Programming Models for Irregular Applications

Katherine A. Yelick [1]

Computer Science Division

University of California at Berkeley

The difficulty of parallelizing a given program is strongly correlated to the degree of irregularity in the program's structures. The applications that have been parallelized most successfully on large scale multiprocessors are those with a high degree of regularity. The goal of our research is to provide languages and libraries that will simplify parallel programming for irregular applications, and consequently increase the class of applications that can be effectively parallelized. We have approached this problem by implementing irregular applications on a CM5, while at the same time developing layers of system level software to provide a common base for the applications.

In this abstract we describe the structure of some irregular applications, focusing on the ways in which irregular structures arise. We then point out common features of the applications, feature which should therefore be supported by languages and systems for these problems. Finally, we describe some basic pieces of systems support and evaluate them based on their usefulness in the applications.

Parallel programs may exhibit at least three different kinds of irregularity. The first kind of irregularity appears as *irregular control structures*, namely conditional statements, which make it inefficient to run on synchronous programming models such as that provided by an SIMD machine. A second kind appears in the form of *irregular data structures*, which include unbalanced trees, graphs, and unstructured grids. These data structures lead to dynamic scheduling and load balancing requirements, since it is often impossible to predict the amount of computation that will be associated with a given data structure. The third type of irregularity is *irregular communication patterns*, which lead to nondeterminism, since one cannot predict the order in which communication events will occur. Communication irregularity is typically caused by either data or control irregularity, and the three together define the most challenging class of irregular problems.

Our conclusions are based on experience with the following applications: a Gröbner basis computation, an event-based timing circuit simulator [6], a Cholesky factorization algorithm for sparse matrices, a BDD-based circuit verifier, an automatic theorem prover, a symmetric eigenvalue computation, a toy ocean simulator, and an electromagnetics model. All of these have at least one kind of irregularity, and most of them display good speedups. The performance of the Gröbner basis program, for example, is comparable to the best shared memory implementation [10] on small numbers of processors, and it scales past a hundred processors when the inputs are large enough. The performance of Cholesky is disappointing, probably due to the lack of blocking in our implementation [2]. However, some of the lessons learned about programming models are still relevant, since they would apply to blocked algorithms as well. The circuit simulator is, again, competitive with shared memory implementations on a small number of processors, and the speedup on 128 processors is as high as 50, with potential for further scaling. The last three applications in the list are more regular, and have nearly linear speedups.

[1] This work was supported in part by AT&T, the Semiconductor Research Corporation, Lawrence Livermore National Laboratory, a National Science Foundation Infrastructure Grant (number CDA-8722788), the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, and the University of California Committee on Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

The Gröbner basis problem is used in a wide variety of application areas, including Robotics and Computer Aided Design, when working with a system of polynomial equations [8]. Given a set of multivariate nonlinear polynomials, the ideal defined by the polynomials has a particular basis called the Gröbner basis. The Gröbner basis is useful in finding roots of polynomials and in related problems. The parallel algorithm, which is based on Buchberger's sequential one, is very irregular: the communication patterns are non-local, and the basic computation steps are unpredictable [10]. Not only do the polynomials vary in size, but the time to perform basic polynomial operations varies even for polynomials of the same size. As a result, it is difficult to schedule the algorithm in advance. The Cholesky factorization is used to factor symmetric positive definite matrices; the irregularity in this case comes from the irregularity in the sparse matrix itself [2]. The computation is divided into an ordering phase, a symbolic factorization phase, and a numeric phase; in our current implementation only the numeric phase runs in parallel.

While the current implementations are written at a low level of abstraction, using C with various levels of system support, the common features of the applications suggest higher level language constructs that would be useful. The first such feature is *nondeterminism*. For Gröbner basis and Cholesky, the most efficient implementations we have are nondeterministic: depending on the timing of communication and computation, different executions may occur. The Gröbner basis computation proceeds by choosing a pair of polynomials from a set and computing a third polynomial, which is then added to the set. Any order of choosing pairs is correct, but the choice can significantly affect performance. Cholesky performs updates on submatrices (by addition), which are determined by operations on columns to the left of the submatrix. The order in which updates occur may vary, depending on the density of columns producing the updates. We believe this kind of nondeterminism is essential to performance in irregular applications, since any deterministic schedule would have to choose an evaluation order which may result in poor performance for *some* input. This observation suggests that parallelizing compilers alone cannot produce these parallel programs, since a compiler does not have sufficient information to determine which alternate behaviors are correct.

The second common feature of the applications is the need for *distributed data structures*, as well as *layout directives* for placing the data. Analogous to the the Fortran D constructs for laying out arrays [4], data structures such as trees may be layed out in a *blocked* fashion, with subtrees placed on a single node, or *cyclic* pattern, with neighboring nodes spread among processors. In a tree where the edges represent dependencies between computations, a cyclic layout results in more parallelism, while a blocked one has less communication overhead. Both layouts are useful for other structures, including graphs, bags (multisets), and sets.

The final observation about the applications relates to *global control operators* that are specific to the data structures. A common control construct is iteration over the elements of a distributed data structure, but unlike data parallel constructs, these iterations overlap with one another, and unlike array iterations, the iteration space may not be explicit. Different iteration semantics are needed in different contexts: in Gröbner basis, an individual iteration must be serializable, although there is no fixed order and multiple iterations may run in parallel; in Cholesky, individual iterations may be executed in parallel.

At the system level, we have developed software layers that simplify programming, and are evaluating these for both performance and added ease of programming. At the lowest level, most of our software was written on top of CMAM, the Connection Machine Active Message layer [11]. On top of this, we have implemented a distributed *work queue* for scheduling tasks, along with some basic synchronization primitives and a mobile object layer. One of our immediate goals is evaluate the programming model provided by each system, although we are also comparing our distributed memory implementations with shared memory implementations done elsewhere.

Work queues are common component of many shared memory applications, particularly when there is irregularity in the computational cost of the tasks. On shared memory machines, the communication cost of moving tasks from one processor to another is negligible, and therefore ignored in the work queue implementations. The work queue was used in the shared memory versions of both Gröbner basis and Cholesky and was retained for the distributed memory versions. However, in the distributed memory program there is a trade-off between load balance and locality that must be carefully evaluated. In addition, a new class of tasks arise from the distributed nature of the program; these typically handle the iteration over a distributed data structure and should only be scheduled on that processor that holds the corresponding partition of the structure. We therefore divide tasks into two classes: *anyone computes*, which correspond to the tasks from the shared memory program, and *owner computes*, which are the tasks to handle data that is local to a particular processor and too expensive to move. The decision of whether a task should be scheduled using owner computes or anyone computes depends on the relative cost of communication and computation of the task, and also on the need at a global level to balance the load.

A related piece of system software is a mobile object model, a version of the Tarmac system [7] for the CM5. This system provides the programmer with a global address space as well as location transparency for objects, and is useful when objects must be relocated for better load balance. The abstraction provided by Tarmac is similar to that of the Emerald system [1], although an important aspect of Tarmac on the CM5 is that object communication is highly tuned to the architecture. The use of some global address space proves to be important in many of our codes, but the location transparency has been less useful than originally anticipated. In our applications, an object is relocated only if it is the local data of a task on the work queue. These tasks are typically independent of one another, so the local data of one is not needed by another. In short, no one cares that the data has been moved. Nevertheless, location transparency may be useful for load balancing applications like the circuit simulator, in which tasks contain a large local states with dependencies between them.

Along with these run-time facilities, we have also developed language support that provides the programmer with a hierarchical shared memory model. The key language extension is that of a global pointers, and the compiler, called DGCC, is an modification of the GCC compiler. A global pointer can be dereferenced and its target read and written using usual C syntax. Thus, a global pointer is semantically equivalent to normal (local) pointer, but has much higher access cost. Our goal in implementing DGCC was to compare the expressive power of the shared memory model and message passing. For some applications the differences are striking: one application with a distributed graph is roughly three times smaller in the shared memory style supported by DGCC than in an asynchronous message passing style. In spite of this expressiveness advantage, DGCC is not a reasonable implementation of shared memory, because the performance overhead is severe. To be true to the semantics of sequentially consistent shared memory [5], our implementation simply waits on every read and write to a global pointer. The dependence analysis that would be required to prefetch reads and buffer writes is non-trivial [9], and we know of no practical implementations that do this, particularly for C. As part of a joint project with David Culler's group, the DGCC compiler has recently been extended to hand the Split-C language, which allows the programmer to specify split-phase read and write operations. The full Split-C language has other features described in [3].

We are currently working on increasing the set of irregular applications, as well as developing different versions that will enable the comparison of other programming models. Our applications efforts have already demonstrated that hierarchical shared memory model is more expressive than explicit messages, but that further language and compiler work is needed to match the performance. We have also identified some of the distributed data structures that appear in irregular applications, as well as the need for multiple layouts and high level control constructs. We will be using these ideas in the development of a distributed data structure library, called Multipol, along with new language mechanisms for layout and control.

Acknowledgements

Most of the projects described here were class projects. The students involved were: David Bacon, Soumen Chakrabarti, Patrick Delano, Etienne Deprit, Inderjit Dhillon, Seth Goldstein, Joey Hellerstein, Arvind Krishnamurthy, Xiaoye Li, Chu-Cheow Lim, Steve Lucco, John Tse, Chih-Po Wen, and Su-Lin Wu.

References

[1] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering*, pages 65–76, January 1987.

[2] M. Heath, E. Ng, and B. Peyton. Parallel algorithms for sparse linear systems. *Parallel Algorithms for Matrix Computations*, 1990.

[3] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, T. von Eicken, K. Yelick. Introduction to Split-C. In preparation.

[4] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1991 International Conference on Supercomputing*, 1991.

[5] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[6] S. Lin, E. Kuh, and M. Marek-Sadowska. A New Accurate and Efficient Timing Simulator. In proceedings of *VLSI Design*. January, 1992.

[7] S. Lucco and D. Anderson. Tarmac: A language system substrate based on mobile memory. In *International Conference on Distributed Computing Systems*. IEEE, 1990.

[8] B. Mishra and C. Yap. Notes on Gröbner bases. *Information Sciences*, 48:219–252, 1989.

[9] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[10] J.-P. Vidal. The computation of Gröbner bases on shared memory multiprocessors. Technical Report CMU-CS-90-163, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, 1990.

[11] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, 1992.