Phil Colella's Seven Dwarfs in Parallel Computing

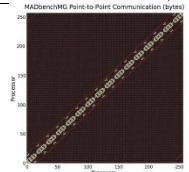
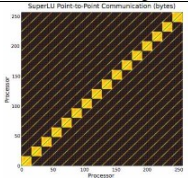
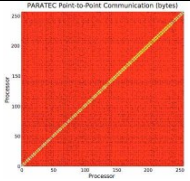
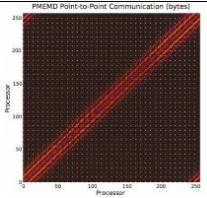
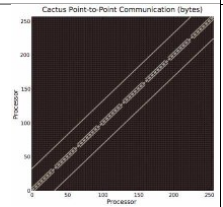
Dwarf	*Description*	*Communication Pattern (Figure axes show processors 1 to 256, with black meaning no communication)*	*NAS Benchmark / Example HW*
1. Dense Linear Algebra (e.g., BLAS [Blackford et al 2002], ScaLAPACK [Blackford et al 1996], or MATLAB [MathWorks 2006])	Data are dense matrices or vectors. (BLAS Level 1 = vector-vector; Level 2 = matrix-vector; and Level 3 = matrix-matrix.) Generally, such applications use unit-stride memory accesses to read data from rows, and strided accesses to read data from columns.	 The communication pattern of MadBench, which makes heavy use of ScaLAPACK for parallel dense linear algebra, is typical of a much broader class of numerical algorithms	Block Triadiagonal Matrix, Lower Upper Symmetric Gauss-Seidel / Vector computers, Array computers
2. Sparse Linear Algebra (e.g., SpMV, OSKI [OSKI 2006], or SuperLU [Demmel et al 1999])	Data sets include many zero values. Data is usually stored in compressed matrices to reduce the storage and bandwidth requirements to access all of the nonzero values. One example is block compressed sparse row (BCSR). Because of the compressed formats, data is generally accessed with indexed loads and stores.	 SuperLU (communication pattern pictured above) uses the BCSR method for implementing sparse LU factorization.	Conjugate Gradient / Vector computers with gather/scatter
3. Spectral Methods (e.g., FFT [Cooley and Tukey 1965])	Data are in the frequency domain, as opposed to time or spatial domains. Typically, spectral methods use multiple butterfly stages, which combine multiply-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others.	 PARATEC: The 3D FFT requires an all-to-all communication to implement a 3D transpose, which requires communication between every link. The diagonal stripe describes BLAS-3 dominated linear-algebra step required for orthogonalization.	Fourier Transform / DSPs, Zalink PDSP [Zarlink 2006]
4. N-Body Methods (e.g., Barnes-Hut [Barnes and Hut 1986], Fast Multipole Method [Greengard and Rokhlin 1987])	Depends on interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others, leading to an $O(N^2)$ calculation, and hierarchical particle methods, which combine forces or potentials from multiple points to reduce the computational complexity to O(N log N) or O(N).	 PMEMD's communication pattern is that of a particle mesh Ewald calculation.	(no benchmark) / GRAPE [Tokyo 2006], MD-GRAPE [IBM 2006]
5. Structured Grids (e.g., Cactus [Goodale et al 2003] or Lattice-Boltzmann Magneto-hydrodynamics [LBMHD 2005])	Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between 2 versions of the grid. The grid may be subdivided into finer grids in areas of interest ("Adaptive Mesh Refinement"); and the transition between granularities may happen dynamically.	 Communication pattern for Cactus, a PDE solver using 7-point stencil on 3D block-structured grids.	Multi-Grid, Scalar Penta-diagonal / QCDOC [Edinburg 2006], BlueGeneL
6. Unstructured Grids (e.g., ABAQUS [ABAQUS 2006] or FIDAP [FLUENT 2006])	An irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighboring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points, and then loading values from those neighboring points.		Unstructured Adaptive / Vector computers with gather/scatter, Tera Multi Threaded Architecture [Berry et al 2006]
7. Monte Carlo (e.g., Quantum Monte Carlo [Aspuru-Guzik et al 2005])	Calculations depend on statistical results of repeated random trials. Considered embarrassingly parallel.	Communication is typically not dominant in Monte Carlo methods.	Embarrassingly Parallel / NSF Teragrid

Figure 3. Seven Dwarfs, their descriptions, corresponding NAS benchmarks, and example computers.

DOI:10.1145/1562764.1562783

Writing programs that scale with increasing numbers of cores should be as easy as writing programs for sequential computers.

BY KRSTE ASANOVIC, RASTISLAV BODIK, JAMES DEMMEL, TONY KEAVENY, KURT KEUTZER, JOHN KUBIATOWICZ, NELSON MORGAN, DAVID PATTERSON, KOUSHIK SEN, JOHN WAWRZYNEK, DAVID WESSEL, AND KATHERINE YELICK

A View of the Parallel Computing Landscape

INDUSTRY NEEDS HELP from the research community to succeed in its recent dramatic shift to parallel computing. Failure could jeopardize both the IT industry and the portions of the economy that depend on rapidly improving information technology. Here, we review the issues and, as an example, describe an integrated approach we're developing at the Parallel Computing Laboratory, or Par Lab, to tackle the parallel challenge.

Over the past 60 years, the IT industry has improved the cost-performance of sequential computing by about 100 billion times overall.[20] For most of the past 20 years, architects have 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 increased 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 terms of transistors and power (such as multiple instruction issue, deep pipelines, out-of-order execution, speculative execution, and prefetching) but that increased performance while preserving the sequential programming model.

The contract worked fine until we hit the power limit a chip is able to dissipate. Figure 1 reflects this abrupt change, plotting the projected microprocessor clock rates of the International Technology Roadmap for Semiconductors in 2005 and then again just two years later.[16] The 2005 prediction was that clock rates should have exceeded 10GHz in 2008, topping 15GHz in 2010. Note that Intel products are today 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 the only viable option was to replace the single power-inefficient processor with many efficient processors on the same chip. The whole microprocessor industry thus declared that its future was in parallel computing, with increasing numbers of processors, or cores, each technology generation 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 and is actually a retreat from the more difficult task of building power-efficient, high-clock-rate, single-core chips.[5]

Many startups have sold parallel computers over the years, but all failed, as programmers accustomed to continuous improvement in sequential performance saw little need to explore parallelism. Convex, Encore, Floating Point Systems, Inmos, Kendall Square

ILLUSTRATION BY LEONELLO CALVETTI

Research, MasPar, nCUBE, Sequent, Silicon Graphics, and Thinking Machines are just the best-known members of the Dead Parallel Computer Society. Given this sad history, multicore pessimism abounds. Quoting computing pioneer John Hennessy, President of Stanford University:

"...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 the hull of an old ship.

Here, we lay out one view of the opportunities, then, as an example, describe in more depth the approach of the Berkeley Parallel Computing Lab, or Par Lab, updating two long technical reports[4,5] that include more detail. Our goal is to recruit more parallel revolutionaries.

Parallel Bridge

The bridge in Figure 2 represents an analogy connecting computer 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 in between is software. We use the bridge analogy throughout this article. The aggressive goal of the parallel revolution is to make it as easy to write programs that are as efficient, portable, and correct (and that scale as the number of cores per microprocessor increases biennially) as it has been to write programs for sequential computers. Moreover, we can fail overall if we fail to deliver even one of these "parallel virtues." For example, if parallel programming is unproductive, this weakness will delay and reduce the number of programs that are able to exploit new multicore architectures.

Hardware tower. The power wall forces the change in the traditional programming model, but the question for parallel researchers is what kind of computing architecture should take its place. There is a technology sweet spot around a pipelined processor of five-to-eight stages that is most efficient in terms of performance per joule and silicon area.[5] Using simple cores means there is room for hundreds of them on the same chip. 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. Just as it's easy to add spares to mask manufacturing defects, "manycore" computers can also have higher yield.

One example of a manycore computer is from the world of network processors, which has seen a great deal of innovation recently due to the growth of the networking market. The best-designed network processor is arguably the Cisco Silicon Packet Processor, also known as Metro, which has 188 five-stage RISC cores, plus four spares to help yield and dissipate 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 speedup 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 other features. Applying Amdahl's Law, here are the speedups (relative to one thin core) of 100 thin cores and 90 thin cores for the parallel code plus one fat core for the sequential code:

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

$$\text{Speedup}_{91} = 1 / (0.1/2 + 0.9/90) = 16.7 \text{ times faster}$$

In this example of manycore processor speedup, a fat core needing 10 times as many resources would be more effective than the 10 thin cores it replaces.[5,15]

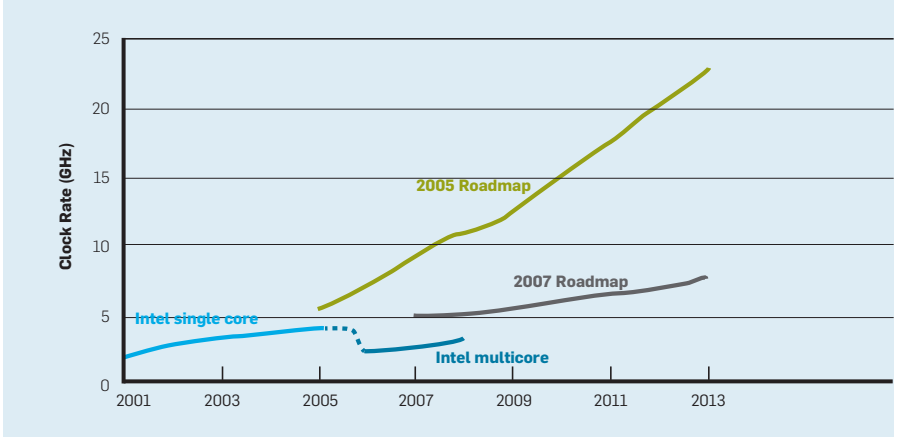
One notable challenge for the hardware tower is that it takes four to five years to design and build chips and port software to evaluate them. Given this lengthy cycle, how could researchers innovate more quickly?

Software span. Software is the main problem in bridging the gap between users and the parallel IT industry. Hence, the long distance of the span in Figure 2 reflects the daunting magnitude of the software challenge.

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

A second challenge is that two critical pieces of system software—compilers and operating systems—have grown large and unwieldy and hence

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



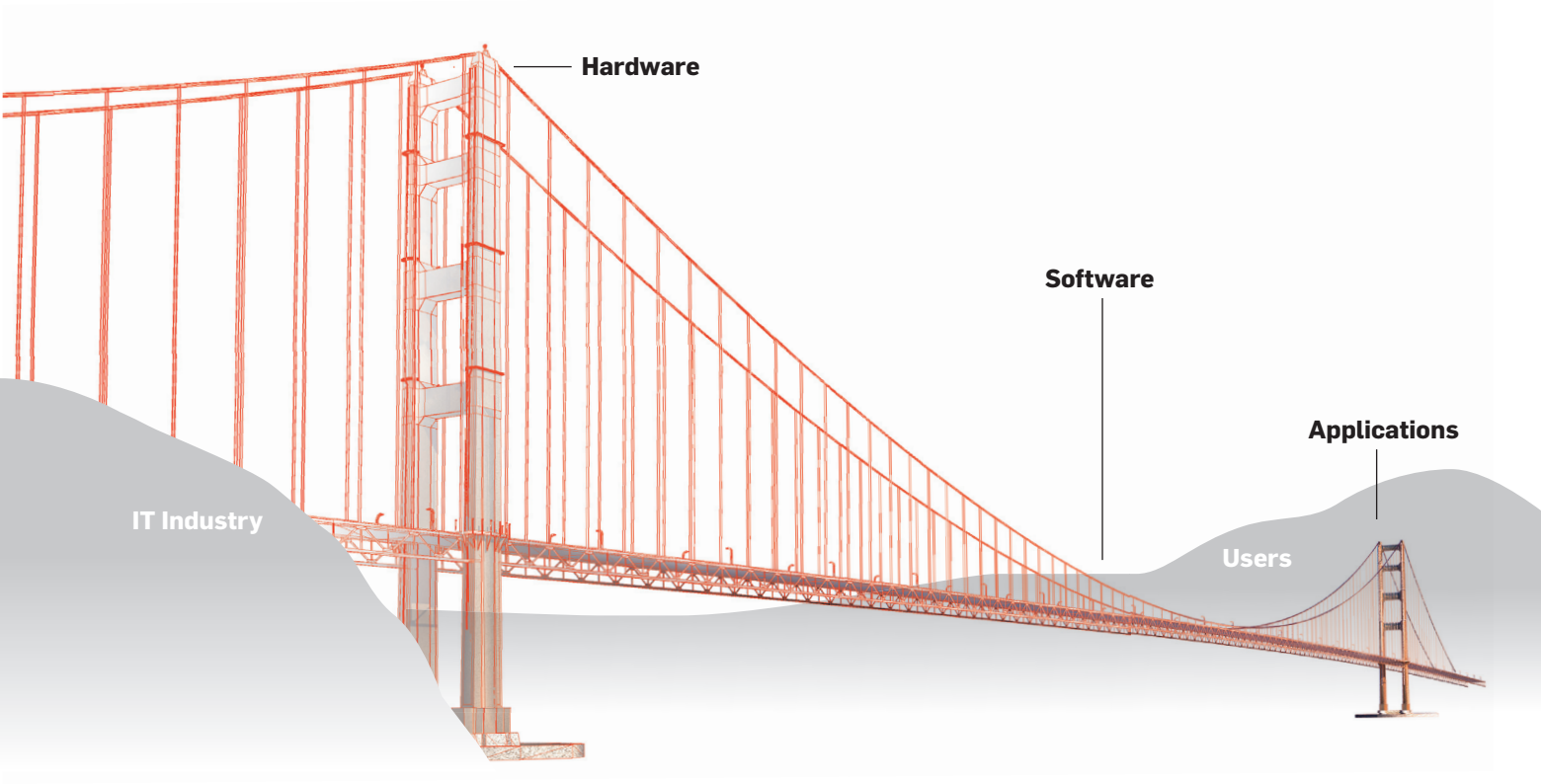


Figure 2. Bridge analogy connecting users to a parallel IT industry, inspired by the view of the Golden Gate Bridge from Berkeley, CA.

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 so glacially?

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

Applications tower. The goal of research into parallel computing should be to find compelling applications that thirst for more computing than is currently available and absorb biennially increasing number of cores for the next decade or two. Success does not require improvement in the performance of all legacy software. Rather, we need to create compelling applications that effectively utilize the growing number of cores while providing software environments that ensure that legacy code still works with acceptable performance.

Note that the notion of "better" is not defined by only average performance; advances could be in, say, worst-case response time, battery life, reliability, or security. To save the IT industry, researchers must demonstrate greater end-user value from an increasing number of cores.

Par Lab

As a concrete example of the parallel landscape, we describe Berkeley's Par Lab project,[a] exploring one of many potential approaches, though we won't know for years which of our ideas will bear fruit. We hope it inspires more researchers to participate, increasing the chance of finding a solution before it's too late for the IT industry.

Given a five-year project, we project the state of the field in five to 10 years, anticipating that IT will be driven to extremes in size due to the increasing popularity of software as a service, or SaaS:

The datacenter is the server. Amazon, Google, Microsoft, and other major IT vendors are racing to construct buildings with 50,000 or more servers to run SaaS, inspiring the new catchphrase "cloud computing."[b] They have also begun renting thousands of machines by the hour to enable smaller companies to benefit from cloud computing. We expect these trends to accelerate; and

The mobile device (laptops and handhelds) is the client. In 2007, Hewlett-Packard, the largest maker of PCs, shipped more laptops than desktops. Millions of cellphones are shipped each day with ever-increasing functionality, a trend we expect to accelerate as well.

Surprisingly, these extremes in computing share many characteristics. Both concern power and energy—the datacenter due to the cost of power and cooling and the mobile client due to battery life. Both concern cost—the datacenter because server cost is replicated 50,000 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 concern about protection and security.

a In March 2007, Intel and Microsoft invited 25 universities to propose five-year centers for parallel computing research; the Berkeley and Illinois efforts were ranked first and second.

b See Ambrust, M. et al. *Above the Clouds: A Berkeley View of Cloud Computing.* University of California, Berkeley, Technical Report EECS-2009-28.

Many datacenter applications have ample parallelism across independent users, so the Par Lab focuses on parallelizing applications for clients. The multicore and manycore chips in the datacenter stand to benefit from the same tools and techniques developed for similar chips in mobile clients.

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

Par Lab application tower. An unfortunate computer science tradition is we build research prototypes, then wonder why applications people don't use them. In the Par Lab, we instead selected applications up-front to drive research and provide concrete goals and metrics to evaluate progress. We selected each application based on five criteria: compelling in terms of likely market or social impact, with short-term feasibility and longer-term potential; requiring significant speedup or smaller, more efficient platform to work as intended; covering the possible platforms and markets likely to dominate usage; enabling technology for other applications; and involvement of a local committed expert application partner to help design, use, and evaluate our technology.

Here are the five initial applications we're developing:

Music/hearing. High-performance signal processing will permit: concert-quality sound-delivery systems for home sound systems and conference calls; composition and gesture-driven live-performance systems; and 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, say, a real-time meeting transcriber with rewind and search. Depending on acoustic conditions, current transcribers can generate many errors;

Content-based image retrieval. Consumer-image databases are growing so dramatically they require automated search instead of manual labeling. Low error rates require processing very high dimensional feature spaces. Current image classifiers are too slow to deliver adequate response times;

Intraoperative risk assessment for stroke patients. Advanced physiological blood-flow modeling based on computational analysis of 3D medical images of a patient's cerebral vasculature enables "virtual stress testing" to risk-stratify stroke victims intraoperatively. Patients thus identified at low risk of complications can then be treated to mitigate the effects of the stroke. This technology will ultimately lower complication rates in treating stroke victims, improve quality of life, reduce medical care expenditures, and save lives; and

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 are pursuing an actor language with implicit parallelism. Such a language may be accessible to Web programmers while allowing them to extract the parallelism in the browser's JIT compiler, thereby turning all Web-site developers unknowingly into parallel programmers.

Application-domain experts are first-class members of the Par Lab project. Rather than try to answer design questions abstractly, we ask our experts what they prefer in each case. Project success is judged by the user experience with the collective applications on our hardware-software prototypes. If successful, we imagine building on these five applications to create other applications that are even more compelling, as in the following two examples:

Name Whisperer. Imagine that your mobile client peeking out of your shirt pocket is able to recognize the person walking toward you to shake your hand. It would search a personal image database, then whisper in your ear, "This man is John Smith. He got an A– from you in CS152 in 1993"; and

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 visualize what you're likely to look like in six months at this rate and what you'd look like if you ate less or exercised more.

Par Lab software span. Software is the major effort of the project, and we're taking a different path from previous parallel projects, emphasizing software architecture, autotuning, and separate support for productivity vs. performance programming.

Architecting parallel software with design patterns, not just parallel programming languages. Our situation is similar to that found in other engineering disciplines where a new challenge emerges that requires a top-to-bottom rethinking of the entire engineering process; for example, in civil architecture, Filippo Brunelleschi's solution in 1418 for how to construct the dome for the Cathedral of Florence required innovations in tools and building techniques, as well as rethinking the whole process of developing an architecture. All computer science faces a similar challenge; 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 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 can't discover new algorithms to exploit concurrency. Programmers in high-performance computing have shown that explicit technologies (such as MPI and OpenMP) can be made to work but too often require heroic effort untenable for most commercial software vendors.

To engineer high-quality parallel software, we plan to rearchitect the software through a "design pattern language." As explored in his 1977 book, civil architect Christopher Alexander wrote that "design patterns" describe time-tested solutions to recurring problems within a well-defined context.[3] An example is Alexander's "family of entrances" pattern, addressing how to simplify comprehension of multiple entrances for a first-time visitor to a site. He defined 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, more associative way of thinking about a problem. A pattern language does not impose a rigid methodology; rather, it fosters creative problem

solving by providing a common vocabulary to capture the problems encountered during design and identify potential solutions from among families of proven designs.

The observation that design patterns and pattern languages might be useful for software design is not new. An example is Gamma et al.'s 1994 book *Design Patterns*, which outlined patterns useful for object-oriented programming.[12] In building our own pattern language, we found Shaw's and Garlan's report,[23] which described a variety of architectural styles useful for organizing software, to be very effective. That these architectural styles may also be viewed as design patterns was noted earlier by Buschmann in his 1996 book *Pattern-Oriented Software Architecture*.[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 structural design patterns.

These patterns define the structure of a program but do not indicate what is actually computed. To address this blind spot, another key part of our pattern language is the set of "dwarfs" of the Berkeley View reports[4,5] (see Figure 3). Dwarfs are best understood as computational patterns providing the computational interior of the structural patterns discussed earlier. 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 useful computational patterns we began with those compiled by Phillip Collela of Lawrence Berkeley National Laboratory of the "seven dwarfs of high-performance computing." Then, in 2006 and 2007 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 the applications we described earlier. Figure 3 shows the results of our pattern mining.

If researchers meet the parallel challenge, the future of IT is rosy. If they don't, it's not.

Computational and structural patterns can be hierarchically composed to define an application's high-level software architecture, 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 2004 book *Patterns for Parallel Programming*[18] was the first such attempt to systematize parallel programming using a complete pattern language. We combine the structural and computational patterns mentioned earlier in our pattern language to literally sit on top of the algorithmic structures and implementation structures in the pattern language in Mattson's book. The resulting pattern language is still under development but is already employed by the Par Lab 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 reason about a software project and develop an architecture but are not themselves implementation mechanisms for producing code.

Split productivity and efficiency layers, not just a single general-purpose layer. A key Par Lab research objective is to enable programmers to easily write programs that run as efficiently on manycore systems as on sequential ones. Productivity, efficiency, and correctness are inextricably linked and must be addressed together. These objectives cannot be accomplished with a single-point solution (such as a universal language). In our approach, productivity is addressed in a productivity layer that uses a common composition and coordination language to glue together the libraries and programming frameworks produced by the efficiency-layer programmer. Efficiency is principally handled through an efficiency layer that is targeted for use by expert parallel programmers.

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 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, which is based on the Model-View-Controller pattern.[26] Users 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 that assist programmers with implementation by abstracting difficult portions of the computation and incorporating them into the framework itself. Historically successful parallel frameworks encode the collective experience of the programming community's solutions to recurring problems. Basing frameworks on pervasive design patterns will help make parallel frameworks broadly applicable.

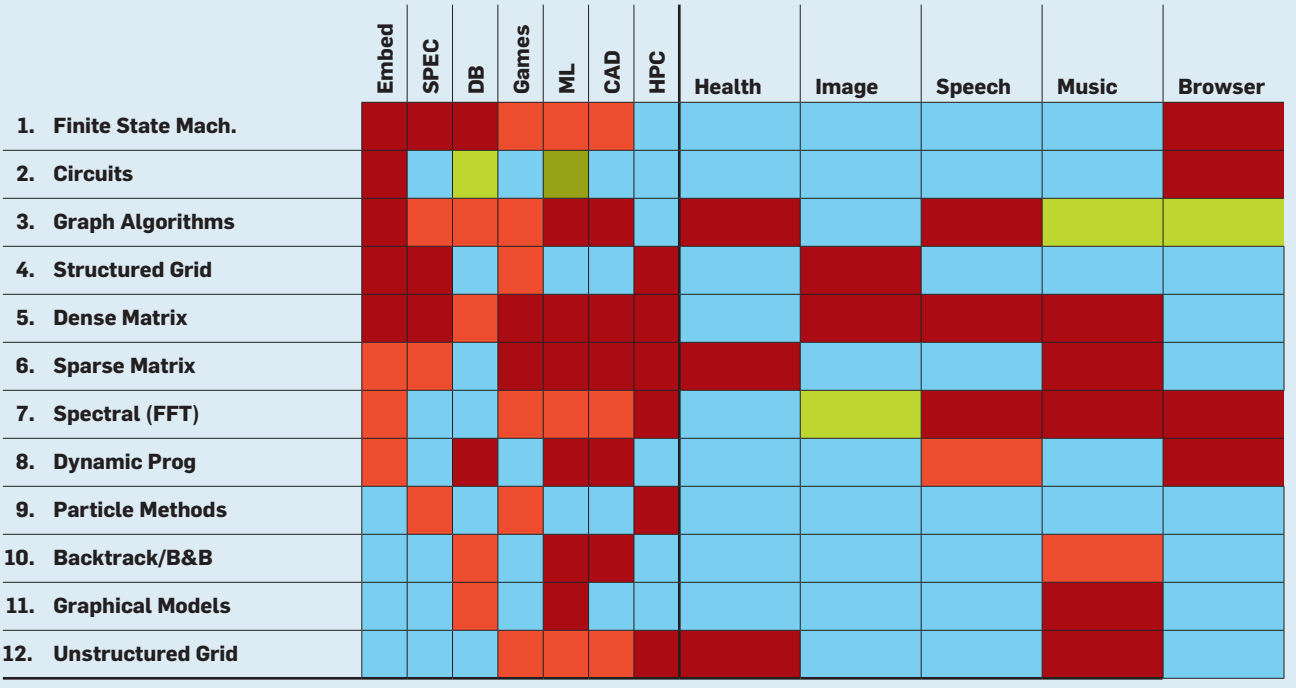
Productivity-layer programmers will compose libraries and programming frameworks into applications with the help of a composition and coordination language.[13] The language will be implicitly parallel; that is, its composition will have serial semantics, meaning the composed programs will be safe (such as race-free) and virtualized with respect to processor resources. It will document and check interface restrictions to avoid concurrency bugs resulting from incorrect composition, as in, say, instantiating a framework with a stateful function when a stateless one is required. Finally, it will support definition of domain-specific abstractions for constructing frameworks for specific applications, offering a programming experience similar to MATLAB and SQL.

Parallel 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 the layer. Unfortunately, existing multicore systems do not offer a common low-level programming model for parallel code. We are thus defining a thin portability layer that runs efficiently across single-socket platforms and includes 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 are defining an API based on the idea of logically partitioned shared memory, inspired by our experience with Unified Parallel C,[27] which partitions memory among processors but not (currently) between on- and off-chip.

We may implement this efficiency language 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). The API will be implemented on some existing multicore and manycore platforms and on our own emulated manycore design.

To engineer parallel software, programmers must be able to start with effective software architectures, and the software engineer would describe the solution to a problem in terms of a design pattern language. Based on this language, the Par Lab is creating a fam-

Figure 3. The color of a cell (for 12 computational patterns in seven general application areas and five Par Lab applications) indicates the presence of that computational pattern in that application; red/high; orange/moderate; green/low; blue/rare.



ily of frameworks to help turn a design into working code. The general-purpose programmer will work largely with the frameworks and stay within what we call the productivity 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 will help general-purpose programmers create parallel software without having to master the low-level details of parallel programming.

Generating code with search-based autotuners, not compilers. Compilers that automatically parallelize sequential code may have great commercial value as computers go from one to two to four cores, though as described earlier, history suggests they will be unable to scale from 32 to 64 to 128 cores. Compiling will be even more difficult, as the switch to multicore means microprocessors are becoming more diverse, since conventional wisdom is not yet established for multicore architectures. For example, the table here 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 numbers 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] optimizing 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, searching may take hours to complete on the target platform. However, search is performed only once, 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 variety of machines. In many cases, the autotuned code is faster 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. Unlike libraries, autotuners also allow tuning to the particular problem size. Autotuners also preserve clarity and support portability by reducing the temptation to mangle the source code to improve performance for a particular computer.

Autotuning also helps with production of parallel code. However, parallel architectures introduce many new optimization parameters; so far, there are few successful autotuners for parallel codes. 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 for traditional serial hardware.

The table lists the results of autotuning on three multicores for three kernels related to the dwarfs' sparse matrix, stencil for PDEs, and structured grids[9,30,31] mentioned earlier. This autotuned code is the fastest known for these kernels for all three computers. Performance increased by factors of two to four over standard code, much better than you would expect from an optimizing compiler.

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

Synthesis with sketching. One challenge for autotuning is how to produce the high-performance implementations explored by the search. One approach is to synthesize these complex programs. In doing so, we rely on the search for performance tuning, as well as for programmer productivity. To address the main challenge of traditional synthesis—the need for experts to communicate their insight with a formal domain theory—we allow that insight to be communicated directly by programmers who write an incomplete program, or "sketch." In it, they provide an algorithmic skeleton, and the synthesizer supplies the low-level mechanics by filling in 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, our system searches a space of candidate programs with constraint solving. 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 additional problems (such as parsing).

Verification and testing, not one or the other. Correctness is addressed differently at the two layers. The productivity layer is free from concurrency problems because the parallelism models are restricted, and the restrictions are enforced. The efficiency-layer code is checked automatically for subtle concurrency errors.

A key challenge in verification is obtaining specifications for programs 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 parallel frameworks and parallel libraries). To simplify specification, we use executable sequential programs with the same behavior as a parallel component, augmented with atomicity constraints on a task,[21] 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 involving large modules; however, most find it convenient to specify local properties of programs using assert statements and type annotations. Local assertions and type annotations are often generated from a program's implicit correctness requirements (such as data race, deadlock freedom, and memory safety). The system propagates implications of these local assertions to the module boundaries through a combination of static verification and directed automated unit testing. These implications create serial contracts that specify how the modules (such as frameworks) are used correctly. When the contracts for the parallel modules are in place, programmers use static program verification to

Autotuned performance in GFLOPS/s on three kernels for dual-socket systems.

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

check if the client code composed with the contracts is correct.

Static program analysis in the presence of pointers and heap memory falsely reports many errors that cannot really occur. For restricted parallelism models with global synchronization, this analysis becomes more tractable, and a recently introduced technique called "directed automated testing," or concolic unit testing, has shown promise for improving software quality through automated test generation using a combination of static and dynamic analyses.[21] The Par Lab combines directed testing with model-checking algorithms to unit-test parallel frameworks and libraries composed with serial contracts. Such techniques enable programmers 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 testing techniques will provide deterministic replay and debugging capabilities at low cost. We will also 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.

Parallelism for energy efficiency. While the earlier computer classes—desktops and laptops—reused the software of their own earlier ancestors, the energy efficiency for handheld operation may need to come from data parallelism in tasks that are currently executed sequentially, possibly from three sources:

Efficiency. Completing a task on slow parallel cores will be more efficient than completing it in the same time sequentially on one fast core;

Energy amortization. Preferring data-parallel algorithms over other styles of parallelism, as SIMD and vector computers amortize the energy expended on instruction delivery; and

Energy savings. Message-passing programs may be able to save the energy used by cache coherence.

We apply these principles in our work on parallel Web browsers. In algorithm design, we observe that to save energy with parallelization, parallel algorithms must be close to "work efficient," that is, they should perform no more total work than a sequential algorithm, or else parallelization is counterproductive. The same argument applies to 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 identifying parallelism in future Web-browser applications, attempting to implement them with efficient message passing.

Space-time partitioning for deconstructed operating systems. Space-time partitioning is crucial for manycore client operating systems. A spatial partition (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 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 programmer-level scheduling in a partition.

The presence of space-time partitioning leads to restructuring 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 (such as file systems) run within their own partitions. Partitions are lightweight and can be resized or suspended with similar overheads to a process-context swap.

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 some way. Within a partition, our approach has 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 for only the coarse-grain scheduling and assignment of resources to partitions and implementation of secure restricted communications among partitions. The Tessellation kernel is much thinner than traditional kernels or even hypervisors. It avoids many of the performance issues associated 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.

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

Here are four examples of this kind of help that illustrate 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 among these components, providing quality-of-service guarantees. The resulting performance predictability improves parallel program performance, simplifies code autotuning and dynamic load balancing, supports real-time applications, and simplifies scheduling.

To save the IT industry, researchers must demonstrate greater end-user value from an increasing number of cores.

Optional explicit control of the memory hierarchy. Caches were invented so 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 backward situation is especially apparent for hardware prefetchers when programmers try to create a particular pattern that will invoke good prefetching. Our approach aims to allow programmers to quickly turn a cache into an explicitly managed local store and the prefetch engines into explicitly controlled Direct Memory Access engines. To make it easy for programmers to port software to our architecture, we also support a traditional memory hierarchy. The low-overhead mechanism we use allows programs to be composed of methods that rely on local stores and methods that rely on memory hierarchies.

Accurate, complete counters of performance and energy. Sadly, performance counters on current single-core computers often miss important measurements (such as prefetched data) or are unique to a computer and only understandable by the machine's designers. We will include performance enhancements in the Par Lab architecture only if they have counters to measure them accurately and coherently. Since energy is as important as performance, we also include energy counters so 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 will lead to a standard for performance counters so schedulers and software development kits can count on them on any multicore.

Intuitive performance model. The multicore diversity mentioned earlier exacerbates the already difficult jobs performed by programmers, compiler writers, and architects. Hence, we developed an easy-to-understand visual

model with built-in performance guidelines to identify bottlenecks in the dozen dwarfs in Figure 3.[29] The Roofline model plots computational and memory-bandwidth limits, then determines the 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 the optimizations that might be useful for improving performance. One goal of the performances counters should be to provide everything needed to automatically create Roofiline models.

A notable challenge from our earlier description of the hardware tower is how to rapidly innovate at the hardware/software interface, when it can take four to five years to build chips and run programs needed to evaluate them. Given the capacity of field-programmable gate arrays (FPGAs), 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 over years. We will leverage the Research Accelerator for Multiple Processors (RAMP) Project (http://ramp.eecs.berkeley.edu/) to build flexible prototypes fast enough to run full software stacks—including new operating systems and our five compelling applications—to enable rapid architecture innovation using future prototype software, rather than past benchmarks.[28]

Reasons for Optimism

Given the history of parallel computing, it's easy to be pessimistic about our chances. The good news is that there are plausible reasons researchers could succeed this time:

No killer microprocessor. Unlike in the past, no one is building the faster serial microprocessor; programmers needing more performance have no option other than parallel hardware;

New measures of success. Rather than the traditional goal of linear speedup for all software as the number of processors increases, success can reflect improved responsiveness or MIPS/Joule for a few new parallel killer apps;

All the wood behind one arrow. As there is no alternative, the whole IT industry is committed, meaning many more people and companies are working on the problem;

Manycore synergy with cloud computing. SaaS applications in data centers with millions of users are naturally parallel and thus aligned with manycore, even if clients apps are not;

Vitality of open source software. The OSS community is a meritocracy, so it's likely to embrace technical advances rather than be limited by legacy code. Though OSS has existed for years, it is more important commercially today than it was;

Single-chip multiprocessors enable innovation. Having all processors on the same chip enables inventions that were impractical or uneconomical when spread across many chips; and

FPGA prototypes shorten the hardware/software cycle. Systems like RAMP help researchers explore designs of easy-to-program manycore architectures and build prototypes more quickly than they ever could with conventional hardware prototypes.

Given the importance of the challenges to our shared future in the IT industry, pessimism is not a sufficient excuse to sit on the sidelines. The sin is not lack of success but lack of effort.

Related Projects

Computer science hasn't solved the parallel challenge though not because it hasn't tried. There could be a dozen conferences dedicated to parallelism, including Principles and Practice of Parallel Programming, Parallel Algorithms and Architectures, Parallel and Distributed Processing, and Supercomputing. All traditionally focus on high-performance computing; the target hardware is usually large-scale computers with thousands of microprocessors. Similarly, there are many high-performance computing research centers. Rather than review this material, here we highlight four centers focused on multicore computers and their approaches to the parallel challenge in academia:

Illinois. The Universal Parallel Computing Research Center (http://www.upcrc.illinois.edu/) at the University of Illinois focuses on making it easy for domain experts to take advantage of parallelism, so the emphasis is more on productivity in specific domains than on generality or performance.[1] It relies on advancing compiler technology to find opportunities for parallelism, whereas the Par Lab focuses 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, it is pursuing a shared-nothing approach that leverages actor-like models used in distributed systems. For application domains that allow greater specialization, it is developing a framework to generate domain-specific environments that either hide concurrency or expose only specialized forms of concurrency to the end user while exploiting domain-specific optimizations and performance measures. Initial applications and domains include teleimmersion via "virtual teleportation" (multimedia), dynamic real-time virtual environments (computer graphics), learning by reading, and authoring assistance (natural language processing).

Stanford. The Pervasive Parallelism Laboratory (http://ppl.stanford.edu/wiki/index.php/Pervasive_Parallelism_Laboratory) at Stanford University takes an application-driven approach toward parallel computing that extends from programming models down to hardware architecture. The key technical concepts are domain-specific languages for increasing programmer productivity and a common parallel runtime environment combining 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 is the Flexible Architecture Research Machine, or FARM, system, combining commercial processors with FPGAs in the memory fabric.

Georgia Tech. The Sony, Toshiba, IBM Center of Competence for the Cell Broadband Engine Processor (http://sti.cc.gatech.edu/) at Georgia Tech focuses on a single multicore computer, as its name suggests. Researchers explore versions of programs on Cell, including image compression[6] and financial modeling.[2] The Center also sponsors workshops and provides remote access to Cell hardware.

Rice University. The Habanero Multicore Software Project (http://habanero.rice.edu/Habanero_Home.html) at

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

Conclusion

We've provided a general view of the parallel landscape, suggesting that the goal of computer science should be making parallel computing productive, efficient, correct, portable, and scalable. We highlighted the importance of finding new compelling applications and the advantages of manycore and heterogeneous hardware. We also described the research of the Berkeley Par Lab. While it will take years to learn which of our ideas work well, we share it here as a concrete example of a coordinated attack on the problem.

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

The software span connecting applications to hardware relies more on parallel software architectures than on parallel programming languages. Instead of traditional optimizing compilers, we depend on autotuners, using 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 at 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 achieve maximum efficiency. Our approach to correctness relies on verification where possible, 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 of such service include support for OS partitioning, explicit control for the memory hierarchy, accurate measurement for 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.

This parallel challenge offers the worldwide research community an opportunity to help IT remain a growth industry, sustain the parts of the worldwide economy that depend on the continuous improvement in IT cost-performance, and take a once-in-a-career chance to reinvent the whole software/hardware stack. Though there are reasons for optimism, the difficulty of the challenge is reflected in the numerous parallel failures of the past.

Combining upside and downside, this research challenge represents the most significant of all IT challenges over the past 50 years. We hope many more innovators will join this quest to build a parallel bridge.

Acknowledgments

This research is sponsored by the Universal Parallel Computing Research Center, which is funded by Intel and Microsoft (Award # 20080469) and by matching funds from U.C. Discovery (Award #DIG07-10227). Additional support comes from the Par Lab Affiliate companies: National Instruments, NEC, Nokia, NVIDIA, and Samsung. We wish to thank our colleagues in the Par Lab and the Lawrence Berkeley National Laboratory collaborations who shaped these ideas.

References

1. Adve, S. et al. *Parallel Computing Research at Illinois: The UPCRC Agenda.* White Paper. University of Illinois, Urbana-Champaign, IL, Nov. 2008.
2. Agarwal, V., Liu, L.-K., and Bader, D. Financial modeling on the Cell broadband engine. In *Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium* (Miami, FL, Apr. 14–18, 2008).
3. Alexander, C. et al. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1997.
4. Asanovic, K. et al. *The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View.* UCB/EECS-2008-23, University of California, Berkeley, Mar. 21, 2008.
5. Asanovic, K. et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* UCB/EECS-2006-183, University of California, Berkeley, Dec. 18, 2006.
6. Bader, D.A. and Patel, S. High-performance MPEG-2 software decoder on the Cell broadband engine. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium* (Miami, FL, Apr. 14–18, 2008).
7. Buschmann, F. et al. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, Inc., New York, 1996.
8. Clarke, D.G. et al. Ownership types for flexible alias protection. In *Proceedings of the OOPSLA Conference* (Vancouver, BC, Canada, 1998), 48–64.
9. Datta, K. et al. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proceedings of the ACM/IEEE Supercomputing (SC) 2008 Conference* (Austin, TX, Nov. 15–21). IEEE Press, Piscataway, NJ, 2008.
10. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R., and Yelick, K. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation 93*, 2 (Feb. 2005), 293–312.
11. Engler, D.R. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Cooper Mountain, CO, Dec. 3–6, 1995), 251–266.
12. Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Reading, MA, 1994.
13. Gelernter, D. and Carriero, N. Coordination languages and their significance. *Commun. ACM 35*, 2 (Feb. 1992), 97–107.
14. Henzinger, T.A. et al. Permissive interfaces. In *Proceedings of the 10th European Software Engineering Conference* (Lisbon, Portugal, Sept. 5–9). ACM Press, New York, 2005, 31–40.
15. Hill, M. and Marty, M. Amdahl's Law in the multicore era. *IEEE Computer 41*, 7 (2008), 33–38.
16. International Technology Roadmap for Semiconductors. Executive Summary, 2005 and 2007; http://public.itrs.net/.
17. Kantowitz, B. and Sorkin, R. *Human Factors: Understanding People-System Relationships.* John Wiley & Sons, Inc., New York, 1983.
18. Mattson, T., Sanders, B., and Massingill, B. *Patterns for Parallel Programming.* Addison-Wesley Professional, Reading, MA, 2004.
19. O'Hanlon, C. A conversation with John Hennessy and David Patterson. *Queue 4*, 10 (Dec. 2005/Jan. 2006), 14–22.
20. Patterson, D. and Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface, Fourth Edition.* Morgan Kaufmann Publishers, Boston, MA, Nov. 2008.
21. Sen, K. and Viswanathan, M. Model checking multithreaded programs with asynchronous atomic methods. In *Proceedings of the 18th International Conference on Computer-Aided Verification* (Seattle, WA, Aug. 17–20, 2006).
22. Sen, K. et al. CUTE: A concolic unit testing engine for C. In *Proceedings of the Fifth Joint Meeting European Software Engineering Conference* (Lisbon, Portugal, Sept. 5–9). ACM Press, New York, 2005, 263–272.
23. Shaw, M. and Garlan, D. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21. CMU Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.
24. Shirako, J., Peixotto, D., Sarkar, V., and Scherer, W. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd ACM International Conference on Supercomputing* (Island of Kos, Greece, June 7–12). ACM Press, New York, 2008, 277–288.
25. Shirako, J., Kasahara, H., and Sarkar, V. Language extensions in support of compiler parallelization. In *Proceedings of the 20th Workshop on Languages and Compilers for Parallel Computing* (Urbana, IL, Oct. 11–13). Springer-Verlag, Berlin, 2007, 78–94.
26. Thomas, D. et al. *Agile Web Development with Rails, Second Edition.* The Pragmatic Bookshelf, Raleigh, NC, 2008.
27. UPC Language Specifications, Version 1.2. Technical Report LBNL-59208. Lawrence Berkeley National Laboratory, Berkeley, CA, 2005.
28. Wawrzynek, J. et al. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro 27*, 2 (Mar. 2007), 46–57.
29. Williams, S., Waterman, A., and Patterson, D. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76.
30. Williams, S. et al. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium* (Miami, FL, Apr. 14–18, 2008).
31. Williams, S. et al. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the Supercomputing (SC07) Conference* (Reno, NV, Nov. 10–16). ACM Press, New York, 2007.

The authors are all affiliated with the Par Lab (http://parlab.eecs.berkeley.edu/) at the University of California, Berkeley.