than trying to push technology nuggets out to our users, we've instead taken an application-driven approach. We refocused our efforts on providing the software and hardware infrastructure necessary to build, compose, and understand large bodies of parallel software written in multiple languages, guided by the commonly recurring patterns actually observed in application codes.

In the rest of this section, we describe the four overarching research themes in our project: compelling applications, identifying computational bottlenecks, developing parallel software, and composable primitives. We describe the details of the individual research projects associated with these research themes in Section 4. Figure 3 gives a pictorial overview of the research themes and the corresponding research projects.

3.1. Applications

We selected applications to drive our research and provide concrete goals and metrics to evaluate progress, both in performance (can they run as quickly as desired, e.g. in real-time, or scaling with the number of cores?) and in productivity (can they be programmed easily by non-experts?). Each application was selected based on the following five criteria: **1)** Compelling in terms of likely market or social impact, with short-term feasibility and longer-term potential. **2)** Requires a significant speed-up or smaller, more efficient platform to work as intended. **3)** Taken together, the applications should cover the possible platforms (handheld, desktop, games) and markets (consumer, business, health) likely to dominate usage. **4)** Enables technology for other applications. **5)** A committed expert application partner will help design, use, and evaluate our technology. We will periodically reevaluate and adjust our application set to make sure we meet these criteria.
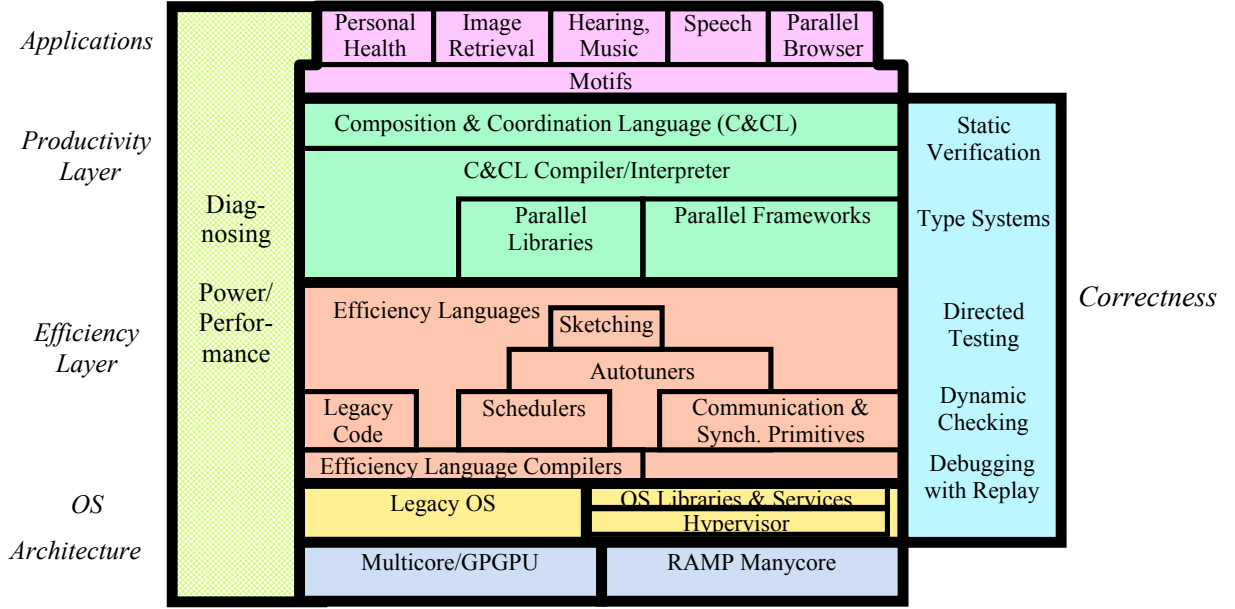
Figure 3. Overview of Parallel Computing Laboratory Research Agenda.

Our generic approach is as follows. We will decompose each application into the productivity-level components discussed in other sections; we have already completed initial decompositions. We will model (and eventually simulate and measure) performance of both the applications and HW/SW components to measure progress towards the required performance and productivity gains. We will also assess productivity gains by observing non-expert programmers. All this will provide feedback to the HW and SW researchers, and provide progress metrics. We discuss each application in turn, leaving details to the projects section in the appendix.

Personal Medicine, applied to Coronary Heart Disease (CHD). The seamless use of advanced physiological modeling of 3D medical images is a new healthcare paradigm we call "virtual stress testing" that will inform both doctor and patient about a patient's medical condition in a way that is scientifically accurate, intuitive, and compelling, and would help encourage healthy behavior and reduce death rate. We have already developed and are commercializing this application for osteoporosis, which we will use as a starting point (Keaveny, Donley et al. 2006).

Context-Based Image Retrieval (CBIR). The size of large media or consumer-generated image databases is growing so fast that their utility will depend on automated search, not manual labeling. Current image classifiers are performance limited, both because low error rates depend on processing very high dimensional feature spaces, and because users demand low response time. We are using Intel's PIRO code-base, and will apply it to consumer and industrial image databases.

Music and Hearing Augmentation. High-performance signal processing in the right form factor will permit a) much improved hearing aids (we are already working with a leading manufacturer, Starkey Labs); b) sound delivery systems for concert halls, conference calls, and home sound systems (using large microphone and speaker arrays, where Meyer Sound Labs is an industrial partner); c) musical information retrieval; d) composition and gesture-driven live-performance systems that exploit new sensor technologies (Wessel and Wright 2002).

Speech Recognition and Digital Diary. Dramatically improved automatic speech recognition performance in moderately noisy and reverberant environments would greatly improve existing applications and enable new ones like a real-time meeting transcriber with rewind and search. In recent NIST evaluations on recognizing speech from meetings, the best system (ours) still

generated word errors 40% of the time, and so there is much room for improvement (Morgan, Zhu et al. 2005). Progress depends on exploiting greatly expanded signal and feature transformations.

Parallel Web Browser. The browser will be the largest and most important application on many mobile devices. Future browser compute demands will grow with increasing connection speeds, richer web pages, and better output devices. We will parallelize browser serial bottlenecks, mainly parsing, layout/rendering, and script processing. Our goal is to make every website developer a parallel programmer, by introducing an implicitly parallel web scripting language supporting parallelized plug-ins.

The five application areas are all destined for mobile computing platforms such as cell phones and laptops. Each of them is computationally intensive and, with the current exception of the personal health application, thrives on rapid and predictable interaction with the user. The music and hearing augmentation apps are the most demanding, requiring very low and jitter-free latency. Our parallel browser must also provide a satisfying user experience with predictable reactiveness. The image processing application must also be responsive and will likely exploit a multitouch user interface implemented in the parallel browser. The digital diary app must also exhibit rapid fire interaction if it is to provide a quality user experience.

- At the first Par Lab retreat January 7 to 9, 2008, the consensus from both the Berkeley and industry attendees was that low-latency and temporal predictability are important to the success of future client applications. This perspective is a focus of our research.

- There is considerable synergy among the applications primarily because they share a number of motifs. Content-based image retrieval, music information retrieval, and speech recognition share a number of common features including a strong machine-learning component.

We certainly imagine expanding the application areas as the project evolves. Gaming is certainly an application area that is already attempting to exploit parallel architectures and programming models. Witness the Sony Play Station 3 with its IBM Cell Processor – a very difficulty processor for which to write software – as well as the aggressive use of manycore graphic processors in high-end desktop gaming.

3.2 Identifying Computational Bottlenecks

We will effectively expand our applications coverage using the concept of a computational *motif* common across a set of applications (Asanovic, Bodik et al. 2006). Note that in previous documents we used the term *dwarf*. We have now abandoned that term in favor of *motif.* Figure 4 is a temperature chart listing the motifs and shows their importance to each application. The Berkeley View team showed that 13 *motifs* cover six app areas (1st 6 columns of the figure), and while these 13 are not necessarily complete, they provide a set of essential computational patterns that systems must support. They actually play 4 roles: **1**) They define patterns for creating frameworks and libraries reusable across application domains (see Section 3.3); **2**) They aid multidisciplinary discourse, in that they are simple enough and few enough so that members of all SIGs understand them and we can rely on them for communication; **3**) Motifs are not tied to code or language artifacts--these "anti-benchmarks" encourage innovation in algorithms, languages, data structures, and hardware; **4**) They decouple research, allowing analysis of programming support without waiting years for full application development.

A natural question is what happens if we try to decompose the five apps above into the 13 motifs. We found that all five apps mapped neatly onto existing motifs, as illustrated by the next five columns in the table (for the browser, we illustrate the core code; plug-ins could look quite different). We believe this mapping success is further strong evidence as to the effectiveness of the motifs.

Motif	Embed	Desktop	Games	DB	ML	HPC	Medicine	Music	Speech	CBIR	Browser	Motif	Desktop	Games	DB	ML	HPC	Medicine	Music	Speech	CBIR	Browser
1 Finite State Mach.												9 N-Body										
2 Combinational												10 MapReduce										
3 Graph Traversal												11 Backtrack/B&B										
4 Structured Grid												12 Graphical Models										
5 Dense Matrix												13 Unstructured Grid										
6 Sparse Matrix												*Temperature Chart of Need*				DB = database						
7 Spectral (FFT)												*Hot*	*Warm*		*Med*	*Cool*	ML = machine learning					
8 Dynamic Prog																HPC = High Perf. Comp.						

Figure 4. Temperature Chart of the 13 Motifs. It shows their importance to each of the original six application areas and then how important each one is to the five compelling applications of Section 3.1. More details on the motifs can be found in (Asanovic, Bodik et al. 2006).

3.3 Developing Parallel Software with Productivity, Efficiency, and Correctness

Our key research objective is to enable programmers to easily write programs that run efficiently on manycore systems. We believe productivity, efficiency, and correctness are inextricably linked and must be addressed together. We do not believe that these three objectives can be accomplished with a single-point solution, such as one universal language. In our approach, efficiency will be principally handled by the use of an **efficiency layer** that is targeted for expert parallel programmers to develop optimized *libraries* and *programming frameworks* (Johnson 1997) tailored for parallel implementation. Productivity is addressed in a complementary **productivity laye**r that will use a common composition and coordination language to glue libraries and parallel programming frameworks into applications and reusable *application frameworks*. Both layers will come with a comprehensive approach to **correctness**.

3.3.1 Efficiency layer: Expert programmers work in the efficiency layer to develop highly tuned parallel *libraries* and parallel *programming frameworks* for use in the productivity layer. Our novel observation is that each motif can be expressed either as a library or as a framework.

Parallel Libraries: A *parallel library* is a set of procedures and data types with serial interfaces and hidden parallelism; such libraries have been successfully used in games, multimedia, and scientific computing. The following motifs are amenable to encapsulation in a library: dense and sparse matrix algorithms, spectral methods, graphical models, and combinatorial algorithms.

Parallel Programming Frameworks: In some algorithms, parallelism cannot be easily hidden behind a serial library interface because the parallel computation depends on the application. Consider a parallel graph traversal: the work performed on nodes differs from application to application. We believe that the remainder of the motifs has this nature. To support them, we will use *programming frameworks*, which we envision as parallel patterns parameterizable with "plug-in" components. The component inserted into the framework must obey certain restrictions: they will have to have a serial interface and in some cases, they will have to be stateless. In the case of parallel graph traversal, the plug-in may be a simple function computing weights, but it could also be a parallel library or (recursively) another framework. The benefit of programming frameworks is reuse of parallelism control patterns. Whereas libraries support function and data abstraction, programming frameworks (under our definition) support parallel control abstraction.

Our use of programming frameworks is based on our understanding of parallelism in our applications, traditional software engineering architectural styles (Shaw and Garlan 1996), and

structural patterns that appear in Mattson's *Patterns for Parallel Programming (Mattson, Sanders et al. 2005)*.We give three examples here: **1**) MapReduce (Dean 2006) algorithms involve independent parallelism followed by global combining algorithms. Two variations include equal computational cost in the map operation, where a static assignment of work to cores is appropriate, and one in which it varies (e.g., Google's MapReduce), demanding a load balancer. Both involve global communication primitives such as reductions and transposes. **2**) Divide-and-conquer parallelism is used in backtrack search and can make use of a task queue that uses randomized load balancing (as in Cilk (Blumofe, Joerg et al. 1995)) or diffusion-based load balancing (if locality between tasks is important). **3**) For the most irregular computations with arbitrary task graphs having unknown structure, such as in factorization algorithms (Demmel, Gilbert et al. 1999; Lewis and Richards 2003; Buttari, Dongarra et al. 2007) and completion procedures (Yelick and Garland 1992; Chakrabarti and Yelick 1993), coarse-grain software dataflow scheduling has proven effective. This last case is very general, because it supports any kind of parallelism pattern, but identifying special cases like 1), 2) and others are important for productivity and efficiency.

Programming at the Efficiency Layer. We offer two novel and complementary techniques for program synthesis that shifts some of the burden of library and framework construction from human time to computer time: ***Sketching*** (Solar-Lezama and Rabbah 2005) allows programmers to specify the skeleton of an optimized algorithm and have the sketching system correctly and verifiably fill in missing pieces relative to an executable specification of the function. ***Autotuning*** (Demmel, Dongarra et al. 2005) uses performance feedback from models and experiments to automatically search a space of proposed implementations for a given function and select the one that is best for a given machine and problem class. Both of these are described in more detail in the Projects appendix.

Programs in the efficiency layer are written very close to the machine with the goal of allowing the best possible algorithm to be written in the primitives of this layer. Unfortunately, existing multicore and planned manycore systems do not offer a common low-level programming model for parallel code. Using the idea of a *software* "Research Accelerator for Multi-Processors", we will define a thin portability layer (*software RAMP*) that runs efficiently across single socket platforms and has features for parallel job creation, synchronization, memory allocation, and bulk memory access. To provide a common model of memory across machines with coherent caches, local stores, and relatively slow off-chip memory, we will define an API based on the idea of logically partitioned shared memory, inspired by our experience with Unified Parallel C (UPC 2005), which partitions memory between processors but currently not between on and off-chip (see Yelick bio). This efficiency language may be implemented either as a set of runtime primitives or as a language extension of C. It will be extensible with libraries to experiment with various architectural features, such as transactions, dynamic multithreading, active messages, and collective communication. This API will be implemented on some existing multicore and manycore platforms and our own emulated manycore design.

3.3.2 Productivity layer: We believe the key to generating a successful software developer community for manycore is to maximally leverage the efforts of parallel programming experts by encapsulating their software for use by the programming masses. As mentioned above, we plan to reuse expert work through parallel libraries and programming frameworks. Productivity layer programmers will compose these into applications with the help of a *composition and coordinatio*n language (Lucco and Sharp 1991; Gelernter and Carriero 1992) . The language will be implicitly parallel (i.e., the composition will have serial semantics), which means that the composed programs will be safe (e.g., race-free) and virtualized with respect to processor resources. The language will document and check interface restrictions to avoid concurrency bugs resulting from incorrect composition; for example, if instantiating a framework with a stateful function when a stateless one is required. Finally, the language will support definition of domain-

specific abstractions to support construction of *application frameworks*, offering a programming experience similar to that in MatLab (Karnofsky 1996), SQL (Date 1989), or Sawzall (Pike 2005). Our audio and browser applications both involve the construction and use of such domain-specific frameworks.

To illustrate the responsibilities of the composition language, consider a structured grid framework, which may require independence of its plug-ins. The programmer writes a stencil operation that reads a limited domain of one grid and writes to a single point in another grid. (The unstructured grid framework is similar, but the data structure is an arbitrary graph.) The side-effect properties of the stencil plug-in will be enforced by either (conceptually) copying input arguments, or by supplying the framework with partition functions that guarantee to produce non-overlapping results. The partition operations may be simple array slices, or a graph partitioner that is guaranteed to output non-overlapping partitions. Thus, a graph partitioner library would have associated semantic properties (independence) with its output; these will be written in the coordination language, although the library itself may be written in expert programmers' languages of choice.

The key problem in composing parallel code is correctness. To understand what restrictions we place on composition to make correctness easier to achieve, it helps to consider two kinds of codes: parallel (P) and serial (S). Their composition can be divided into four types: S$\rightarrow$S, S$\rightarrow$P, P$\rightarrow$S, and P$\rightarrow$P (here the "$\rightarrow$" refers to a calling relationship, i.e., S$\rightarrow$P means serial code calls parallel code). S$\rightarrow$S is well supported in existing languages, and S$\rightarrow$P is a common invocation of a parallel library, so we focus on P$\rightarrow$S and P$\rightarrow$P composition. We believe that P$\rightarrow$P is only for experts and is only allowed in the efficiency layer. P$\rightarrow$S corresponds to parameterizing a parallel framework with a component that has a serial interface.

The ability to compose parallel pieces also introduces potential performance problems, since it involves combinations of parallel schedulers (Cosnard, Jeannot et al. 2001; Gürsoy and Kale 2004; Cicotti and Baden 2006). For example, after composition, a scheduler designed to carefully map a fixed set of computations on an equal number of cores may be used within the computations in a task-stealing framework. To allow for scheduler cooperation, frameworks will be parameterized by their resource requirements so that they can adapt to the context in which a framework is running. These requirements may change across program phases. We will develop tools to custom-generate thread scheduling code from the coordination and composition language.

3.3.3 A Comprehensive Approach to Program Correctness: Our approach to correctness uses a combination of specification generation, modular verification (Henzinger, Jhala et al. 2003; Flanagan, Freund et al. 2005), directed automated unit testing (also known as concolic testing) (Godefroid, Klarlund et al. 2005; Sen, Marinov et al. 2005; Majumdar and Sen 2007) and runtime monitoring (Havelund and Ro 2001; Barringer, Goldberg et al. 2004). Correctness is addressed differently at the two layers: the productivity layer will be free from concurrency problems because the parallelism models are very restricted and those restrictions will be enforced; the efficiency layer code will be checked for subtle concurrency errors (Dinning and Schonberg 1991; Savage, Burrows et al. 1997; Flanagan and Freund 2001; Choi, Lee et al. 2002; Flanagan and Freund 2004; Narayanasamy, Wang et al. 2007).

A key challenge in verification is obtaining specifications for programs against which to verify. Modular verification and automated unit test generation require the specification of high-level serial semantic constraints on the behavior of the individual modules such as the parallel frameworks and the parallel libraries. We will use executable sequential programs having the same behavior as a parallel component, augmented with atomicity constraints on a task (Sen and Viswanathan 2006; Jhala and Majumdar 2007), predicate abstractions of the interface of a module (Henzinger, Jhala et al. 2005), or multiple ownership types (Clarke, Potter et al. 1998). Programmers often find it difficult to specify such high-level contracts of large modules; however, most programmers find it convenient to specify local properties of programs using

assert statements and type annotations. Often, local assertions and type annotations can also be generated from implicit correctness requirements of a program, such as data race and deadlock freedom and memory safety (Necula, Condit et al. 2005). Implications of these local assertions are propagated to the module boundaries by using a combination of static verification and directed automated unit testing. The propagated implications create serial contracts (Meyer 1992) that specify how the modules, such as frameworks, can be correctly used. Once the contracts for the parallel modules are in place, we will use static program verification to check if the client code composed with the contracts is correct.

Static program analysis in the presence of pointers and heap memory report many errors that are not possible. For restricted parallelism models with global synchronization, these problems become more tractable and a recent technique called ***directed automated testing*** or ***concolic*** unit testing (Sen, Marinov et al. 2005) has shown promise to improve software quality through automated test generation using a combination of static and dynamic analyses. We propose to combine directed testing with model checking algorithms to unit test parallel frameworks and libraries composed with serial contracts. Such techniques enable us to quickly test executions for data races and deadlocks directly, since a combination of directed test input generation and model checking hijacks the underlying scheduler and controls the synchronization primitives. Our techniques will also provide deterministic replay and debugging capabilities at very low cost. For scheduler hijacking, we will leverage hardware support. We propose to develop randomized extensions of our directed testing techniques to build a probabilistic model of path coverage. The probabilistic models will give a more realistic estimate of coverage of race and other concurrency errors in parallel programs.

A critical aspect of our verification work is to ensure the techniques provide real value to application developers. We will work with our application experts to determine the forms of verification they require, and the level of annotation they are comfortable providing.

3.4 OS and Architecture: Composable Primitives not Pre-Packaged Solutions
Our approach to both operating systems and hardware architecture is to deconstruct conventional functionality into primitive mechanisms that software can compose to meet application needs.

A traditional OS is designed to multiplex a large number of sequential jobs onto a small number of processors, with virtual machine monitors (VMMs) adding another layer of virtualization (Rosenblum and Garfinkel 2005). We instead propose a very thin hypervisor layer that exports spatial hardware partitions to application-level software. These "un-virtual" machines allow each parallel application to use custom processor schedulers without fighting fixed policies in OS/VMM stacks. The hypervisor supports hierarchical partitioning, with mechanisms to allow parent partitions to swap child partitions to memory, and partitions can communicate either through protected shared memory or messaging. Traditional OS functions are provided as unprivileged libraries or in separate partitions. For example, device drivers run in separate partitions for robustness, and to isolate parallel program performance from I/O service events. This approach is similar to the MIT Exokernel project (Engler, Kaashoek et al. 1995), but instead of having a single isolation barrier (user/supervisor), we support arbitrarily nested partitions. An important difference from earlier systems is that we also export physical processor scheduling and virtualization out of the fixed hypervisor layer to better support manycore machines. This deconstructed and distributed OS style improves both scalability and resiliency to hardware and software faults. Our hardware architecture enforces partitioning of not only of cores and on-chip/off-chip memory, but it also partitions the communication bandwidth between these components, with quality of service guarantees for system interconnect. The resulting performance predictability improves parallel program performance, simplifies code (auto)tuning and dynamic load balancing, and supports real-time applications.

Our partitioned architecture allows software to directly use new low-level hardware primitives to build customized parallel run-time systems, while protecting the larger system from

bugs in any partition. On-chip memory can be configured as either cache or scratchpad RAM, and user-programmable DMA engines move data efficiently through the memory hierarchy (Williams, Oliker et al. 2007). Atomic fetch-and-op instructions provide efficient memory-side synchronization, including across partitions. Barrier hardware provides rapid synchronization within a partition. As an example, the combination of scratchpad RAM, DMA, and fast barriers, provides a simple and highly efficient run-time for many motifs: structured grids, unstructured grids, dense matrices, sparse matrices, and spectral methods.

We will use the RAMP FPGA-emulation infrastructure (Wawrzynek, Patterson et al. 2007) to allow rapid co-design of the hardware and software features. RAMP makes it possible to run whole applications at reasonable speed, allowing full-system evaluation of our proposed system stack.

4. Summary: Research Agenda, Radical Collocation, and Rapid Innovation
The research agenda of Par Lab was 2.5 years in the making. The goal is the make it easy to write correct programs that run efficiently on manycore systems, and which scale as the number of cores doubles every two years.

Taking a fresh approach to this longstanding problem, the research will be driven by compelling applications developed by domain experts. To get breadth as well as depth, we will ensure that our technology works well for the 13 motifs. These motifs help us avoid the pitfalls of traditional benchmarks, give us a multidisciplinary vocabulary, help bootstrap the research in the early years, and provide a starting point for our parallel libraries and frameworks.

The development of parallel software is the heart of the research agenda. We divide the task into two layers: an efficiency layer that aims at low overhead for 10% of the best programmers, and a productivity layer for the rest of the programming community that reuses the parallel software developed at the efficiency layer. Key to the success of this approach is efficient composition, and so we will develop a Composition and Coordination Language to aid in that task. Correctness is entwined with our goals of productivity and efficiency, so we restrict the productivity layer to prevent many types of concurrency errors, and then rely on specification generations, modular verification, directed automated unit testing, and runtime monitoring to discover those that occur.

The role of the operating system and the architecture is to support software and applications in achieving these goals. Our OS philosophy is to develop a thin virtual machine monitor and then a library of traditional OS functions, and to schedule bandwidth as well as hardware resources. Our architecture philosophy is to avoid pre-packaged hardware solutions and instead provide software-composable primitives as required by the developers, including primitives for partitioning, fast synchronization, and reconfigurable memory hierarchies.

As we said at the beginning of Section 3, a key part of our fresh approach to parallelism is to creating a culture that encourages multidisciplinary interaction and cooperation. Critical to our multidisciplinary project culture has been our twice a year, three-day retreats, where everyone on the project and industrial guests review progress and offer advice. (We can't imagine how to manage a large project without retreats. (Patterson 2002)) The first retreat was held January 7-9, 2008 and the second will be June 2008. We invite experts from applications and the IT industry to our retreats to give us continuing feedback on the project, as is our tradition.

The second piece of this culture is to copy the success of the Berkeley RAD Lab by moving UPCRC faculty and students into a single, open space to accelerate innovation by living together and by encouraging spontaneous multidisciplinary meetings (Teasley, Covi et al. 2000). The eight PIs will live in the lab and get the largest graduate student support. The six Co-PIs will retain their offices, and they will (co-) supervise students or post-docs who work in the lab. Thus, we will interact daily, rather than shortly before site visits. Such a space will significantly simplify the management and coordination of such a large project.

We intend to follow a third part of the Berkeley culture, which is to share the technology that we develop to help other researchers working on parallelism. All researchers want to see their research have impact. A difficult question is whether it's more effective to try to transfer technology to existing companies or to transfer technology by starting new companies. While all universities do both, Berkeley is more often associated with the former approach and other universities are known for the latter. A report from the National Academies of Engineering and Sciences shed light on the impact of these contrasting traditions. It reports on 19 examples of IT research that eventually led to multibillion dollar IT industries. Figure 5 associates the 16 universities cited in the report with the 19 multibillion dollar industries. The last two rows summarize the total number of citations and the research impact from just the last two decades. Berkeley leads in total industry involvement, with 7 of the 19 multibillion dollar industries over four decades. Furthermore, in the last two decades Berkeley helped create 5 of those multibillion-dollar industries, almost as many in this period as several top universities combined together.

While the goal of the Par Lab is daunting, we believe that our overall approach, the talents of the faculty and students involved, Berkeley's success at technology transfer, and the financial support from Intel and Microsoft offer the Par Lab an excellent chance to make timely progress on this difficult challenge facing our field.

#	$1B+ Industry	Berkeley	Caltech	Cambridge	CERN	CMU	Hawaii	Illinois	MIT	Purdue	Rochester	Stanford	Tokyo	Toronto	UCLA	Utah	Wisconsin
1	Timesharing	■							■								
2	Client/server	■			■	■											
3	Graphics								■							■	
4	Entertainment								■		■						
5	Internet						■								■		
6	LANs			■										■			
7	Workstations								■			■					
8	GUI										■						
	Subtotal (from ≈1960 to ≈1980)	**2**	0	1	1	1	1	**0**	**4**	0	2	**1**	0	1	1	1	0
9	VLSI design	■	■														
10	RISC processors	■										■					
11	Relational DB	■															■
12	Parallel DB												■		■		■
13	Data mining											■					■
14	Parallel computing		■			■		■									
15	RAID disk arrays	■															
16	Portable communication	■								■							
17	World Wide Web				■			■									
18	Speech recognition					■			■								
19	Broadband last mile											■			■		
	Subtotal (from ≈1980 to ≈2000)	**5**	2	0	1	2	0	**2**	**1**	1	0	**3**	1	0	2	0	3
	Total (from ≈1960 to ≈2000)	**7**	2	1	2	3	1	**2**	**5**	1	2	**4**	1	1	3	1	3

Figure 5. 16 Universities associated with 19 multibillion dollar IT industries. The four finalists for the Universal Parallel Computing Research Center were Berkeley, Illinois, MIT, and Stanford. Source: Derived from Figure 1 of *Innovation in Information Technology*, CSTB Report, National Research Council of the National Academies, National Academies Press, 2003.

13

Appendix A

This appendix gives more details on each of the 13 projects listed in Figures 3 and 4.

A.1 Personalized Medicine for Coronary Heart Disease (CHD)
Project Themes: Application-driven research, Motifs
Principal Investigator: Keaveny, Demmel, Yelick

CHD, the nation's single leading cause of death, accounts for over 450,000 deaths/year in the U.S. with 16M Americans currently suffering symptoms and 72M having high blood pressure. Our goal is to enable physiologically realistic models of blood flow through arteries based on a 3D medical image of the patient's blood vessels during a patient visit. This has the potential to profoundly alter the diagnosis and treatment of CHD. Local computation is critical, since there are barriers to transferring medical images through hospital firewalls. Ultimately, patients will own devices that perform computations to predict and assess their personalized health benefits of treatment, diet, and exercise using daily monitoring of blood pressure, medication levels, and other sensor devices. We will implement rapid, local analysis techniques for finite element-based computational solid/fluid dynamics analyses on blood vessels, based on CT angiography exams (Jones, Athanasiou et al. 2006; Frauenfelder, Boutsianis et al. 2007). We will use autotuning for sparse matrices and build on expertise in parallel simulations of large non-linear solid problems (Adams, Bayraktar et al. 2004), clinical application of physiological (but simpler) models for osteoporosis (Eswaran, Gupta et al. 2006; Keaveny, Donley et al. 2006) and simulation of fluids with immersed elastic structures (Givelberg and Yelick 2006).

A.2 Hearing Augmentation and Music
Project Themes: Application-driven research, Motifs, and Engineering Parallel Software
Principal Investigators: Wessel, Wawrzynek, Lee

These compute-intensive audio and music applications share common software components, are mappable to motifs, require real-time, low-latency I/O (<<5ms), and high reliability. Real-time is needed to ensure that audio output samples are generated at a steady rate. If a sample is late, an artifact (audio glitch or click) results. Reliability is critical, as failures cannot be tolerated in concert presentations or consumer applications.

Handsets for Hearing Augmentation (HA). This application will run on a many-core handset with wireless connections to ear bud or hearing-aid devices equipped with microphones. We implement dynamics processing, multi-band compressors, noise reduction, source separation, and beam forming in an adaptive manner. These will result in a hearing aid that responds dynamically to the changing context of the listener selecting optimal settings for music (Wessel, Fitz et al. 2007), speech, television, car, and so forth.

Large Microphone and Speaker Arrays. Dense arrays of speakers and microphones will enhance listener experience in conference calls, living rooms, and in concert halls. Wavefield synthesis allows us to place auditory images in fixed locations in a room. Beam-forming microphone arrays aid in location and separation of sound sources. We have demonstrated success on a 120-channel speaker array and will provide a 400-channel speaker and matched microphone array as a testbed.

Handsets and Laptops as Hosts for a Broad Class of Musical Instruments. The electronic musical instruments of the future will use mobile computing devices – witness the current use of laptops in live-performance contexts. Such mobile musical instrumentation involves not only real-time audio input and output but also the use of gestural controllers built with various kinds of sensors. The analysis and mapping of user gestures to musical processes can in itself be computationally intensive. Building on current trends in multi-touch surfaces, we have specified a controller system based on a dense array of *taxels,* the haptic analog of *pixels.* We propose to sample a 2k by 2k array of such *taxels* at an 8 kHertz rate and use algorithms from computer vision to

recognize and map hand gestures to a variety of processes that generate musical material. Using this approach, we will get at the dynamic details of what we call the hand-force-image. Here the computational requirements are a factor of 100 more demanding than the analogous processing of a visual image. One thing is clear: the computer music community has an insatiable appetite for real-time computation, be it for audio generation and processing itself or for the analysis of performance gestures.

Our work in these three areas of audio processing, hearing augmentation, speaker arrays, and musical instrumentation has benefited greatly from the use of graphical data-flow inspired programming environments, e.g., Max/MSP/Jitter (Zicarelli 1998), PD (Puckette, Apel et al. 1998), and Open Sound World (OSW) (Chaudhary, Freed et al. 2000). These programming languages and their environments have provided us with a very high degree of productivity but have not been adapted to multicore much less manycore. We will combine our extensive experience with these languages with features of our manycore programming layers to retarget the languages for manycore.

A.3. Automatic Speech Recognition (ASR)
Project Themes: Application-driven research, Motifs, and Engineering Parallel Software
Principal Investigators: Morgan, Asanovic
Conventional ASR systems have found limited acceptance because they work reliably only under good acoustic situations. We will use the computational capabilities of manycore systems to implement novel ASR systems that perform reliably in noisy environments, such as meetings and teleconferences. We will use our own feature generation subsystems, including Quicknet and Feacalc (Hermansky and Morgan 1994; Morgan, Zhu et al. 2005). These systems are frequently downloaded (ICSI 2007) and we have used them in combination with SRI's DECIPHER ASR components (Stolcke, Chen et al. 2006) to produce a collaborative system that has been extremely successful at every NIST evaluation on meeting speech. For this project we will use elements from a publicly available system such as Cambridge's HTK (used at universities worldwide, and recently updated to include large vocabulary enhancements) or IDIAP's Juicer (IDIAP 2006).

We envision using these computationally intensive ASR technologies to implement a digital diary application. The digital diary provides a highly interactive environment for reviewing a meeting.

The proposed application will identify speakers, provide rapid transcriptions of speech, and provide a multimodal (text, speech, gesture) query system for searching and summarizing content.

A.4 Fast, Energy-Efficient Content-Based Image Recognition (CBIR)
Project Themes: Application-driven research, Motifs, Engineering Parallel Software
Principal Investigators: Keutzer
Useful search of large media databases is currently only possible by using explicit tags from the user. Content-based image retrieval (CBIR) systems (Smeulders, Worring et al. 2000) search large image databases based on implicit criteria generated from a set of positive and negative examples. Such functionality is quickly becoming a necessity, since the value of an image database depends on the ability to find relevant images. The system core is a set of feature extractors and classifiers. The feature extraction routines analyze the images to create various signatures from the image, while the classifiers learn boundaries separating useful and irrelevant images, and then classify the remaining images in the database. The classifiers are based on support-vector machines and k-nearest neighbor classifiers (both use matrix motifs). Current image classification systems are performance limited. Manycore will enable more intelligent feature extraction and more precise queries, thereby increasing the relevance of search results. We will leverage Intel's PIRO (Personal Image Retrieval Organizer**)** code-base consisting of 35K

lines of code for the base content-based image retrieval system and ~165K lines of libraries for feature extraction and other utilities. We have already performed a conceptual re-architecting of PIRO using a hierarchical pipe-and-filter stream pattern at the top level. The feature extractor, classifier trainer, and classifier elements of PIRO are then filters based on variants of the MapReduce (Dean and Ghemawat 2004) pattern.

A.5. Parallel Web Browser
Project Themes: Application-driven research, Motifs, Engineering Parallel Software
Principal Investigators: Bodik, Asanovic
With 4G networks and better output devices, building a parallel browser will be the last step to desktop-quality, AJAX-full browsing on an energy-limited handheld (Bodik, Jones et al. 2007). Only a manycore handheld will provide the cycles needed for *parsing* (including translation to DOM)*, rendering* (including page layout), and *scripting*—the three primary browser activities. Unfortunately, all three are considered nearly inherently sequential. In this project, we will explore techniques to parallelize all three. Older discarded styles of parsers appear promising for parallelization. Although irregular dependencies make parallelization of rendering challenging (e.g., changing one letter may force re-layout of an entire page), rendering is a dynamic programming motif composed with a graph traversal motif, and is parallelizable with some software speculation. Our goal is to turn every website developer into a parallel programmer without plaguing them with threads and/or locks. A transactional DOM has been proposed by others (Eich 2007; O'Callahan 2007), but we think we can do without threads and transactions, remaining entirely implicitly parallel thanks to a streaming language, SkipJax (derived from Brown University's FlapJax (FlapJax 2007)) that combines reactive and media programming.

A.6. Coordination and Composition Language for Productivity Programming
Project Theme: Engineering Parallel Software
Principal Investigators: Yelick, Keutzer, Lee, Bodik, Sen, Asanovic
We will develop a language for specifying high-level program composition and coordination to provide programmers with a programming model that is safe and efficient. The language will include support for parallelism patterns, such as divide-and-conquer (Blumofe, Joerg et al. 1995), MapReduce (Dean 2006), data-driven execution (Kale and Krishnan 1996; Cicotti and Baden 2006), and data parallelism (Blelloch, Chatterjee et al. 1994; Charles, Donawa et al. 2005; Chamberlain, Callahan et al. 2007). The language will also support the specification of side effects on functions and a rich set of data partitioning operations for shared data structures (Lucassen and Gifford 1988; Clarke, Potter et al. 1998). The research products will be a language definition, analyses to ensure lack of concurrency errors (Cheng, Feng et al. 1998; Naik, Aiken et al. 2006; Kamil and Yelick 2007), runtime techniques for composing schedulers within various frameworks, and a prototype implementation that runs on existing multicore platforms and our own manycore design. We will develop use the coordination and composition language to build domain-specific application frameworks for our driving applications, e.g., a parallel streaming/pipe-and-filter environment for media applications (audio or image) with transformations based on spectral and structured grid libraries.
(Objective: The overarching goal of the work is that we demonstrate the creation of multiple application frameworks, which demonstrate 10X productivity improvements over handcrafted parallel implementations.)

A.7 Software RAMP for Efficiency Level Programming
Project Theme: Engineering Parallel Software, Motifs
Principal Investigators: Yelick, Asanovic, Bodik, Kubiatowicz
The goal of this project is to accelerate research by our own team and others in investigating programming models, libraries and frameworks for programming multicore and manycore

machines. Software RAMP will be a programming model for existing multicore and future manycore systems. It will include a basic shared address space programming language extending UPC (UPC 2005; Yelick, Bonachea et al. 2007) with explicit non-blocking DMA operations to move data from off-chip DRAM and (as needed) between local stores on a chip. The language will include synchronization, atomic operations (Ananian, Asanovic et al. 2005), and collective communication. There will be variations of the language for static and dynamic threading models. Portability is critical for widespread use, although some machine-specific features may be required to enable exploration of novel hardware. We will incorporate our synthesis tools into this language and build frameworks and libraries within it. Code at this level will also create a test suite for correctness tools and for dynamic hardware and OS features that help identify concurrency errors. (Objective: The overarching goal of the work is that by the third year we have demonstrated that expert programmers can use SW RAMP to achieve results within 20% of the performance (speed and power dissipation) of hand-crafted implementations.)

A.8 Correctness in Parallel Programs
Project Theme: Engineering Parallel Software, Motifs
Principal Investigators: Sen, Necula
The goal of this project is to build scalable tools to improve confidence in the correctness of parallel programs. The specific objectives are: 1) Demonstrate that automated unit testing of parallel modules is no harder than their sequential counterparts are. 2) Demonstrate that modular verification of the motifs is tractable. 3) Demonstrate that at least 60% of the serial contracts required for modular verification and unit testing can be generated automatically. 4) Demonstrate that our verification and testing tools can give more than 90% confidence in correctness.

A.9 Automatic Performance Tuning (Autotuning)
Project Theme: Engineering Parallel Software, Motifs
Principal Investigators: Demmel, Yelick, Asanovic, Bodik, Patterson
Autotuning uses a combination of empirical search and performance modeling to create highly optimized libraries tailored to specific machines. We will identify critical libraries based on application needs and build autotuners in the efficiency layer. We will also develop fundamental technology, including optimization and code generation strategies for manycore, search algorithms, and performance models to guide search and aid in identifying hardware bottlenecks. We will leverage existing autotuners from ourselves and others, adding support for novel hardware features, and build new autotuners when none exist. We will expand our tuning activities in the BeBOP and LAPACK (Anderson, Bai et al. 1995) groups in several motifs linear algebra, regular and irregular meshes, as well as collective communication routines. Priorities will come from application needs, e.g., the health application requires repeated sparse matrix-vector multiplies, which can be treated as a single operation to save repeated reads of the matrix; the resulting code is very complicated, so we plan to use sketching to fill in pieces of the optimized version.

A.10 Programming by Sketching
Project Theme: Engineering Parallel Software
Principal Investigators: Bodik, Demmel, Necula, Sen, Yelick
Expert programmers will write highly optimized library implementations using sketching and autotuning. Programmers may sketch sophisticated implementations, which will then be synthesized and autotuned. Sketching serves as an alternative to optimizing compilers for the efficiency layer, and supports optimizations that adapt to the application and hardware needs. For example, a data-structure-agnostic sketch will be synthesized into a library module composable with the application data structures. We will develop sketching for 1) grid operations that go beyond stencils, 2) lock-free data structures, and 3) turning synchronous specifications into

asynchronous implementations that use non-blocking memory operations. We will add linguistic support for sketching into language(s) for programming in the efficiency layer, develop synthesizers for completing holes in sketches, and build tool support for sketching. The goals are to make sketching as natural as programming (with no formal systems training) and to make synthesis scalable enough for high-performance implementations of most motifs.

(Objective: The overall success metric for this work is that where Sketching is applied it will show a 10X productivity improvement over developing parallel code by hand.)

A.11 Manycore Operating Systems
Project Themes: OS and Architecture
Principal Investigators: Kubiatowicz, Asanovic
The goal of this project is to develop a functioning operating system for a manycore system. We will simultaneously explore mechanisms to *aid* in parallel programming while investigating *opportunities* to improve performance, fault-tolerance, and security. We exploit several key ideas:

Spatial Partitioning: Groups of processors can be combined into protected "partitions". Boundaries will be enforced through hardware mechanisms restricting, among other things cross-partition shared memory. Messaging between partitions will be restricted based on a flexible, tag-based access control mechanism. OS functionality such as device drivers and file systems will be spatially distributed rather than time-multiplexed; we refer to this as "spatial deconstruction."

Minimalism: Only a thin "hypervisor" layer is resident on every processor. Many system facilities will be *linked at user level* as a set of optional "systems libraries." Fine-grained hardware protection mechanisms will allow direct, user-level access to facilities such as networking and I/O. Parallel applications will be given "bare metal" partitions of processors that are dedicated to the given application for sufficiently long to provide performance predictability.

User-Level Protected Messaging: Messages will be used to cross protection domains rather than more traditional trap-based mechanisms. Through hardware mechanism and/or static analysis, applications will have direct, user-level access to network interfaces and DMA. Further, fast exception handling and hardware dispatch of active message handlers will permit low overhead communication without polling. The OS will directly support user-level messages.

Integrated Support for Checkpoint/Restart: Direct support for checkpoint and restart will be provided to users. In cooperation with the compiler and execution frameworks, this will provide fault-tolerance that is minimally invasive to the bulk of parallel programmers. We anticipate combining this mechanism with dependency tracking for speculative execution.

A.12. Manycore Architecture
Project Themes: OS and Architecture
Principal Investigators: Asanovic, Kubiatowicz, Patterson, Wawrzynek
We will develop a new manycore architecture to support the needs of the efficiency layer. 1) A **dynamically reconfigurable memory hierarchy** that can support both software-managed memories and a cache-coherent or transactional memory (TM) system. (Objective: Hybrid HW/SW TM system performs within 20% of dedicated HW); 2) **Synchronization primitives** (barriers, active messages, atomic memory operations (AMOs)) to support efficiency layer parallel run-time systems (Objective: Threads can synchronize every 100 instructions with <20% slowdown); 3) **Efficient on-chip networks** and protocols with support for low-latency cross-chip communication and QoS guarantees (Objective: Can send packet across chip with <2X latency hit over optimized wire. Can run multiple real-time codes using >60% of system bandwidth); 4) Area and energy-efficient **latency-tolerant execution core designs** (Objective: code using <5% of cores can saturate available DRAM bandwidth with useful traffic). All designs will be implemented as emulations, but with delay and power figures based on realistic future VLSI process parameters.

A.13 RAMP Manycore System
Project Themes: OS and Architecture
Principal Investigators: Wawrzynek, Asanovic, Patterson
Using the separately funded RAMP infrastructure, we will develop and support a RAMP emulation model of our proposed Manycore architecture. This will support the OS and architecture projects by providing a platform for experimentation with new hardware features, and support all other projects by providing a standard, highly scalable, parameterizable, and observable platform for experimentation. (Objective: OS/architecture projects can change hardware design with single day turnaround; Emulation runs parallelized applications faster than native serial code on contemporary laptop while keeping full statistics.)

References

Adams, M. F., H. H. Bayraktar, et al. (2004). "Ultrascalable Implicit Finite Element Analyses in Solid Mechanics with over a Half a Billion Degrees of Freedom." SC2004, Pittsburgh, PA, November.

Ananian, C. S., K. Asanovic, et al. (2005). "Unbounded transactional memory." High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on: 316-327.

Anderson, E., Z. Bai, et al. (1995). "LAPACK Users' Guide, Release 2.0." SIAM, Philadelphia 326: 327.

Asanovic, K., R. Bodik, et al. (2006). "The Landscape of Parallel Computing Research: A View from Berkeley." Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December 18: 2006-183.

Barringer, H., A. Goldberg, et al. (2004). Rule-Based Runtime Verification. Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04).

Blelloch, G. E., S. Chatterjee, et al. (1994). "Implementation of a Portable Nested Data-Parallel Language." Journal of Parallel and Distributed Computing 21(1): 4-14.

Blumofe, R. D., C. F. Joerg, et al. (1995). "Cilk: an efficient multithreaded runtime system." Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming: 207-216.

Bodik, R., C. Jones, et al. (2007). "Browsing Web 3.0 on 3.0 Watts: Why browsers will be parallel." from http://parallelbrowser.blogspot.com/2007/09/hello-world.html.

Buttari, A., J. Dongarra, et al. (2007). "Multithreading for synchronization tolerance in matrix factorization." Journal of Physics: Conference Series 78(1): 012028.

Chakrabarti, S. and K. Yelick (1993). "Implementing an irregular application on a distributed memory multiprocessor." Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming: 169-178.

Chamberlain, B. L., D. Callahan, et al. (2007). "Parallel Programmability and the Chapel Language." International Journal of High Performance Computing Applications 21(3): 291-312.

Charles, P., C. Donawa, et al. (2005). X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. Proceedings of the ACM 2005 OOPSLA Conference.

Chaudhary, A., A. Freed, et al. (2000). An Open Architecture for Real-time Music Software. Proceedings of the International Computer Music Conference 2000, ICMA.

Cheng, G.-I., M. Feng, et al. (1998). Detecting Data Races in Cilk Programs that Use Locks. Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98), Puerto Vallarta, Mexico.

Choi, J. D., K. Lee, et al. (2002). Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM.

Cicotti, P. and S. B. Baden (2006). "Poster reception---Asynchronous programming with Tarragon." Proceedings of the 2006 ACM/IEEE conference on Supercomputing.

Clarke, D. G., J. M. Potter, et al. (1998). Ownership Types for Flexible Alias Protection. OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver. British Columbia, Canada, ACM Press, New York.

Cosnard, M., E. Jeannot, et al. (2001). Scheduling of Parameterized Task Graphs on Parallel Machines. Nonlinear Assignmen Problems: Algorithims and Applications. L. Pitsoulis and P. Pardalos, Kluwer Academic Publishers: 217-236.

Date, C. J. (1989). A guide to the SQL standard, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Dean, J. (2006). "Experiences with MapReduce, an abstraction for large-scale computation." International Conference on Parallel Architecture and Compilation Techniques.

Dean, J. and S. Ghemawat (2004). "MapReduce: Simplified Data Processing on Large Clusters." Proceedings of the 6th OSDI: 137-150.

Demmel, J., J. Dongarra, et al. (2005). "Self-adapting linear algebra algorithms and software." Proceedings of the IEEE 93(2): 293-312.

Demmel, J. W., J. R. Gilbert, et al. (1999). "An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination
." SIAM J. Matrix Analysis and Applications **20**(4): 915-952.

Dinning, A. and E. Schonberg (1991). Detecting Access Anomalies in Programs with Critical Sections. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

Eich, B. (2007). "Threads Suck." from http://weblogs.mozillazine.org/roadmap/archives/2007/02/threads_suck.html.

Eicken, T. v., D. E. Culler, et al. (1992). "Active Messages: a Mechanism for Integrated Communication and Computation." Proceedings of the 19th Int'l Symp. on Computer Architecture, May 1992.

Engler, D. R., M. F. Kaashoek, et al. (1995). "Exokernel: an operating system architecture for application-level resource management." Proceedings of the fifteenth ACM symposium on Operating systems principles: 251-266.

Eswaran, S. K., A. Gupta, et al. (2006). "Cortical and Trabecular Load Sharing in the Human Vertebral Body." Journal of Bone and Mineral Research **21**: 307-314.

Flanagan, C. and S. N. Freund (2001). Detecting Race Conditions in Large Programs. Proceedings of the Program Analysis for Software Tools and Engineering Conference.

Flanagan, C. and S. N. Freund (2004). Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. Proceedings of the 31st Symposium on Principles of Programming Languages (POPL'04).

Flanagan, C., S. N. Freund, et al. (2005). "Modular Verification fo Multithreaded Programs." Theoretical Computer Science **338**(1-3): 153-183.

FlapJax. (2007). from http://www.flapjax-lang.org/.

Frauenfelder, T., E. Boutsianis, et al. (2007). "In-vivo flow simulation in coronary arteries based on computed tomography datasets: feasibility and initial results." European Radiology **17**(5): 1291-1300.

Gelernter, D. and N. Carriero (1992). "Coordination languages and their significance " Commun. ACM **35**(2): 97-107.

Givelberg, E. and K. Yelick (2006). "Distributed Immersed Boundary Simulations in Titanium." SIAM Journal on Scientific Computing **28**(4): 1361-1378.

Godefroid, P., N. Klarlund, et al. (2005). DART: Directed Automated Random Testing. Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI).

Gürsoy, A. and L. V. Kale (2004). "Performance and modularity benefits of message-driven execution." Journal of Parallel and Distributed Computing **64**(4): 461-480.

Havelund, K. and G. Ro (2001). Monitoring Java Programs with Java PathExplorer, Elsevier Science.

Henzinger, T. A., R. Jhala, et al. (2005). Permissive Interfaces. Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, New York.

Henzinger, T. A., R. Jhala, et al. (2003). Thread-Modular Abstraction Refinement. Proceedings of the 15th International Conference on Computer-Aider Verification (CAV),
Lecture Notes in Computer Science 2725, Springer-Verlag.

Hermansky, H. and N. Morgan (1994). "RASTA processing of speech." Speech and Audio Processing, IEEE Transactions on **2**(4): 578-589.

Hill, M. D., D. Hower, et al. (2007). A Case for Deconstructing Hardware Transactional Memory Systems. Univ. of Wisconsin Computer Sciences Technical Report CS-TR-2007-1594, June 2007.

ICSI. (2007). "ICSI Speech Group Tools." from http://www.icsi.berkeley.edu/Speech/icsi-speech-tools.html.

IDIAP (2006). Juicer: A Weighted Finite-State Transducer Speech Decoder
Report 06-21. IDIAP Research Reports. Martigny, Switzerland, IDIAP.

Jhala, R. and R. Majumdar (2007). Interprocedural Analysis of Asynchronous Programs. POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium Nice, France, ACM Press, New York.

Johnson, R. E. (1997). "Frameworks=(components+ patterns)." Communications of the ACM **40**(10): 39-42.

Jones, C. M., T. Athanasiou, et al. (2006). "Multi-slice computed tomography in coronary artery disease." European Journal of Cardio-Thoracic Surgery **30**(3): 443.

Kale, L. V. and S. Krishnan (1996). Parallel Programming Using C++. G. V. Wilson and P. Lu. Cambridge, MA, USA MIT Press**:** 175-213.

Kamil, A. and K. Yelick (2007). Hierarchical Pointer Analysis for Distributed Programs. The 14th International Static Analysis Symposium (SAS 2007, Kongens Lyngby, Denmark.

Karnofsky, K. (1996). "Speeding DSP algorithm design." Spectrum, IEEE **33**(7): 79-82.

Keaveny, T. M., D. W. Donley, et al. (2006). "The effects of teriparatide and alendronate on vertebral strength as assessed by finite element modeling of QCT Scans in women with osteoporosis." J Bone Miner Res.

Lewis, B. and K. Richards. (2003). "LU Factoriaztion Case Study Using FAST: Dataflow Parallelism with the Forte Application Scalability Tool." from http://developers.sun.com/prodtech/cc/articles/FAST/lu_content.html.

Lucassen, J. M. and D. K. Gifford (1988). Polymorphic Effect Systems. Proceeding of the ACM Conference on Prinicples of Programming Languages.

Lucco, S. and O. Sharp (1991). "Parallel programming with coordination structures." Annual Symposium on Principles of Programming Languages, 1991.

Majumdar, R. and K. Sen (2007). Hybrid Concolic Testing. Proceedings of the 29th International Conference on Software Engineering (ICSE'07), IEEE.

Mattson, T. G., B. A. Sanders, et al. (2005). Patterns for parallel programming, Addison-Wesley Boston.

Meyer, B. (1992). "Applying "Design by Contract"." Computer **25**(10): 40-51.

Morgan, N., Q. Zhu, et al. (2005). "Pushing the envelope-aside." Signal Processing Magazine, IEEE **22**(5): 81-88.

Naik, M., A. Aiken, et al. (2006). Effective Static Race Detection for Java. .

Narayanasamy, S., Z. Wang, et al. (2007). "Automatically Classifying Benign and Harmful Data Racesallusing Replay Analysis." SIGPLAN Not. **42**(6): 22-31

Necula, G. C., J. Condit, et al. (2005). "CCured: Type-Safe Retrofitting of Legacy Software." ACM Transactions on Progamming Languages and Systems **27**(3): 477-526.

O'Callahan, R. (2007). Mozilla Corporation.

O'Hanlon, C. (2006). "A conversation with John Hennessy and David Patterson." Queue **4**(10): 14-22.

Patterson, D. (2002). "Fostering Community Within a CS&E Department: A Berkeley Perspective." Computing Research News **14**(4): 4-7.

Pike, R. (2005). "Interpreting the data: Parallel analysis with Sawzall." Scientific Programming **13**(4): 277-298.

Puckette, M., T. Apel, et al. (1998). "Real-time audio analysis tools for Pd and MSP." Proceedings of the International Computer Music Conference: 109–112.

Rosenblum, M. and T. Garfinkel (2005). "Virtual Machine Monitors: Current Technology and Future Trends." IEEE Computer **38**(5): 39-47.

Savage, S., M. Burrows, et al. (1997). "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." ACM Transaction on Computer Systems **15**(4).

Sen, K., D. Marinov, et al. (2005). CUTE: AConcolic Unit Testing Engine foor C. Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05).

Sen, K. and M. Viswanathan (2006). Model Checking Multithreaded Programs with Asychronous Atomic Methods. Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06).

Shaw, M. and D. Garlan (1996). Software architecture: perspectives on an emerging discipline, Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

Smeulders, A. W. M., M. Worring, et al. (2000). "Content-based image retrieval at the end of the early years." Pattern Analysis and Machine Intelligence, IEEE Transactions on **22**(12): 1349-1380.

Solar-Lezama, A. and T. Rabbah (2005). Programming by sketching for bit-streaming programs Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation Chicago, IL

Stolcke, A., B. Chen, et al. (2006). "Recent innovations in speech-to-text transcription at SRI-ICSI-UW." IEEE Transactions on Speech, Audio and Language Processing.

Teasley, S., S. Covi, et al. (2000). How does radical collocation help a team succeed? Proceedings of the 2000 ACM conference on Computer supported cooperative work.

UPC (2005). The UPC Language Specifications, version 1.2. Lawrence Berkeley National Laboratory Technical Report, LBNL-59208, 2005. . Berkeley, Lawrence Berkeley National Laboratory.

Wawrzynek, J., D. Patterson, et al. (2007). "RAMP: A Research Accelerator for Multiporcessoors." IEEE Micro.

Wessel, D., K. Fitz, et al. (2007). Optimizing Hearing Aids for Music Listening. 19th International Congress on Acoustics
 Madrid

Wessel, D. and M. Wright (2002). "Problems and prospects for intimate musical control of computers." Computer Music Journal **26**(3): 11-22.

Williams, S., L. Oliker, et al. (2007). Optimization of sparse matrix-vector multiply on emerging multicore platforms. Proc. Supercomputing, Reno 2007, Reno NV.

Witchel, E., J. Cates, et al. (2006). "Mondrian Memory Protection."

Xu, M., R. Bodik, et al. (2003). "A" flight data recorder" for enabling full-system multiprocessor deterministic replay." Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on: 122-133.

Yelick, K., D. Bonachea, et al. (2007). Productivity and Performance Using Partitioned Global Address Space Languages. Proceedings of Parallel Symbolic Computation (PASCO), , London, Ontario.

Yelick, K. A. and S. J. Garland (1992). "A parallel completion procedure for term rewriting systems." Conference on Automated Deduction, Saratoga Springs, NY.

Zicarelli, D. (1998). "An Extensible Real-Time Signal Processing Environment for Max." Proceedings of the 1998 International Computer Music Conference: 463–466.