REVIEW ON PARALLEL COMPUTING BECOME UBIQUITOUS

Mr. Gajanan P. Datir*

Prof. P. A. Tijare*

Abtract –

In this paper, how to make parallel programming succeed given industry's recent shift to multicore computing. Here we are present the future microprocessors will have hundreds of cores and are working on applications, programming environments, and architectures that will meet this challenge and how to achieve the ubiquitous parallel computing. .In this paper briefly surveys the similarities and difference in research review from the Berkeley, Illiniois and Stanford.

Keyword: Parlab, UPSRC, SEJITS, AST, SIMD, DSL, CPR

* Dept. of Computer Science & Engg, Sipna's COET Amravati, SGB, Amravati University. India.

I. INTRODUCTION

The microelectronics industry has followed the trajectory set by Moore's Law from more than three decades. **Moore's law** is a rule of thumb for the history of computing hardware whereby the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years. The microprocessor industry has leveraged this evolution to increase uniprocessor performance by decreasing cycle time and increasing the average number of executed instructions per cycle (IPC).

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

Parallelism for client and mobile applications is harder because they are turnaround oriented, and the use of parallelism to reduce response time requires more algorithmic work. Furthermore, mobile systems are power constrained, and improved wireless connectivity enables shifting computations to the server

This suggests the some questions as the following way:

A. Can such applications force parallelism?

B. Will compelling new client applications emerge that require significant added client performance?

C. Can we develop a programming environment that enables a large programmer community to develop parallel codes?

D. Can multicore architectures and their software scale to hundreds of cores that hardware will be able to support in decade?

On such topic is ongoing at Parallel Computing Laboratory (ParLab) at the university of California, Berkeley, the Universal Parallel Computing Research Center at the University of

Illinois, Urbana- Champaign (UPCRC-Illinois), and Stanford University's Pervasive Parallelism Laboratory (PPL). These three centres's are parallelizing specific applications, rather than developing technology for undefined future applications. Primary focus on client computing, designing technology that can work well up through hundreds of cores. All three also reject a single-solution approach, assuming instead that software is developed by teams of programmers with different specialties requiring different sets of tools. Even though the work at the three centers bears many similarities, they focus on different programming environments, approaches for the production of parallel code, and architectural supports.[1]

II. PARALLEL COMPUTING LABORATORY (ParLab)

Intel and Microsoft invited 25 universities to propose parallel computing centers for recognizing the difficulty of the multicore challenge. The Berkeley ParLab was selected in 2008. Many datacenter applications have ample parallelism across independent users, so the Par Lab focuses on parallelizing applications for clients. Ii is the teeam of 50 PhD students and dozens of faculty from many fields who work together toward making parallel computing productive, performant, energy-efficient, scalable, portable, and at least as correct as sequential programs..[2]

The goal is to enable the productive development of efficient parallel programs by tomorrow's programmers. We believe future programmers will be either domain experts or sophisticated computer scientists because few domain experts have the time to develop performance programming skills and few computer scientists have the time to develop domain expertise. We believe that software architecture is the key to designing parallel programs, and the key to these programs' efficient implementation is frameworks. In our approach, the basis of both is design patterns and a pattern language. Borrowed from civil architecture, design pattern means solutions to recurring design problems that domain experts learn. A pattern language is an organized way of navigating through a collection of design patterns to produce a design. [2][3]

A software architecture is hierarchical composition of computational and structural patterns, which we refine using lower-level design patterns. They define a pattern-oriented software framework as an environment built around a software architecture in which

customization is only allowed in harmony with the framework's architecture. For example, if based on the pipe-and-filter style, customization involves only modifying pipes or filters. [2]

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

III. SEJITS: BRIDGING THE PRODUCTIVITY- EFFICIENCY GAP

When they write in high-level productivity layer languages (PLLs) e.g. Python or Ruby with abstractions that match the application domain; studies have reported factors of three to 10 fewer lines of code and three to five times faster development when using PLLs rather than efficiency-level languages (ELLs) such as C++ or Java. However, PLL performance might be orders of magnitude worse than ELL code, in part due to interpreted execution. They are developing new approach to bridge the gap between PLLs and ELLs. The modern scripting languages such as Python and Ruby include facilities to allow late binding of an ELL module to execute a PLL function. When first called to determine whether the AST(abstract syntax tree) can be transformed into one that matches a computation performed by some ELL module. If the AST cannot be matched to an existing ELL module or the module targets the wrong hardware, the PLL interpreter just continues executing as usual. This approach preserves portability because it doesn't modify the source PLL program.

Just-in-time (JIT) code generation and specialization is well established with Java and Microsoft .NET, In the approach selectively specializes only those functions that have a matching ELL code generator, rather than having to generate code dynamically for the entire PLL. Also, the introspection and meta programming facilities let us embed the specialization machinery in the PLL directly rather than having to modify the PLL interpreter. Hence, we call our approach

selective, embedded, just-in-time specialization (SEJITS).[4] SEJITS helps domain experts use the work of efficiency programmers. Efficiency programmers can ‘‘drop’’ new modules specialized for particular computations into the SEJITS framework, which will make runtime decisions when to use it. The goal is to specialize an entire image processing computation, such as in Figure 1.

IV. APPLICATIONS and PLATFORMS

In future applications will have footholds in both mobile clients and cloud computing. We’re investigating parallel browsers because many client applications are downloaded and run inside a browser.[5] both client and cloud are concerned with responsiveness and power efficiency. The application client challenge is responsiveness while preserving battery life given various platforms. The application cloud challenge is responsiveness and throughput while minimizing cost in a pay as you go environment. In addition, cloud computing providers want to lower costs, which are primarily power and cooling. [6]

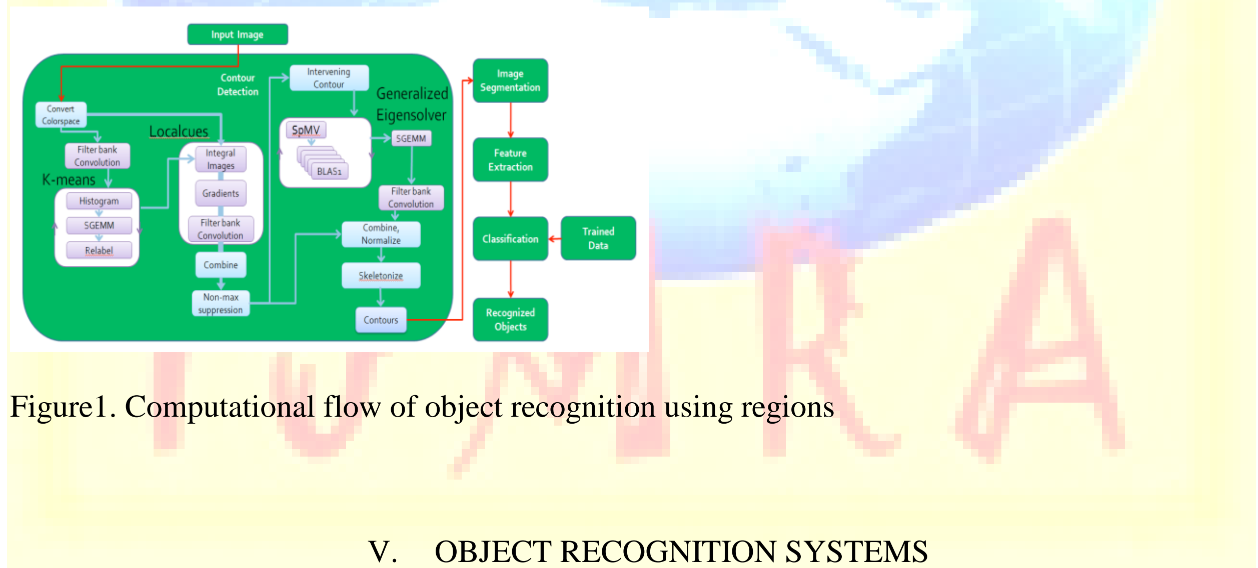

Figure1. Computational flow of object recognition using regions

V. OBJECT RECOGNITION SYSTEMS

Clearly, an object recognition system will be a major component of such an application. Figure 1 shows the computational flow of a local object Recognizer, which gets good results on vision bench marks. Although high quality, it is computationally intensive: it takes 5.5 minutes to identify five kinds of objects in a 0.15MB image, which is far too slow for our application.[7]

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

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

We choose application patterns manually. Now we investigate the contour detector, which uses 75% of the time, as an example. Among all contour detectors, the gPb algorithm achieves The highest accuracy. Figure 1 summarizes our version of the gPb algorithm. The major computational bottlenecks are the localcue computation and the generalized eigensolver. For the localcue computation, we replace the explicit histogram accumulation by integral image, and Apply parallel scan for realizing integral image. For the generalized eigensolver, we proposed a highly parallel SpMV kernel and investigated appropriate reorthogonal approaches for the Lanczos algorithm. By selecting good patterns to form a proper software architecture that reveals parallelism and then exploring appropriate algorithmic approaches and parallel implementations within that software architecture, we accelerated contour execution time by 140X from 4.2 minutes to 1.8 seconds on a GPU [1], [7].

VI. PERFORMANCE MEASUREMENT and AUTOTUNING

The prevailing hardware trend of dynamically improving performance with little software visibility has become counterproductive; software must adapt if parallel programs are going to be

portable, fast, and energy efficient. Hence, parallel programs must be able to understand and measure any computer so that they can adapt effectively. This perspective suggests architectures with transparent performance and energy consumption and Standard Hardware Operation Trackers (SHOT). [8] SHOT enables parallel programming environments to deliver portability, performance, and energy efficiency. For example, we used SHOT to examine alternative data structures for the image contour detector, by examining realized memory bandwidth versus the data layout. Autotuners produce high-quality code by generating many variants and measuring each variant on the target platform. The search process tirelessly tries many unusual variants of a particular routine. Unlike libraries, autotuners also allow tuning to the particular problem size. Autotuners also preserve clarity and help portability by reducing the temptation to mangle the source code to improve performance for a particular computer.

VII. THE ARCHITECTURE and OS of 2020

We expect the client hardware of 2020 will contain hundreds of cores in replicated hardware tiles. Each tile will contain one processor designed for instruction-level parallelism for sequential code and a descendant of vector and GPU architectures for data-level parallelism. Task-level parallelism occurs across the tiles. The number of tiles per chip varies depending on cost-performance goals. Thus, although the tiles will be identical for ease of design and fabrication, the chip supports heterogeneous parallelism. [1]

VIII. UPCRC-ILLINOIS

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

The work on applications focuses on the creation of compelling 3D Web applications and human-centered interfaces. We have a strong focus on programming language, compiler, and

runtime technologies aimed at supporting parallel programming models that provide simple, sequential-by-default semantics with parallel performance models and that avoid concurrency bugs. Our architecture work is focused on efficiently supporting shared memory with many hundreds of cores[1]

IX. APPLICATIONS

The 3D Internet will enable many new compelling applications. For example, the Teeve 3D teleimmersion framework and its descendants have supported remote collaborative dancing, remote Tai-Chi training, manipulation of virtual archeological artifacts, training of wheelchair-bound basketball players, and gaming.[1] Applications are limited by a low frame rate and inability to handle visually noisy environments. Broad real time usage in areas such as multiplayer gaming or telemedicine requires several orders of magnitude performance improvements in tasks such as the synthesis of 3D models from multiple 2D images; the recognition of faces, facial expressions, objects, and gestures; and the rendering of dynamically created synthetic environments. Such tasks must execute on mobile clients to reduce the impact of Internet latencies on real-time interactions. Many of the same tasks will be at the core of future human centered ubiquitous computing environments that can understand human behavior and anticipate needs, but this will require significant technical progress.

X. USING PROGRAMMING ENVIRONMENT

Here distinguish between concurrent programming that focuses on problems where concurrency is part of the specification and parallel programming that focuses on problems where concurrent execution is used only for improving a computation’s performance. Reactive systems (such as an operating system, GUI, or online transaction-processing system) where computations (or transactions) are triggered by nondeterministic, possibly concurrent, requests or events use concurrent programming. Parallel programming is used in transformational systems, such as in scientific computing or signal processing, where an initial input (or an input stream) is mapped through a chain of (usually) deterministic transformations into an output (or an output stream).

The prevalence of multicore platforms doesn't increase the need for concurrent programming or make it harder; it increases the need for parallel programming.

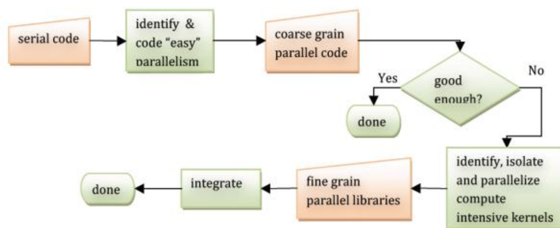

Figure 2 Parallel code development work flow

Figure 2 represents a schematic view of the parallel software creation process. Developers often start with a sequential code, although starting from a high-level specification is preferred. It's often possible to identify outer loop parallelism where we can encapsulate all or most of the sequential code logic into a parallel execution framework (pipeline, master-slave, and so on) with little or no change in the sequential code. If this doesn't achieve the desired performance, developers identify compute-intensive kernels, encapsulate them into libraries, and tune these libraries to leverage parallelism at a finer grain, including single-instruction, multiple-data (SIMD) parallelism. Whereas the production of carefully tuned parallel libraries will involve performance coding experts, simple, coarse-level parallelism should be accessible to all. Here propose the "simple parallelism" by developing refactoring tools that let us convert sequential code into parallel code written using existing parallel frameworks in C# or Java as well as debugging tools that help identify concurrency bugs.[1]

XI. ARCHITECTURE

In fact, parallel programs communicate and synchronize in stylized ways. A key to shared memory scaling is adjusting coherence protocols to leverage the prevalent structure of shared memory codes for performance. Exploring three approaches, The Bulk Architecture is executing coherence operations in bulk, committing large groups of loads and stores at a time.[9]

In this architecture, memory accesses appear to interleave in a total order, even in the presence of data races which helps software debugging and productivity while the performance is high through aggressive reordering of loads and stores within each group.

XII. The STANFORD UNIVERSITY PERVASIVE PARALLELISM LABORATORY

Stanford University officially launched the Pervasive Parallelism Laboratory in May 2008. PPL's goal is to make parallelism accessible to average software developers so that it can be freely used in all computationally demanding applications. The PPL pools the efforts of many leading Stanford computer scientists and electrical engineers with support from Sun Microsystems, NVIDIA, IBM, Advanced Micro Devices, Intel, NEC, and Hewlett-Packard under an open industrial affiliates program. [1]

XIII. The PPL APPROACH to PARALLELISM

A fundamental premise of the PPL is that parallel computing hardware will be heterogeneous. This is already true today; personal computer systems currently shipping consist of a chip multiprocessor and a highly data parallel GPU coprocessor. Large clusters of such nodes have already been deployed and most future high-performance computing environments will contain GPUs. To fully leverage the computational capabilities of these systems, an application developer must contend with multiple, sometimes incompatible, programming models. Shared memory multiprocessors are generally programmed using threads and locks (such as pthreads and OpenMP), while GPUs are programmed using data-parallel languages (such as CUDA and OpenCL), and communication between the nodes in a cluster is programmed with a message passing library (such as MPI). Heterogeneous hardware systems are driven by the desire to improve hardware productivity, measured by performance per watt and per dollar. This desire will continue to drive even greater hardware heterogeneity that will include special purpose processing units.

XIV. The PPL RESEARCH AGENDA

In such application areas demand enormous amounts of Computing power to process large amounts of information, often in real time. They include traditional scientific and engineering applications from geosciences, mechanical engineering, and bioengineering; a massive virtual world including a client-side game engine and a scalable world server; personal robotics including autonomous driving vehicles and robots that can navigate home and office environments; and sophisticated data-analysis applications that can extract information from huge amounts of data.

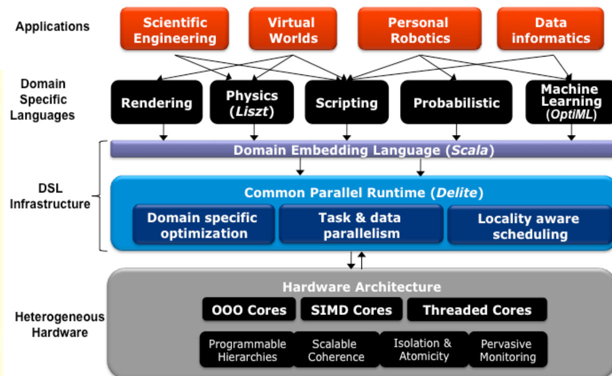

Figure 3: A layered approach to the problem of parallel computing.

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

We expect that most programming of future parallel systems will be done in DSLs at the abstraction level of Matlab or SQL. DSLs enable the average programmerto be highly productive in writing parallel programs by isolating the programmer from the details of parallelism, synchronization, and locality. The use of DSLs also recognizes that most applications aren't written from scratch, but rather built by combining existing systems and libraries. [10]

In common parallel runtime system (CPR), called Delite, supports both implicit task-level parallelism for generality and explicit data-level parallelism for efficiency. The CPR maps the parallelism extracted from a DSL-based application to heterogeneous architectures and manages

the allocation and scheduling of processing and memory resources. The mapping process begins with a task graph, which exposes task- and data-level parallelism and retains the high-level domain knowledge expressed by the DSL.

XV. LISZT

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

XVI. DELITE

Delite's layered approach to parallel heterogeneous programming hides the complexity of the underlying machine behind a collection of DSLs. The use of DSLs creates two types of programmers: application developers, who use DSLs and are shielded from parallel or heterogeneous programming constructs and DSL developers, who use the Delite framework to implement a DSL. The DSL developer defines the mapping of DSL methods into domain-specific units of execution called Ops. It holds the implementation of particular domain specific operation and other information which is needed by runtime for parallel execution. [1]

Research Centers	Programming Environment	Applications	Architecture

Berkeley	PLL(Productivity Layer Language) Python or Rubby ELL(Efficiency Level Language) C++ or Java SEJITS (Selective, embedded, just-in-time)	Mobile clients and cloud computing	Instruction level parallelism for sequential code and a descendant of vector and GPU architecture
Illinois	C# or Java	3D Web Applications and Human centered interfaces	The bulk architecture i.e. system with hundreds of conventional cores, and possibly thousands of light weight cores
Stanford	Domain specific Languages Such as Metlab or SQL, mesh-based PDE DSL called Liszt, Machine Learnig DSL Called OptiML	Traditional scientific and engineering applications from geosciences, mechanical engineering, and bioengineering; massive virtual world include a client side game engine and a scalable world	Layer approach architecture

		server; personal robotics	

Table 1: Shows the programming environment, applications and architecture from Berkeley, Illinois and Stanford[1]

XVII. CONCLUSION

From all of these three centers carry many similarities. In this paper we are review on architecture, programming environment, applications from all three centers that is Berkeley, Illinois and Stanford. All three centers assume that future microprocessors will have hundreds of cores and will combine different types of cores in one chip. These center works to support well a division of labor between efficiency expert and application domain expert. However Berkeley to use the application framework or programming language such as Python, while efficiency expert programs libraries in language like C++. The performance is gained by Selectively Embedded, Just-in –time. Illinois and Stanford are concerned with better architectural support for communication and synchronization while Berkeley is focused on isolation and measurement. And finally, Berkeley and Stanford are concerned with novel OS and run-time structure to support parallelism.

REFERENCES

[1]. Bryan Catanzaro, Armando Fox, Kurt Keutzer, David Patterson, Bor-Yiing Su, Marc Snir, Kunle Olukotun, Pat Hanrahan, Hassan Chafi, "Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford," IEEE Micro, vol. 30, no. 2, pp. 41-55, March/April, 2010.

[2.] K. Asanovic´ et al., ‘‘A View of the Parallel Computing Landscape,’’ Comm.ACM, vol. 52, no. 10, Oct. 2009, pp. 56-67.

[3.] K. Keutzer and T. Mattson, ‘‘A Design Pattern Language for Engineering (Parallel) Software,’’ to appear in Intel Technical J. vol. 13, no. 4, 2010.

[4.] B. Catanzaro et al., ''SEJITS: Getting Productivity and Performance with Selective Embedded JIT Specialization,'' Proc. 1st Workshop Programmable Models for Emerging Architecture (PMEA), EECS Dept., Univ. of California, Berkeley, tech. report UCB/EECS-2010-23, Mar. 2010.

[5]. C.G. Jones et al., ''Parallelizing the Web Browser,'' Proc. 1st Usenix Workshop Hot Topics in Parallelism (HotPar 09), Usenix Assoc., 2009, http://parlab.eecs.berkeley. edu/publication/220.

[6.] M. Armhurst et al., ''Above the Clouds: A Berkeley View of Cloud Computing,'' Comm. ACM, vol. 53, no. 4, Apr. 2010, pp. 50-58.

[7]. C. Gu et al., ''Recognition Using Regions,'' Proc. Computer Vision and Pattern Recognition (CVPR 09), [8]. IEEE CS Press, 2009, pp. 1030-1037. B. Catanzaro et al., ''Efficient, High-Quality Image Contour Detection,'' Proc. IEEE Int'l Conf. Computer Vision (ICCV 09), 2009; http://www.cs.berkeley.edu/~catanzar/Damascene/iccv2009.pdf.

[8]. S. Bird et al., ''Software Knows Best: Portable Parallelism Requires Standardized Measurements of Transparent Hardware,'' EECS Dept., Univ. of California, Berkeley, tech. report, Mar. 2010.

[9]. J. Torrellas et al., ''The Bulk Multicore for Improved Programmability,'' Comm. ACM, Dec. 2009, pp. 58-65.

[10]. K. Fatahalian et al., ''Sequoia: Programming the Memory Hierarchy,'' Proc. 2006 ACM/IEEE Conf. on Supercomputing, ACM Press, 2006; http://graphics.stanford.edu/papers/sequoia/sequoia_sc06.pdf.