A View of the Parallel Computing Landscape

Krstie Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick
Parallel Computing Laboratory, U.C. Berkeley

ABSTRACT

Industry needs help from the research community to succeed in its sudden shift to parallel computing. Failure could jeopardize both our field and the portions of the economy that depend upon rapidly improving information technology. This paper gives one view of the issues, and to give an example, describes one integrated approach to tackling the parallel challenge.

1. INTRODUCTION: THE POWER WALL

Over the last 60 years, the information technology industry has improved cost-performance of sequential computing by about 100 billion overall [20]. For most of the last 20 years, architects used the rapidly increasing transistor speed and budget made possible by silicon technology advances to double performance every 18 months. The implicit hardware/software contract was that increases in transistor count and power dissipation were OK as long as architects maintained the existing sequential programming model. This contract led to innovations that were inefficient in transistors and power—such as multiple instruction issue, deep pipelines, out-of-order execution, speculative execution, and prefetching—but which increased performance while preserving the sequential programming model.

This contract worked fine until we hit the power limit that a chip could dissipate. Figure 1 illustrates the abrupt change, plotting the projected microprocessor clock rates of the International Technology Roadmap for Semiconductors in 2005 and then just two years later[16]. They predicted that clock rates should exceed 10 GHz by now, topping 15 GHz next year! Note that Intel products are far below even the conservative 2007 prediction.

After crashing into the power wall, architects were forced to find a new paradigm to sustain ever-increasing performance. The industry decided that only viable option was to replace the single power-inefficient processor by many efficient processors on the same chip. The whole microprocessor industry thus declared that its future was in parallel computing, with an increasing number of processors or cores each technology generation, which occur every two years. This style of chip was labeled a multicore microprocessor. Hence, the leap to multicore is not based on a breakthrough in programming or architecture; it’s actually a retreat from the even harder task of building power-efficient, high-clock-rate, single-core chips [4].

Many startups sold parallel computers over the years. They all failed, as programmers accustomed to continuous improvements in sequential performance saw little need to explore parallelism. Convex, Encore, Floating Point Systems, INMOS, Kendall Square Research, MasPar, nCUBE, Sequent, and Thinking Machines are just the best-known members of the Dead Parallel Computer Society, whose ranks are legion. Given this sad history, multicore pessimism abounds. Quoting computing pioneer 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.” [19]

Jeopardy for the IT industry means opportunity for the research community. If researchers meet the parallel challenge, the future of IT is rosy. If they don’t, it’s not. Hence, there are few restrictions on potential solutions. Given an excuse to reinvent the whole software/hardware stack, this opportunity is also a once-in-a-career chance to fix other weaknesses in computing that have accumulated over the decades like barnacles on an old ship.

This short paper lays out one view of the opportunities, and then as an example describes in more depth the approach of the Berkeley Parallel Computing Lab. It is an update of two long technical reports, which include more details.[4][5]. Our goal in this paper is to recruit more parallel revolutionaries.

Figure 1. Microprocessor clock rates of Intel products vs. projections from the International Technology Roadmap for Semiconductors in 2005 and 2007. [16]

2. A VIEW OF THE PARALLEL BRIDGE

Figure 2 shows an analogy of a bridge to connect computers users on the right to the IT industry on the left. The left tower is hardware, the right tower is applications, and the long span that connects these two far-apart towers together is software. We use the bridge analogy throughout this paper.

We believe the goal of the revolution should be to make it easy to write programs that are efficient, portable, correct, and scale as the number of cores per microprocessor increases biennially as it has been to write programs for sequential computers. Note that we can fail overall if we fail to deliver even at one of these “parallel virtues.” For example, if parallel programming is unproductive, then this weakness will delay and reduce the number of programs that can exploit new multicore architectures.

2.1 The Hardware Tower

The power wall is forcing the change in the programming model, but the question is what to build. There is a technology sweet spot around a pipelined processor of 5 to 8 stages, which is most efficient in performance per joule and in silicon area.[4] Using

*Submitted to Communications of the ACM May 2008; Revised November 2008.*
simple cores means there is room for 100s of them. Moreover, having many such simple cores on a chip simplifies hardware design and verification since each core is simple and replication of cores is nearly trivial. As it’s easy to add spares to mask manufacturing defects, such “manycore” computers can also have higher yield.

One example comes from the world of network processors, which saw a great deal of innovation recently given the growth of the networking market. The winner is arguably the Cisco Silicon Packet Processor, also known as Metro. It has 188 5-stage RISC cores, plus 4 spares to help yield, and it dissipates just 35 Watts.

It may be reasonable to assume that manycore computers will be homogeneous like the Metro, but there is an argument for heterogeneous manycores as well. For example, suppose 10% of the time a program gets no speed up on a 100-core computer. To run this sequential piece twice as fast, assume a single fat core would need 10 times as many resources as a thin core due to larger caches, a vector unit, and so on. Using Amdahl’s Law, the speedups (relative to 1 thin core) of 100 thin cores and 90 thin cores for the parallel code plus 1 fat core for the sequential code:

\[
\text{Speedup}_{100} = \frac{1}{(0.1 + 0.9/100)} = 9.2 \text{ times faster}
\]

\[
\text{Speedup}_{90} = \frac{1}{(0.1/2 + 0.9/90)} = 16.7 \text{ times faster}
\]

In this example, a fat core needing 10 times as many resources would be more effective than the 10 thin cores it replaces [4][15].

One challenge for the hardware tower is that it takes 4 to 5 years to design and build chips and to port software to evaluate them. How can researchers rapidly innovate given this lengthy cycle?

2.2 The Software Span

We believe software is the main problem in bridging the gap between users and the parallel IT industry. Hence, the large size of the span in Figure 2 reflects the size of software’s challenges.

One challenge for the parallel software span is that sequential programming accommodates the wide range of skills of today’s programmers. Our experience in teaching parallelism suggests that not every programmer can understand the nitty gritty of concurrent software and parallel hardware: locks, barriers, deadlocks, load balancing, scheduling, and memory consistency are difficult concepts. How can researchers develop technology so that all programmers can benefit from the parallel revolution?

A second challenge is that two critical pieces of system software—compilers and operating systems—have become large and unwieldy and hence resistant to change. One estimate is that it takes a decade for a new compiler optimization to become part of production compilers. How can researchers innovate rapidly if compilers and operating systems evolve at a glacial pace?

A final challenge is how to measure improvement in parallel programming languages. The history has largely been of researchers deciding on what they think would be better and then building it for others to try. As human beings write programs, we wonder whether human psychology and human subject experiments shouldn’t play a larger role in this revolution [17].

2.3 The Applications Tower

The goal should be to find compelling applications that thirst for more computing than currently available and could absorb biennial increasing number of cores for the next decade or two. We believe that success does not require that the performance of all legacy software improve. Instead, we aim to create compelling applications that effectively utilize the growing numbers of cores while providing software environments that ensure that legacy code still works with acceptable performance.

Note that better is not only defined by performance. Advances could come, say, in battery life or reliability or security. To save the IT industry, researchers just need to demonstrate greater end-user value from an increasing number of cores.

3. THE BERKELEY PAR LAB

To give one concrete example of how to explore the parallel landscape, this section describes Berkeley’s Par Lab project. It illustrates just one of many potential approaches, and we won’t know for years which of our ideas will bear fruit. We hope this example inspires more researchers to participate to increase the chance of finding a solution before it’s too late for the IT industry.

Given a 5-year project, we projected the state of the field in 5-10 years. We predict IT will be driven to extremes in size, due to the increasing popularity of software as a service:

- **The datacenter is the server.** Amazon, Google, Microsoft, and others are racing to construct buildings with 10,000 servers to run software as a service (SaaS). They have also begun renting hundreds of machines by the hour to enable smaller companies to offer SaaS. We expect these trends to accelerate.

- **The mobile device (laptop/handheld) is the client.** 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 this to accelerate as well.

Surprisingly, the extremes share many characteristics. Both are concerned about power and energy: the datacenter due to the cost

---

1 In March 2007, Intel and Microsoft invited 25 universities to propose 5-year centers for parallel computing research; Berkeley’s effort was ranked first, and Illinois’ ranked second.

---

*Submitted to *Communications of the ACM* May 2008; Revised November 2008.*
of power and cooling, and the mobile client for battery life. Both are also concerned about cost: the datacenter because server cost is replicated 10,000 times, and mobile clients because of a lower unit price target. Finally, the software stacks are becoming similar, with more layers for mobile clients and increasing concerns about protection and security.

Many datacenter applications have ample parallelism across independent users, so we’re focusing on parallelizing applications for clients in Par Lab. We believe the multicore and manycore chips in the datacenter can benefit from the same tools and techniques developed for similar chips for mobile clients.

Given this projection of the future, we decided to take a fresh approach. The Par Lab will be driven top-down, from the applications first, then software, and finally hardware.

3.1 The Par Lab Application Tower

It’s an unfortunate computer science tradition to build research prototypes and then wonder why applications people don’t use them. We instead selected applications upfront to drive our research and provide concrete goals and metrics to evaluate progress. 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 and markets likely to dominate usage. 4) Enables technology for other applications. 5) Finally, there must be a local committed expert application partner will help design, use, and evaluate our technology. Here are the five initial applications.

- Music/Hearing. High-performance signal processing will permit a) concert-quality sound delivery systems for home sound systems and conference calls, b) composition and gesture-driven live-performance systems, and c) much improved hearing aids.
- Speech Understanding. Dramatically improved automatic speech recognition in moderately noisy and reverberant environments would greatly improve existing applications and enable new ones like a real-time meeting transmitter with rewind and search. Current transcribers can generate many errors depending on the acoustic conditions.
- Content Based Image Retrieval. Consumer image databases are growing so fast that they require automated search vs. manual labeling. Low error rates require processing very high dimensional feature spaces. Current image classifiers are too slow for fast response times.
- Personal Medical Advisor. Advanced physiological modeling of 3D medical images enables “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 the doctor make timely treatment decisions to encourage healthy behavior and reduce death rate.
- Parallel Browser. The browser will be the largest and most important application on many mobile devices. We will first parallelize sequential browser bottlenecks. Rather than parallelizing JavaScript programs, we argue for an actor language with implicit parallelism. Such a language may be accessible to web programmers while allowing us to extract the parallelism in the browser’s JIT compiler, thereby turning all website developers unknowingly into parallel programmers.

The application domain experts are first-class members of the Par Lab project. Rather than trying to answer design questions abstractly, we will ask our experts what they need. Success of the project will be judged by the user experience with the collective applications on our hardware-software prototypes. If successful, we can imagine building upon these five applications to create ones that are even more compelling. Here are two examples.

- Name Whisperer. Imagine your mobile client peeking out of your shirt pocket so as to recognize someone walking towards you to shake your hand. This application would search the personal image database and then whisper in your ear that “That man is John Smith. He got an A- from you in CS152 in 1993.”
- Health Coach. As your mobile client is always with you, you could take pictures and weigh your dishes (assuming it has a built-in scale) before and after each meal. It would also record how much you exercise. Given calories consumed and burned and an image of your body, it could show what you’ll look like in six months at this rate, and what you would look like if you ate less or exercised more.

3.2 The Par Lab Software Span

Software is the major effort of the Par Lab project.

3.2.1 Architecting Parallel Software with Design Patterns, Not Just Parallel Programming Languages

Our situation today is similar to that found in other engineering disciplines; where a new challenge emerges that requires a whole rethinking of the entire engineering process. For example, in civil architecture, Brunelleschi’s solution to the dome for the Cathedral of Florence required innovations in tools, building techniques, as well as a rethinking of the whole process of developing an architecture. We believe we are facing a similar magnitude of challenge and that parallel programming is overdue for a fundamental rethinking of the process of designing software.

Programmers have been trying to craft parallel code for decades, and we’ve learned a great deal about what works and what doesn’t work. Automatic parallelism doesn’t work. Compilers are great at low level scheduling decisions, but they can’t discover new algorithms to exploit concurrency. Programmers in HPC have shown that explicit technologies such as MPI and OpenMP can be made to work, but they all too often require heroic effort untenable for most commercial independent software vendors.

We believe that to engineer high quality parallel software, we need to rearchitect the software. To help us accomplish this goal, we use the device of a design pattern language. As described by civil architect Christopher Alexander, design patterns describe time-tested solutions to recurring problems within a well-defined context [3]. An example is Alexander’s family of entrances pattern, which addresses how to simplify comprehension of multiple entrances for a first-time visitor to a site. He defines a pattern language as a collection of related and interlocking patterns, constructed such that the patterns flow into each other as the designer solves a design problem.

Computer scientists are trained to think in well-defined formalisms. Pattern languages encourage a less formal and more associative way of thinking about a problem. A pattern language does not impose a rigid methodology. Instead, it fosters creative problem solving by providing a common vocabulary to capture
the problems encountered during design, and to identify potential solutions from among families of useful prior designs.

The observation that design patterns and pattern languages could be useful for software design is not new. One example is the book Design Patterns, which outlines patterns useful for object-oriented programming [12]. In building our own pattern language we found Shaw and Garlan’s report [23], which describes a variety of architectural styles that are useful for organizing software, to be very effective. That these architectural styles may also be viewed as design patterns was noted earlier by Buschmann [7]. In particular, we adopted Pipe-and-Filter, Agent-and-Repository, Process Control, and Event-Based architectural styles as structural patterns within our pattern language. To this list, we add MapReduce and Iterator as additional structural patterns.

These patterns define the structure of a program, but they do not indicate what is actually computed. Thus, another key part of our pattern language is the set of dwarfs of the Berkeley View reports [4][5]; see Figure 3. We believe that dwarfs are best understood as computational patterns that provide the computational interior of the structural patterns above. By analogy, the structural patterns describe a factory’s physical structure and general workflow. The computational patterns describe the factory’s machinery, flow of resources, and work-products. Structural and computational patterns can be combined to architect arbitrarily complex parallel software systems.

Convention holds that truly useful patterns are not invented but mined from successful software applications. To arrive at our list of computational patterns we began with Phillip Collela’s list of seven dwarfs of high performance computing. Then, over a period of two years we worked with domain experts to broadly survey other application areas including embedded systems, general-purpose computing (SPEC benchmarks), databases, games, artificial intelligence/machine learning, computer-aided design of integrated circuits, and high performance computing. We then focused in depth on the patterns in our applications described in Section 3.1. Figure 3 shows the results of our pattern mining.

Computational and structural patterns can be hierarchically composed to define a high-level software architecture of an application, but a complete pattern language for application design must at least span the full range from high-level architecture to detailed software implementation and tuning. Mattson et al’s book [18] was the first such attempt to systematize parallel programming using a complete pattern language. We combine the structural and computational patterns mentioned above in our pattern language to literally sit on top of algorithmic structures and implementation structures in the pattern language developed in Mattson’s book. The resulting pattern language is still under development, but it has already been successfully employed to develop the software architectures and parallel implementations of such diverse applications as content-based image retrieval, large-vocabulary continuous speech recognition, and timing analysis for integrated circuit design.

Patterns are conceptual tools that help a programmer to reason about a software project and develop an architecture, but they are not themselves implementation mechanisms for producing code.

3.2.2 Split Productivity and Efficiency Layers, not Just a Single General-Purpose Layer

A key research objective is to enable programmers to easily write programs that run as efficiently on manycore systems as on sequential ones. We believe productivity, efficiency, and correctness are inextricably linked and must be addressed together. We do not believe that these objectives can be accomplished with a single-point solution, such as one universal language. In our approach, productivity is addressed in a productivity layer that will use a common composition and coordination language to glue together the libraries and programming frameworks produced by the efficiency layer programmer. Efficiency will be principally handled by the use of an efficiency layer that is targeted for expert parallel programmers.

We believe the key to generating a successful multicore software developer community is to maximally leverage the efforts of parallel programming experts by encapsulating their software for use by the programming masses. We use the term programming framework to mean a software environment that supports the implementation of the solution proposed by the associated design pattern. The difference between a programming framework and a general programming model or language is that in a programming framework the customization is performed only at specified points that are harmonious with the style embodied in the original design pattern. An example of a successful sequential programming framework is the Ruby on Rails framework that is based on the Model-View-Controller pattern [26]. Users of the framework have ample opportunity to customize the framework but only in harmony with the core Model-View-Controller pattern.

Frameworks include libraries, code generators, and runtime systems, which assist the programmer with implementation by abstracting difficult portions of the computation and incorporating them into the framework itself. Successful parallel frameworks from the past encode the collective experience of the programming community’s solutions to recurring problems. We believe that basing frameworks on pervasive design patterns will help make parallel frameworks broadly applicable.

Productivity layer programmers will compose libraries and programming frameworks into applications with the help of a composition and coordination language [13]. 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 frameworks for specific applications, offering a programming experience similar to MATLAB or SQL.

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 systems do not offer a common low-level programming model for parallel code. We will define a thin portability layer 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 [27], which partitions memory between processors but currently not between on and off-chip.
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.

To summarize, to engineer parallel software programmers should start with effective software architectures. The software engineer would describe the solution to a problem in terms of a design pattern language. Based on this design pattern language, we’ll create a family of frameworks to help turn a design into working code. The general-purpose programmer will largely work with the frameworks and stay within what we call the productively layer. Specialist programmers trained in the details of parallel programming technology will work within the efficiency layer to implement the frameworks and map them onto specific hardware platforms. This approach helps general-purpose programmers create parallel software without having to master the low level details of parallel programming.

### 3.2.3 Generating Code with Search-Based Autotuners, Not Compilers

Compilers that automatically parallelize sequential code may have great commercial value as computers go from 1 to 2 to 4 cores, but we don’t believe they will scale up from 32 to 64 to 128 cores. Compiling will be even harder as the switch to multicore means microprocessors are becoming even more diverse, since there is no conventional wisdom yet for multicore architectures. For example, Table 1 shows the diversity in designs of x86 and SPARC multicore computers. In addition, as the number of cores increase, manufacturers will likely offer products with differing number of cores per chip to cover multiple price-performance points. They will also allow each core to vary its clock frequency to save power. Such diversity will make the goals of efficiency, scaling, and portability even more difficult for conventional compilers, at a time when innovation is desperately needed.

In recent years, autotuners have become popular for producing high-quality, portable scientific code for serial microprocessors [10]. Autotuners optimize a set of library kernels by generating many variants of a kernel and measuring each variant by running on the target platform. The search process effectively tries many or all optimization switches; hence, it may take hours to complete on the target platform. Search needs to be performed only once, however, when the library is installed. The resulting code is often several times faster than naive implementations. A single autotuner can be used to generate high-quality code for a wide variety of machines. In many cases, the autotuned code is faster.

#### Table 1 Autotuned Performance in GFLOPS/s on 3 kernels for dual-socket systems.

<table>
<thead>
<tr>
<th>MPU Type</th>
<th>Intel e5345 Xeon 4 out-of-order cores, 2.3 GHz</th>
<th>AMD 2356 Opteron X4 4 out-of-order cores, 2.3 GHz</th>
<th>Sun 5140 UltraSPARC T2 8 multithreaded cores, 1.2 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>SpMV Stencil LBTHD</td>
<td>SpMV Stencil LBTHD</td>
<td>SpMV Stencil LBTHD</td>
</tr>
<tr>
<td>Optimization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>1.0 1.3 3.5</td>
<td>1.4 1.5 3.0</td>
<td>2.1 0.5 3.4</td>
</tr>
<tr>
<td>NUMA</td>
<td>1.0 -- 3.5</td>
<td>2.4 2.6 3.7</td>
<td>3.5 0.5 3.8</td>
</tr>
<tr>
<td>Padding</td>
<td>-- 1.3 4.5</td>
<td>-- 3.1 5.8</td>
<td>-- 0.5 3.8</td>
</tr>
<tr>
<td>Vectorization</td>
<td>-- 4.6</td>
<td>-- 7.7</td>
<td>-- 9.7</td>
</tr>
<tr>
<td>Unrolling</td>
<td>-- 1.7 4.6</td>
<td>-- 3.6 8.0</td>
<td>-- 0.5 9.7</td>
</tr>
<tr>
<td>Prefetching</td>
<td>1.1 1.7 4.6</td>
<td>2.9 3.8 8.1</td>
<td>3.6 0.5 10.5</td>
</tr>
<tr>
<td>Compression</td>
<td>1.5 -- --</td>
<td>3.6 -- --</td>
<td>4.1 -- --</td>
</tr>
<tr>
<td>S/TLB block</td>
<td>-- 2.2 --</td>
<td>-- 4.9 --</td>
<td>-- 5.1 --</td>
</tr>
<tr>
<td>Collab thread</td>
<td>-- -- --</td>
<td>-- -- --</td>
<td>-- 6.7 --</td>
</tr>
<tr>
<td>SIMD</td>
<td>-- 2.5 5.6</td>
<td>-- 8.0 14.1</td>
<td>-- -- --</td>
</tr>
<tr>
<td>Final</td>
<td>1.5 2.5 5.6</td>
<td>3.6 8.0 14.1</td>
<td>4.1 6.7 10.5</td>
</tr>
</tbody>
</table>

Figure 3. Twelve computational patterns, seven general application areas, and five Par Lab applications. The color of a cell indicates the presence of that computational pattern in that application. Red/High; Orange/Moderate; Green/Low; Blue/Rare.
than vendor libraries that were specifically hand-tuned for the target machine. This surprising result is partly explained by the way the autotuner tirelessly tries many unusual variants of a particular routine. They also allow tuning to the particular problem size, unlike libraries. Autotuners also preserve clarity and help portability by reducing the temptation to mangle the source code to improve performance for one particular computer.

We believe that autotuning can also help with the production of parallel code. Parallel architectures, however, introduce many new optimization parameters, and so far, few successful autotuners for parallel codes exist. For any given problem, there may be several parallel algorithms, each with alternative parallel data layouts. The optimal choice may depend not only on the processor architecture but also on the parallelism of the computer and memory bandwidth. Consequently, in a parallel setting, the search space will be much larger than that for serial hardware.

For example, Table 1 shows the results of autotuning on three multicores for three kernels related to the dwarfs mentioned above: sparse matrix, stencil for PDEs, and structured grids [9][29][30]. This autotuned code is the fastest known for these kernels for all three computers. Performance increased by factors of 2 to 4 over standard code, which is much higher than you would expect from an optimizing compiler.

Efficiency layer programmers will build autotuners for use by domain experts and other efficiency layer programmers to help deliver on the goals of efficiency, portability, and scalability.

### 3.2.4 Synthesis with Sketching

One challenge in autotuning is how to produce the high-performance implementations explored by the search. One approach is to synthesize these complex programs. By doing so, we rely on the search not only for performance tuning but also for programmer productivity. To address the main challenge of traditional synthesis—the need for an expert to communicate his insights with a formal domain theory—we allow the insight to be communicated directly by the programmer who writes an incomplete program called a sketch. In the sketch, the programmer provides an algorithmic skeleton and the synthesizer supplies the low-level mechanics, by completing the holes in the sketch. The synthesized mechanics could be barrier synchronization expressions or tricky loop bounds in stencil loops. Our sketching-based synthesis is to traditional, deductive synthesis what model checking is to theorem proving: rather than interactively deriving a program, we search a space of candidate programs with constraint solving. The efficiency is achieved by reducing the problem to one solved with two communicating SAT solvers. In future work, we hope to synthesize parallel sparse matrix codes and data-parallel algorithms for new problems such as parsing.

### 3.2.5 Verification and Testing, Not One or the Other

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.

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 [22], predicate abstractions of the interface of a module [14], or multiple ownership types [8].

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. 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 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 has shown promise to improve software quality through automated test generation using a combination of static and dynamic analyses [21]. We will 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. We will 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.

### 3.2.6 Parallelism for Energy Efficiency

While the previous computer classes (desktops and laptops) reused software of their ancestors, the energy efficiency for handheld operation may need to come from data parallelism in tasks that are currently executed sequentially. The energy efficiency may come from three sources: 1) completing a task on slow parallel cores will be more efficient than completing the task in the same time sequentially on one fast core; 2) preferring data-parallel algorithms over other styles of parallelism, as SIMD and vector computers amortize the energy burnt on instruction delivery; and 3) message-passing programs may save the energy used by cache coherence.

We are applying these principles in our work on parallel web browser. In algorithm design, we observe that to save energy with parallelization, parallel algorithms must be close to work efficient; that is, perform no more total work than a sequential algorithm or else parallelization is counterproductive. The same argument applies for optimistic parallelization. Work efficiency is a demanding requirement since for some “inherently sequential” problems, like finite-state machines, only work-inefficient algorithms are known. In this context, we developed a nearly work-efficient algorithm for lexical analysis. We are also working on data-parallel algorithms for web page layout and we
are identifying parallelism in future web browser applications, attempting to implement them with efficient message passing.

3.2.7 Space-Time Partitioning for Deconstructed OS
We believe that space-time partitioning is crucial for manycore client operating systems. A spatial partition (or partition for short) is an isolated unit containing a subset of physical machine resources such as cores, cache partitions, guaranteed fractions of memory or network bandwidth, and an energy budget. Space-time partitioning virtualizes spatial partitions by time-multiplexing whole partitions onto available hardware, but at a coarse-enough granularity to allow efficient user-level scheduling in a partition.

The presence of space-time partitioning leads to a restructuring of systems services as a set of interacting distributed components. We propose a new "deconstructed OS" called Tessellation, structured around space-time partitioning and two-level scheduling between the operating system and application runtimes. Tessellation implements scheduling and resource management at the partition granularity. Applications and OS services (e.g. file systems) run within their own partitions. Partitions are lightweight, and can be resized or suspended with similar overheads to a context switch.

A key tenet of our approach is that resources given to a partition are either exclusive (such as cores or private caches) or guaranteed via a quality-of-service contract (such as a minimum fraction of network or memory bandwidth). During a scheduling quantum, the application runtime within a partition is given unrestricted "bare metal" access to its resources and may schedule tasks onto them in any way. Within a partition, our view bears much in common with the Exokernel [11]. In the common case, we expect many application runtimes to be written as libraries (similar to libOS). Our Tessellation kernel is a thin layer responsible only for the coarse-grain scheduling and assignment of resources to partitions, and the implementation of secure restricted communications between partitions. The Tessellation kernel is much thinner than traditional kernels or even hypervisors. It avoids many of the performance issues with traditional microkernels by providing OS services through secure messaging to spatially co-resident service partitions, rather than context-switching to time-multiplexed service processes.

3.3 The Par Lab Hardware Tower
Past parallel projects were often driven by the hardware, which set the application and software environment. The Par Lab is being driven top down from the applications, so the question this time is what can architects do to help with the goals of productivity, efficiency, correctness, portability, and scalability?

Here are four examples to suggest our approach:

- **Supporting OS Partitioning.** Our hardware architecture enforces partitioning of not only the cores and on-chip/off-chip memory, but also the communication bandwidth between these components providing quality-of-service guarantees. The resulting performance predictability improves parallel program performance, simplifies code autotuning and dynamic load balancing, and supports real-time applications.

- **Optional Explicit Control of the Memory Hierarchy.** Caches were invented so that hardware could manage a memory hierarchy without troubling the programmer. When it takes hundreds of clock cycles to go to memory, programmers and compilers try to reverse engineer the hardware controllers to make better use of the hierarchy. This backwards situation is especially apparent for hardware prefetchers, where programmers try to create the pattern that will invoke good prefetching. Our approach will be to allow programmers to rapidly turn a cache into an explicitly managed local store, and to turn the prefetch engines into explicitly controlled DMA engines. To make it easy to port software to our architecture, we also support a traditional memory hierarchy.

- **Accurate, Complete Counters of Performance and Energy.** Sadly, performance counters on current computers often miss important measurements, such as prefetched data, or are unique to a computer and require its designers to interpret their meaning. We will only include performance enhancements if they have counters to measure them accurately and intelligibly. Since energy is as important as performance, we also include energy counters so that software can improve both. Moreover, these counters must be integrated with the software stack to provide insightful measurements to the efficiency layer and productivity layer programmers. Ideally, this research would lead to a standard for performance counters so that schedulers and software develop kits can count on them on any multicore.

- **An Intuitive Performance Model.** The multicore diversity mentioned earlier exacerbates the already difficult job of programmers, compiler writers, and even architects. Hence, we developed an easy-to-understand, visual model that offers performance guidelines to identify bottle necks in the roofline model plots computational and memory bandwidth limits and then determines best possible performance of a kernel by examining the average number of operations per memory access. It also plots ceilings below the "roofline" to suggest which optimizations to try to improve performance. One goal of our new performances counters should be to provide everything needed to automatically create Roofline models.

One challenge from Section 2.1 was how to rapidly innovate at the hardware/software interface when it can take four to five years to build chips and run programs to evaluate them. Given the capacity of FPGAs today, researchers can prototype full hardware and software systems that run fast enough to investigate architectural innovations. This flexibility means researchers can "tape out" every day, rather than wait years. We will leverage the Research Accelerator for Multiple Processors (RAMP) Project to build flexible prototypes that are fast enough to run full software stacks—including new operating systems and our five compelling applications—to enable rapid architecture innovation using prototype software of the future rather than benchmarks of the past.

4. REASONS FOR OPTIMISM
Given the history of parallel computing, it’s easy to be pessimistic about our chances. In addition to industry’s desperation, the good news is that there are reasons researchers might succeed this time:

- **No Killer Microprocessor.** Unlike the past, no one is building the faster serial microprocessor. Programmers needing more performance have no other option than parallel hardware.

- **All the Wood Behind One Arrow.** As there is no alternative, the whole IT industry is committed, which means this time many more people and companies are working on the problem.
Single-Chip Multiprocessors Enable Innovation. Having all processors on the same chip enables inventions that were impractical or uneconomical when built from many chips.

FPGA prototypes shorten hardware/software cycle. Systems like RAMP help us explore the space and build believable prototypes more quickly than conventional hardware prototypes.

Vitality of Open Source Software (OSS). OSS community is a meritocracy, so it’s more likely to embrace technical advances than be hemmed in by legacy code. While OSS existed previously, it was not as important commercially as it is today.

Given the importance of the problem to our shared future, we believe that pessimism is not a sufficient excuse to sit on the sidelines. The sin is not lack of success; it’s lack of effort.

5. RELATED PROJECTS

Although computer science has not solved the parallel challenge, it’s not for lack of trying. There are likely a dozen conferences dedicated to parallelism, including Principles and Practice of Parallel Programming, Parallel Algorithms and Architectures, Parallel and Distributed Processing, and Supercomputing. These conferences are traditionally focus on high performance computing (HPC) and the target hardware is usually large-scale computers with thousands of microprocessors. Similarly, there are many HPC research centers. Rather than review that material, in the limited space available we instead highlight four centers focused on multicore computers to show other recent approaches to the parallel challenge in academia.

The University of Illinois’s Universal Parallel Computing Research Center (UIUC/UPCRC) is focused on making it easy for domain experts to take advantage of parallelism, so the emphasis is more on productivity in specific domains rather than generality or performance [1]. UIUC/UPCRC relies on a compiler technology to find opportunities for parallelism whereas the Par Lab is focusing on autotuning. The center is pursuing deterministic models that allow programmers to reason with sequential semantics for testing, while naturally exposing a parallel performance model for “WYSIWYG” performance. For reactive programs where parallelism is part of the problem, the center is pursuing a shared-nothing approach that leverages actor-like models used in distributed systems. For application domains that allow greater specialization, the center is developing a framework to generate domain-specific environments that either hide concurrency or expose only specialized forms of it to the end-user, while exploiting domain-specific optimizations and performance measures. Initial applications and domains include tele-immersion via “Virtual Teleportation” (multimedia), dynamic real-time virtual environments (computer graphics), learning by reading and authoring assistance (natural language processing).

Stanford’s Pervasive Parallelism Laboratory (PPL) follows an application-driven approach towards parallel computing that extends from programming models down to hardware architecture. The key technical concepts for PPL are domain-specific languages in order to increase programmers’ productivity and a common parallel runtime environment that combines dynamic and static approaches for concurrency and locality management. There are domain specific languages for Artificial Intelligence and robotics, business data analysis, and virtual worlds and gaming. The experimental platform for PPL is the FARM system that combines commercial processors with FPGAs in the memory fabric.

Georgia Tech’s STI Center of Competence for the Cell Broadband Engine Processor focuses on a single multicore computer, as its name suggests. Researchers explore versions of programs on Cell such as image compression [6] and financial modeling [2]. The center also sponsors workshops and provides remote access to Cell hardware.

Rice University’s Habanero Multicore Software Project is developing languages, compilers, managed runtimes, concurrency libraries, and tools that support portable parallel abstractions with high productivity and high performance for multicores. Specific examples include parallel language extensions [24] and optimized synchronization primitives [25].

6. CONCLUSIONS

This paper gives a general view of the parallel landscape, and suggests the goal is making parallel computing productive, efficient, correct, portable, and scalable. We point out the importance of finding new compelling applications and the advantages of manycore and heterogeneous hardware. We also describe the research of the Berkeley Par Lab. While it will take years to learn which of our ideas work well, we provide it now to give one concrete example of a coordinated attack on the problem.

Unlike the traditional approach of making hardware king, the Par Lab is application driven. We are working with domain experts to create compelling applications in music, image and speech recognition, personalized medicine, and parallel browsers.

The software span that connects applications to hardware relies on parallel software architectures more than parallel programming languages. Instead of traditional optimizing compilers, we depend on autotuners, which use a combination of empirical search and performance modeling to create highly optimized libraries tailored to specific machines. By splitting the software stack into a productivity layer and an efficiency layer and targeting them to domain experts and programming experts respectively, we hope to bring parallel computing to all programmers while keeping domain experts productive and allowing expert programmers to get maximum efficiency. Our approach to correctness relies on verification where possible and then uses the same tools to reduce the amount of testing where verification is not possible.

The hardware tower of the Par Lab serves the software span and application tower. Examples include supporting OS partitioning, explicitly controlling the memory hierarchy, accurately counting performance and energy, and an intuitive, multicore performance model. We also plan to try to scrape off the barnacles that have accumulated on the hardware/software stack over the years.

In conclusion, this parallel challenge offers the research community an opportunity to help IT remain a growth industry, to sustain the parts of the economy that depend on the continuously improvement in cost-performance of IT, and a once-in-a-career chance to re-invent the whole software/hardware stack. Although there are reasons for optimism, the difficulty of the challenge is demonstrated by the numerous parallel failures of the past.

When the upsides and downsides are combined together, we view this research challenge as the most significant in the last 50 years. We hope many more will join this quest to build a parallel bridge.
7. ACKNOWLEDGMENTS

This research was sponsored by the Universal Parallel Computing Research Center, which is funded by Intel and Microsoft (Award # 20080469) and by matching funds from U.C. Discovery (Award #DIG07-10227). We wish to thank colleagues in the Par Lab and the Lawrence Berkeley National Labs collaborations that shaped these ideas.

8. REFERENCES


Categories and Subject Descriptors
C.1.2 [Processor Architectures] Multiple Data Stream Architectures (Multiprocessors), D.1.3 [Programming Techniques]: Concurrent Programming, D.3.2 [Programming Languages]: Concurrent, distributed, and parallel languages

General Terms
Algorithms, Design, Measurement, Performance, Languages, Verification

Keywords
Parallel Computer, Multicore Computer, Multiprocessor, Parallel language