Research Statement

Katherine Yelick
University of California, Berkeley

Applications on large-scale multiprocessors like the ASCI machines, are often written using message passing or, when the nodes are shared memory multiprocessors, a combination of message passing with threads. The programming model is difficult to use, and the performance often disappointing. For example, many important applications run at under 10% efficiency on the machines. Some of the efficiency loss is due to overheads of parallelism and some to poor uniprocessor performance. Moreover, algorithms that are theoretically more efficient are often avoided in favor of simpler, more regular algorithms, because of the difficulty of programming the machines.

The goal of my research is to develop techniques for obtaining high performance on a wide range of computational platforms and to ease the programming effort required to obtain performance. My approach has been multi-faceted, in short, pursuing any technique that is like to lead to better performance. As a result, my work does not fit into a single category within Computer Science, but rather covers programming languages, compilers, systems, algorithms, and architecture.

Global Address Space Languages

Parallel programming is inherently harder than sequential programming, but most programmers agree that programming in a shared memory model is easier than a message-passing model. In 1991 my group demonstrated that a relatively small change to a standard compiler (in this case gcc), when combined with lightweight communication primitives, could allow one to run shared memory style programs on distributed memory machines. This work later developed into one of the first shared address space languages, Split-C, in a research collaboration with David Culler [12]. Global address space languages make it possible to program large scale, distributed memory multiprocessors and clusters using a shared address space. The key idea is to change the representation of pointers and arrays in the language to allow data structures to be spread over processors. The compiler is responsible for decoding pointer and array references into message calls on machines that do not directly support shared memory.

More recently, I was involved in the design of Unified Parallel C (UPC), which merged some of the ideas from three shared address space dialects of C: Split-C, AC (from IDA), and PCP (from LLNL) [7]. In recent years, UPC has gained recognition as an alternative to message passing programming for large-scale machines. Compaq, Sun, Cray, HP, and SGI are implementing UPC, and I am currently leading a large effort at LBL to implement UPC on Linux clusters and IBM machines and to develop new optimizations.

Meanwhile, in a joint research effort with Susan Graham, Paul Hilfinger, Alex Aiken, and Phil Colella (from LBL) we developed a shared address space language called Titanium, which is based on Java rather than C [8]. The primary design goals of Titanium are performance and safety. It inherits some of its safety features from Java and adds some additional ones to for parallelism. To satisfy the performance goal, Titanium uses the SPMD model and the same partitioned global address space of Split-C and UPC: any thread may access an object on any processor, but the programmer controls the alignment of data with threads. To avoid the overhead of byte-code interpretation, the Titanium specification allows for compilation directly to native code. The Titanium project has demonstrated several important results: 1) Titanium provides a common programming model for shared memory and distributed memory -- it runs on any shared memory machine with POSIX threads, on most distributed memory multiprocessors and clusters. 2) Code generation from Titanium is quite competitive with C. It is within 20% of a C performance (gcc on Linux) on four of six SciMark benchmarks, and within 2x on the other two. 3) Complex applications have been written in

Titanium, including challenging adaptive mesh refinement algorithms and the heart simulation described below.

The optimization problem for explicitly parallel languages like UPC, Titanium, or even Java, is different than for sequential languages or data-parallel languages (which have a sequential execution model). Most optimizations involve some form of code motion, which, in sequential programs, are limited by dependences in the sequential execution defined by the source program. In a parallel execution model, an additional analysis is needed to ensure that operations reordered on one processor cannot be observed on another. There is no common agreement on what semantics should be allowed -- the current debate over the parallel semantics on Java, which is producing a new language specification, is one example; lack of uniform semantics across C compilers in the meaning of volatile and non-volatile shared variables is another. In both cases, compilers may be overly conservative or may incorrectly order statements relative to programmer expectations. In particular conventional C compilers are often used in multi-threaded code, but they are not designed for such environments, and they often perform transformations that are at best surprising and at worst incorrect.

Together with Arvind Krishnamurthy, I developed new compiler analyses that allow one to implement a stronger semantic model on a weaker model [11] For example, it can be used to ensure sequential consistency on a machine with weaker semantics. (MIPS, SPARC, Alpha, and Pentium processors all have weaker semantics.) While several other researchers have done theoretical work in the area, my work is key to demonstrating that this kind of analysis may be practical. We developed new algorithms to handle programs with branches and to avoid an exponential blowup in the number of processors, and we also have the only implementation of the analysis, which was done for both Titanium and for a pointer-free subset of C. We also incorporated common synchronization primitives into the analysis, which were a key to making the analysis practical [9]. The analysis preserves program safety in the following sense: programs that use built-in synchronization primitives will be susceptible to more precise analysis and therefore better optimizations, but programs that use primitives implemented directly on shared memory will still be correct. This is much stronger than the kinds of analyses suggested by some of the architectural work on memory systems (such as “proper labeling”), which requires that system synchronization primitives be used exclusively, or correctness is not ensured.

Single Processor Optimizations

It is well understood that blocking or tiling of many dense matrix algorithms improves their performance on the deep memory hierarchies of modern processors, although the specific choice of blocking factor can be very difficult to find. The problem is even more difficult for problems with irregular memory access patterns. A canonical example is sparse matrix-vector multiplication, the core of many sparse iterative solvers. These algorithms exhibit little spatial locality, because of the indexed representation of the matrix, and little temporal locality, because there are only two floating-point operations performed per matrix element.

In the Sparsity project, Eun-Jin Im and I developed techniques for automatically optimizing sparse matrix algorithms for memory hierarchies. The three basic optimization techniques in Sparsity are: register blocking, cache blocking, and use of multiple vectors [3]. Register blocking differs from the dense case, because the zeros in the matrix are filled in to achieve a uniform block size; the small dense sub-problems are then optimized by loop unrolling and instruction scheduling. Register blocking significantly improves the ``raw" MFLOP/s rate of the code, but can slow execution time due to the extra computation on added zeroes. Sparsity contains a code generation engine as well as a two-component performance model for selecting a good register block size. The first component of the model is a performance profile of the machine, measured by running a dense matrix in sparse format for a large range of block sizes. The second component is matrix-specific, and is an estimate of the number of zero elements that would be filled in for

each block size in a given range. The two components are combined to minimize overall running time. We did an extensive set of experiments on over 40 matrices from a variety of application domains on several different machines. This included validation of the model by comparing it to exhaustive search on a smaller number of matrices. Overall, we have shown speedups up to 3x, with more recent machines like the Pentium IV showing some of the highest speedups. We expect these optimizations will be increasingly important for future machines as the gap between processor speed and DRAM performance grows.

The second optimization technique used in Sparsity is cache blocking, which is useful primarily for enormous rectangular matrices in which the source vector does not fit in cache. Cache blocking reorganizes the matrix layout by storing blocks of the matrix contiguously in memory, but each block retains its sparse format, so there is little overhead. Again, we developed a model for determining whether cache blocking is useful and for estimating the best cache block size. Cache blocking showed speedups of 3x for a matrix taken from web document retrieval, whose dimensions are the number of keywords by the number of documents on the web [4].

Sparsity also implements a variation of basic sparse matrix-vector multiplication in which a sparse matrix is multiplied by a set of dense vectors. This operation arises, for example, when there are multiple right-hand sides in a linear solver or when a higher-level algorithm has been blocked. The introduction of multiple vectors offers enormous optimization opportunities, effectively changing a matrix-vector (BLAS-2) operation into a matrix-matrix (BLAS-3) operation. Even for dense matrices, the latter algorithms have much higher data reuse and therefore better performance. (BLAS-3 performance is typically 5x or more that of BLAS-2 performance, when both are hand-tuned.) Sparsity shows speedups as high as 6x when the code is organized to take advantage of multiple vectors.

The work done in Sparsity is continuing in an NSF-funded project called BeBOP (Berkeley Benchmarking and OPtimization), which is a joint project with Jim Demmel. Over the past summer, we worked with 12 undergraduate students on various aspects of the BeBOP agenda. One group of students looked at the problem of taking advantage of symmetry in optimizing sparse matrix-vector multiplication for symmetric matrices. Support for symmetric matrices is important in many applications, and we were able to demonstrate speedups of 50% by exploiting symmetry, and an additional 2x by combining it with register blocking. The issues of combining the symmetry and register blocking were surprising subtle, due to the particular structure of non-zeros that arise in practice, and to the increased register demands in processing a single block of a symmetric matrix. Another group looked at a related algorithm, sparse triangular solve, and demonstrated similar speedups using both register blocking and various heuristics that switch from a sparse to dense representation and algorithm when the matrix becomes relatively dense. This latter technique was a especially useful in a matrix from Spice circuit simulation, demonstrating speedups of 50% overall. Technical reports on the work done by both groups are forthcoming.

IRAM: An Architecture for Memory-Intensive Applications

The difficulty of optimizing many algorithms for memory hierarchies originally led to my interest in alternate memory system designs, and in particular the use of mixed logic and DRAM, which avoids the off-chip accesses to DRAM, thereby gaining bandwidth, while lowering latency and energy consumption. In the IRAM project, a joint effort with David Patterson, we developed an architecture to take advantage of this technology. The IRAM processor is a single chip system designed for low power and high performance on multimedia applications and achieves an estimated 6.4 GOP/s in a 2 Watt design [2]. The project is in the final stages of design, and will be fabricated by IBM later in the next few months.

Many architectural ideas that appear to be useful from a hardware standpoint fail to achieve wide acceptance due to lack of compiler support. The IRAM architecture is based on vector instructions, historically reserved for expensive vector supercomputers designed for large-scale scientific and engineering

applications. An important component of this project has been the compiler and benchmarking work done to help evaluate different designs and to demonstrate the ability to achieve high performance from compiled code. We demonstrated that a vectorizing compiler can effectively expose the on-chip bandwidth on IRAM, and that vectorization can be applied to the narrow data types (8, 16, and 32-bit) that arise in multimedia applications [5]. Vector architectures are becoming increasingly popular for graphics and multimedia in the commercial arena, although they often come under the name ``SIMD extension." Our work on automatic vectorization has shown that the IRAM instruction set, with it full support for strided and indexed memory operations on variable-width datatypes, is a better compilation target than the SIMD extensions like Intel's MMX.

Earlier benchmarking work on IRAM for the FFT problem directly influenced the instruction set design. Initial work showed that IRAM needed an efficient mechanism for computing reductions. Our work on benchmarking the FFT (in assembly) revealed that a small generalization of the instructions for reductions could make the FFT run 5x faster, without significant changes to the hardware. Far from being an ``FFT instruction" we have recently shown how to use these instructions in sorting and computing histograms. In addition to the IRAM project on campus (funded by DARPA and MICRO), the benchmarking work is continuing with a project at LBL (with Xiaoye Li, Lenny Oliker, and Parry Husbands) to understand emerging architectures that might be available for future high-end machines. This may allow DOE to anticipate future architecture for scientific computing, rather than having them disappear due to market shifts out of their control.

ISTORE: Reliability of Large-Scale Clusters

Many of the systems used for large-scale storage, web service, or scientific computing are often based on the scalable cluster model. It has the advantage of scalability and low cost, but even high profile systems at eBay, Amazon, and the New York Stock Exchange have proven susceptible to outages. The ISTORE project looked at hardware and software techniques for building highly available systems. The ISTORE hardware has several levels of redundancy, as well as built-in monitoring for temperature, humidity, vibration, and intrusion, as well as a separate diagnostic system that contains a processor per node and an independent network [1]. One of my students, Daniel Hettena, has been developing an interface for accessing the diagnostic information, and also looking at ways of using the redundant Ethernet interfaces in each node to dynamically achieve high bandwidth (by striping messages across the interfaces) or high availability (using subsets of interfaces when there is a failure in an interface or network link).

On the software side, I am working with another student, Noah Treuhaft, on the general trade-off between high available and high performance in the design of distributed data structures. In earlier work, we developed dynamic techniques to address performance heterogeneity, which is increasingly important for large-scale clusters. This was an extension of the work described in the “River” paper [6]. Our work address heterogeneity in parallel I/O, which arise from different hardware or software versions, fragmentation, or simply different layouts. The general problem of performance heterogeneity will be increasingly important as scientific applications are developed for loosely coupled clusters and ``the grid."

Application Level Tools

My research group has a strong tradition of doing research in the applications, as a driving function for our systems work. Our most ambitious effort is to build a tool for fluid simulation using the immersed boundary method. The method was developed by Charlie Peskin and Dave McQueen at Courant Institute, models a biological system as a set of elastic fibers within an incompressible fluid. Best known for its use for heart simulation, the method has also been used to simulate platelet coagulation during clotting, embryo growth, fluid flow in the cochlea, and other biological (as well as non-biological) systems. Several research groups in the U.S. and in Europe have used the immersed boundary method code, but the version developed

at Courant runs only on vector and shared memory supercomputers, for which availability is quite limited. We have developed an implementation of the immersed boundary method in Titanium language, which is designed for distributed memory machines and large clusters as part of the NPACI partnership at SDSC. We have run a full heart model on a small number of time steps, and are working on several optimizations, including use of a more scalable solver based on the dissertation work of my former postdoc, Greg Balls. We are also developing a performance model for the application, similar to our model used several years ago for the 2D problem [10]. We plan to run a full execution of a heartbeat (which consumed roughly 100 hours of Cray C90 time) before the end of the year. While our short term goal is accurate and efficient modeling of the heart, our longer term goal is to produce a tool for the immersed boundary method that uses the features of Java (and therefore Titanium) to allow Biologists to instantiate other physical systems within our generic parallel framework. We believe such a system would be useful in building a full simulation of several interacting organ systems or, ultimately, even a “digital human.”

1. Oppenheimer, D., A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D.A. Patterson, and K. Yelick, “ROC-1: Hardware Support for Recovery-Oriented Computing.” To appear in IEEE Transactions on Computers, Special Issue on Embedded Fault-Tolerant Computer Systems, 2001.
2. C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, K. Yelick, “Hardware/Compiler Co-development for an Embedded Media Processor,” Proceedings of the IEEE. To appear.
3. E.-J. Im and K. Yelick, “Optimizing Sparse Matrix Computations for Register Reuse in Sparsity,” Proceedings of the International Conference on Computational Science, San Francisco, May 2001.
4. E.-J. Im and K. Yelick, ``Optimization of Sparse Matrix Kernels for Data Mining," Proceedings of Text Mine Workshop '01, Chicago, April 7, 2001.
5. D. Judd, K. Yelick, C. Kozyrakis, D. Martin, and D. Patterson, “Exploiting On-Chip Memory Bandwidth in the VIRAM Compiler,” Second Workshop on Intelligent Memory Systems, Cambridge, November 2000.
6. R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. A. Yelick, “Cluster I/O with River: Making the Fast Case Common, Workshop on I/O in Parallel and Distributed Systems,” Atlanta, GA, May 1999.
7. W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, “Introduction to UPC and Language Specification,” CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
8. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. ``Titanium: A High-Performance Java Dialect," Concurrency: Practice and Experience, September-November 1998, pp. 825-36. Earlier version was presented at the ACM Workshop on Java for High-Performance Network Computing, February 1998.
9. Krishnamurthy and K. Yelick, ``Analyses and Optimizations for Shared Address Space Programs." Journal of Parallel and Distributed Computation, vol.38, (no.2), Academic Press, 1 Nov. 1996. pp.130--44.
10. S. Steinberg, J. Yang and K. Yelick, ``Performance Modeling and Composition: A Case Study in Cell Simulation." International Parallel Processing Symposium, April 1996.
11. Krishnamurthy and K. Yelick, ``Optimizing Parallel Programs with Explicit Synchronization." Proceedings of the SigPlan Conference on Programming Language Design and Implementation, San Diego, California, June 1995.
12. D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eiken, and K. Yelick, ``Parallel Programming in Split-C," Supercomputing '93, November 1993.