4th Symp. on Experiences with Distributed and Multiprocessor Systems,
San Diego, CA, Sept. 1993

The Prospects for Parallel Programs on Distributed Systems

Position Paper

Michael L. Scott
University of Rochester
scott@cs.rochester.edu

Abstract

Programmers want shared memory. They can get it on special-purpose multiprocessor architectures, but the speed of technological improvements makes it difficult for these architectures to compete with systems built from commodity parts. Shared-memory parallel programming on distributed systems is therefore an appealing idea, but it isn't practical yet. Practicality will hinge on a prudent mix of compiler technology, dynamic data placement with relaxed consistency, and simple hardware support.

1. Introduction

The distinction between multiprocessors, multicomputers, and local-area distributed systems is becoming increasingly blurred. Interconnection networks are getting faster all the time, and processors (and their primary caches) are getting faster at an even higher rate. Improvements in memory and bus speed are comparatively slow. As a result, more and more parallel and distributed systems can be approximated simply as a collection of processors with caches, in which local memory is a long way away, and other processors are somewhat farther away. The more aggressive hardware designs do a better job of masking the latency of remote operations, but they cannot eliminate it completely, and their added complexity increases cost and time to market.

Given technology trends, it seems prudent to take a careful look at the benefits and costs of special-purpose inter-processor memory and communication architectures. This paper takes the position that aggressive hardware is unlikely to stay ahead of the ‘‘technology curve,’’ and that parallel programming on simpler, more distributed systems is therefore a good idea. Even with fixed technology, a customer with limited funds may not necessarily get better performance by investing in fancy memory or communication, rather than in more or faster processors.

The following sections discuss the nature of the shared-memory programming model, the relative roles of compiler technology and distributed shared memory, the design of systems that integrate the two, and appropriate hardware support. The conclusion proposes directions for future research.

2. Shared Memory

Prior to the late 1980s, almost all commercially significant parallel applications used a shared-memory programming model, and ran on machines (from Alliant, Convex, Cray, Encore, Sequent, etc.) with modest numbers of processors. These applications tended to employ parallel extensions to a sequential programming language (typically Fortran 77) or, usually for systems programming, a parallel library package called from a sequential language (typically C).

This work was supported in part by NSF Institutional Infrastructure award number CDA-8822724, and by ONR research contract number N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology — High Performance Computing, Software Science and Technology program, ARPA Order No. 8930).

In recent years, the availability of large-scale multicomputers (e.g. from Intel and NCube) has spurred the development of message-passing library packages, such as PVM [31] and MPI [10]. There is considerable anecdotal evidence, however, that programmers prefer a shared-memory interface, and many research efforts are moving in this direction. Some (e.g. the Kendall Square and Tera corporations) are pursuing large-scale hardware cache coherence. Others (e.g. the various distributed shared memory systems [25]) prefer to emulate shared memory on top of distributed hardware. Somewhere in the middle are the so-called NUMA (non-uniform memory access) machines, such the Cray T3D and BBN TC2000, which provide a single physical address space, but without hardware cache coherence.

In all these systems (with the possible exception of the Tera machine), it is important to note that a shared memory programming model does *not* imply successful fine-grain sharing or low-latency access to arbitrary data. Modern machines display a huge disparity in latencies for local and remote data access. Good performance depends on applications having substantial per-processor locality. Shared memory is a programming *interface*, not a performance model.

The advantage of shared memory over message passing is that a single notation suffices for all forms of data access. The Cooperative Shared Memory project at the University of Wisconsin refers to this property as *referential transparency* [14]. Naive patterns of data sharing in time-critical code segments may need to be modified for good locality on large machines, but referential transparency saves the programmer the trouble of using a special notation for non-local data. More important, non-time-critical code segments (initialization, debugging, error recovery), which typically account for the bulk of the program text, need not be modified at all.

3. Smart Compilers

If parallel programs are to be modified to maximize per-processor locality, how are these modifications to be achieved? One might simply leave it up to the application programmer, but this is unlikely to be acceptable. Experience with shared-memory systems of the past (e.g. the BBN Butterfly [17] and the IBM RP3 [6]) suggests that achieving enough locality to obtain near-linear speedups on large numbers of processors is a very difficult task, and the growing disparity between processor and memory speeds suggests that the difficulty will increase in future years [22].

For many of the most demanding parallel applications (e.g. large-scale ''scientific'' computations), most time-critical data accesses occur in loop-based computational kernels that perform some regular pattern of updates to multi-dimensional arrays. Compilers are proving to be very good at detecting these patterns, and at modifying the code to maximize locality of reference, via loop transformations [11, 16, 20, 28, 33], prefetching [7, 24], data partitioning and distribution [1, 2, 13, 19], etc.

Much of the recent work on parallelizing compilers, particularly in the HPF/Fortran-D/-Fortran-90 community, has focused on generating message-passing code for distributed systems, but several groups are beginning to look at compiling for per-processor locality on machines with a single physical address space [1, 12, 20, 26]. Such machines can be programmed simply by generating block copy operations instead of messages, but they also present the opportunity to perform remote references at a finer grain than is feasible with software overhead, and to load directly into registers (and the local cache), bypassing local memory.

For the sorts of sharing patterns they are able to analyze, compilers are clearly in a better position than programmers to make appropriate program modifications. Operations such as hoisting prefetches, transforming loops and re-computing bounds, accessing multiple copies of data at multiple addresses, and invalidating outdated copies require meticulous attention to detail, something that compilers are good at and programmers are not. It seems inevitable that every serious parallel computing environment will eventually require aggressive compiler technology.

4. Distributed Shared Memory

What then is the role for distributed shared memory? For regular computations on arrays, it is probably a bad idea: compilers can do a better job. But there remain important problems (e.g. combinatorial search [9] and ''irregular'' array computations [21]) for which compile-time analysis fails. For problems such as these, good performance is likely to require some sort of run-time data placement and coherence. In general, these operations will need to occur at the direction of the compiler. They will apply only to those data structures and time periods for which static analysis fails. Even then, the compiler will often be able to determine the specific points in the code at which data placement and/or coherence operations are required. As a last resort, the compiler will need to be able to invoke some sort of automatic coherence mechanism driven by data access patterns observed ''from below.''

This *behavior-driven* coherence mechanism can be implemented entirely in hardware, or it can employ a mixture of hardware and software support. Using trace-driven simulation, we compared several of the alternatives on a suite of small-scale (7 processor) explicitly-parallel programs (no fancy compiler support). (Full details appear in [4].) Figure 1 displays results for a typical application: the Cholesky factorization program from the Stanford SPLASH suite. The graph presents the mean cost (in cache cycles) per data reference as a function of the size of the coherency block (cache line or page). The five machine models represent directory-based hardware cache coherence (CC), directory-based hardware cache coherence with optional single-word remote reference (CC+), VM-based NUMA memory management (NUMA), distributed shared memory (DSM), and distributed shared memory with optional VM-based single-word remote reference (DSM+). The models share a common technology base, with high-bandwidth, high-latency remote operations. All five are sequentially consistent. In the cases where there is a choice between remote reference and data migration, the simulator makes an optimal decision.

Several conclusions are suggested by this work. First, block size appears to be the dominant factor in the performance of behavior-driven data placement and coherence systems. Second, with large blocks, it is valuable to be able to make individual references to remote data, without migrating the block. The benefit is large enough to allow NUMA memory management (with remote reference) to out-perform hardware cache coherence (without remote reference) on even 256-byte blocks. In addition, the difference in performance between DSM and DSM+ suggests that VM fault-driven remote reference facilities would be a valuable addition to distributed shared-memory systems, given reasonable trap-handling overheads. Finally, additional experiments reveal that much of the performance loss with large block sizes is due not to migration of unneeded data, but to unnecessary coherence operations resulting from false sharing (see the paper by Bolosky and Scott elsewhere in these proceedings).

5. Putting the Pieces Together

In an attempt to improve price-performance, we are pursuing the design of systems that use static analysis where possible and behavior-driven data placement and coherence where necessary, with modest hardware support. Like many researchers, we are basing our work on relaxed models of memory consistency [23]. We also expect to make heavy use of program annotations.

The goal of relaxed consistency is to reduce the number of ''unnecessary'' coherence operations (invalidations and/or updates). Generally, systems based on a relaxed consistency model enforce consistency across processors only at synchronization points. (Compilers do this as a matter of course.) Hardware implementations of relaxed consistency [18, 30] typically initiate coherence operations as soon as possible, but only wait for them to complete when synchronizing. Software implementations [8, 15, 27] are often more aggressive, delaying the initiation of the operations as well. Among other things, the delay serves to mitigate the effects of false sharing. (It also supports programs that can correctly utilize stale data, allows messages to be batched, and

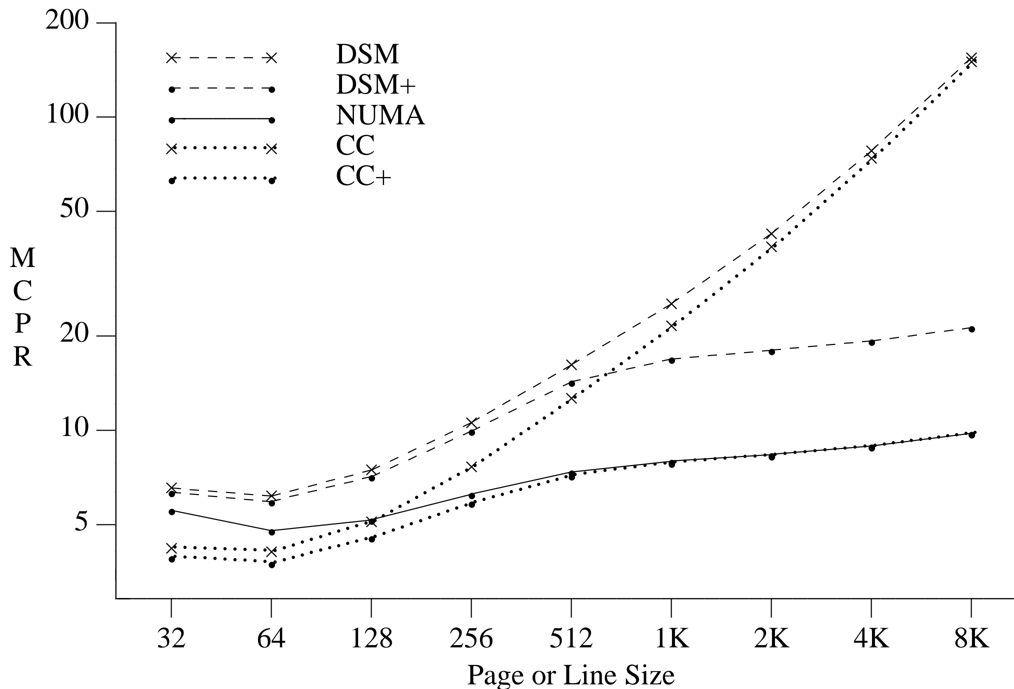

Figure 1: Mean cost per data reference for five sequentially-consistent machine models running Cholesky factorization (log scales).

capitalizes better on unilateral evictions.) Because false sharing is unintentional, it can occur at a very fine grain; limiting its impact to synchronization points can be a major win.

Figure 2 displays preliminary results of recent experiments in which we compared the performance of sequentially-consistent hardware cache coherence and NUMA memory management with that of a distributed implementation of software cache coherence with relaxed consistency. The bars in the graph report millions of execution cycles in the execution-driven simulation of a 64-processor machine. The mp3d and water applications are from the Stanford SPLASH suite; sor is a local implementation of sequential over relaxation. Each application displays distinctive characteristics. Mp3d has a lot of fine-grain sharing. The NUMA memory management system wins by freezing blocks in place and accessing them remotely. The other two systems lack remote reference; the hardware implementation moves smaller blocks, with less fragmentation and less false sharing. Sor is very well behaved. It has relatively little shared data, all of which is falsely shared between barriers. The NUMA system freezes the falsely shared data in place; the relaxed consistency system permits inconsistent local copies.[1] Water has significantly more false sharing with big blocks than with small ones, but relaxed consistency mitigates the impact of that sharing.

These experiments suggest that a system employing both relaxed consistency and remote reference would in some sense enjoy the best of all worlds, with good performance on a wide range of programs. It might even have an edge on a hardware implementation of relaxed consistency, if there are useful protocol options too complex to reasonably implement in hardware. Program annotations provide one possible source of such benefits and complexity.

[1] One should really use static compiler analysis to manage the data in sor, but we did not attempt to do so.

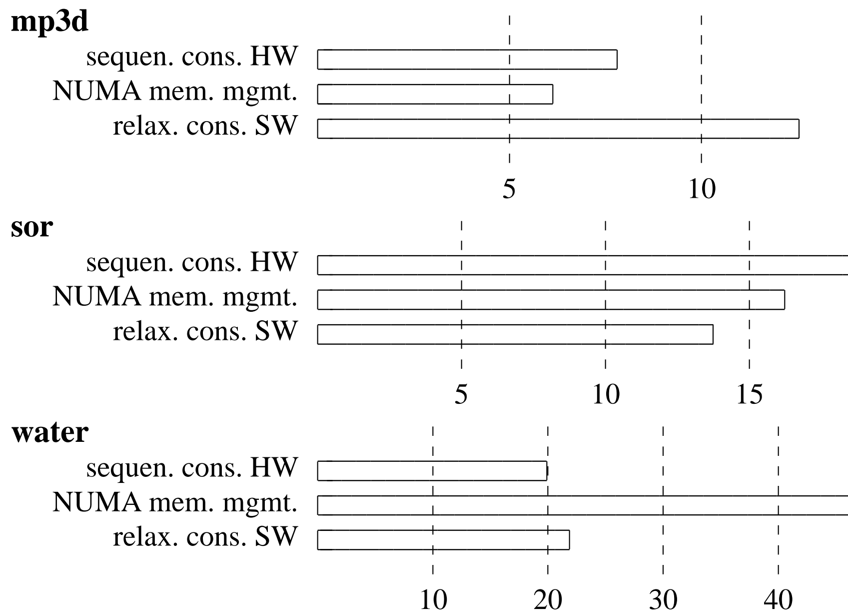

Figure 2: Execution time (in millions of cycles) for three applications and three coherence protocols on a simulated 64-processor machine.

We believe strongly in the use of program annotations to convey semantic information to a behavior-driven data placement and coherence system. Annotations could be provided by the compiler or the programmer. In either case, we prefer to cast them in a form that describes the behavior of the program in a machine independent way, and that does not change the program's semantics. Example annotations include ''this is migratory data,'' ''this is mostly read,'' ''I won't need this before it changes,'' and ''this is never accessed without holding lock X.'' Each of these permits important optimizations.[2]

6. Simple Hardware Support

Short of full-scale cache coherence, we see several opportunities to improve performance via simple hardware support. For compiler-generated data placement and coherence, fast user-level access to the message-passing hardware (as on the CM-5) is clearly extremely important. For behavior-driven data placement and coherence, our experiments testify to the importance of a remote reference facility, particularly on machines with large block sizes. This facility amounts to the use of unique, system-wide physical addresses, with the ability to map remote memory in such a way that a cache miss generates a message to the home node. Ideally, the messages would be generated in hardware, but fast page faults (which are useful for other purposes as well) would clearly be better than nothing.

When messages are received, there is a similar need to handle common operations without interrupting the processor. Reads and writes are two examples. Others include atomic fetch-and-Φ operations, and more general *active messages* [32]. In order for processors to cache local

[2] The last annotation may lead to incorrect behavior if it's wrong. This is not as nice as an annotation that affects only performance, but it's better than an annotation that may lead to incorrect behavior if *omitted*.

data that others may access remotely, it is also important that the processor and the network interface on each node be mutually coherent. (The Cray T3D has this property; the BBN TC2000 did not.)

With changes to current trends in processor design, one might envision machines with very small pages for VM-based coherence, or with valid bits at subpage granularities [5, 29]. Either of these would eliminate much of the advantage of hardware coherence, with its cache-line size blocks. For systems that trade remote reference against migration, it would also be useful for each block to have a counter that could be initialized to a given value when a mapping is created, and that would be decremented on each reference, producing a fault at zero [3].

With additional hardware support, but still short of full-scale hardware coherence, a hybrid hardware/software system like Wisconsin's Dir1SW proposal may also prove attractive [14]. It is not yet clear at what point additional hardware will cease to be cost effective.

7. Conclusion

Much of the research in distributed shared memory and NUMA memory management has occurred in an environment devoid of good compilers, or even good applications. As these become more widely available, the nature of parallel systems research is likely to change substantially. Parallel systems of the future will need to use compilers for the things that compilers are good at. These include managing both data and threads for regular computations, partitioning data among cache lines in order to reduce false sharing, invoking some coherence operations explicitly, and generating annotations to guide behavior-driven data placement and coherence. The behavior-driven techniques — distributed shared memory, NUMA memory management, etc.– should properly be regarded as a fall-back, to be enabled by the compiler when static analysis fails.

Both compiler technology and high-quality run-time systems will reduce the need for hardware cache coherence. Simpler levels of hardware support — remote memory reference in particular — are more likely to be worth the cost.

Acknowledgments

The data in figure 2 comes from the thesis work of Leonidas Kontothanassis. My thanks to Bill Bolosky and to Leonidas for their helpful comments on this paper.

Bibliography

[1] J. M. Anderson and M. S. Lam, ''Global Optimizations for Parallelism and Locality on Scalable Parallel Machines,'' *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, 21-25 June 1993.

[2] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, ''An Interactive Environment for Data Partitioning and Distribution,'' *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[3] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler and A. L. Cox, ''NUMA Policies and Their Relation to Memory Architecture,'' *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 8-11 April 1991, pp. 212-221.

[4] W. J. Bolosky and M. L. Scott, ''A Trace-Based Comparison of Shared Memory Multiprocessor Architectures,'' TR 432, Computer Science Department, University of Rochester, July 1992.

[5] W. J. Bolosky, ''Software Coherence in Multiprocessor Memory Systems,'' Ph. D. Thesis, TR 456, Computer Science Department, University of Rochester, May 1993.

[6] R. Bryant, H.-Y. Chang and B. Rosenburg, ‘‘Experience Developing the RP3 Operating System,’’ *Proceedings of the Second USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 21-22 March 1991, pp. 1-18.

[7] D. Callahan, K. Kennedy and A. Porterfield, ‘‘Software Prefetching,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 8-11 April 1991, pp. 40-52.

[8] J. B. Carter, J. K. Bennett and W. Zwaenepoel, ‘‘Implementation and Performance of Munin,’’ *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 14-16 October 1991, pp. 152-164.

[9] L. A. Crowl, M. Crovella, T. J. LeBlanc and M. L. Scott, ‘‘Beyond Data Parallelism: The Advantages of Multiple Parallelizations in Combinatorial Search,’’ TR 451, Computer Science Department, University of Rochester, April 1993.

[10] J. J. Dongarra, R. Hempel, A. J. G. Hey and D. W. Walker, ‘‘A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment,’’ ORNL/TM-12231, October 1992.

[11] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li and D. Padua, ‘‘Restructuring Fortran Programs for Cedar,’’ *Proceedings of the 1991 International Conference on Parallel Processing*, V. I, Architecture, August 1991, pp. 57-66.

[12] G. Fox, S. Ranka and others, ‘‘Common Runtime Support for High-Performance Parallel Languages,’’ First Report, Parallel Compiler Runtime Consortium, July 1993. Available from the Northeast Parallel Architectures Center at Syracuse University.

[13] D. Gannon, W. Jalby and K. Gallivan, ‘‘Strategies for Cache and Local Memory Management by Global Program Transformations,’’ *Journal of Parallel and Distributed Computing 5* (1988), pp. 587-616.

[14] M. D. Hill, J. R. Larus, S. K. Reinhardt and D. A. Wood, ‘‘Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors,’’ *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 12-15 October 1992, pp. 262-273.

[15] P. Keleher, A. Cox and W. Zwaenepoel, ‘‘Lazy Consistency for Software Distributed Shared Memory,’’ *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.

[16] K. Kennedy, K. S. McKinley and C.-W. Tseng, ‘‘Interactive Parallel Programming Using the ParaScope Editor,’’ *IEEE Transactions on Parallel and Distributed Systems 2*:3 (July 1991), pp. 329-341.

[17] T. J. LeBlanc, M. L. Scott and C. M. Brown, ‘‘Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor,’’ *Proceedings of the First ACM Conference on Parallel Programming: Experience with Applications, Languages and Systems*, 19-21 July 1988, pp. 161-172.

[18] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta and J. Hennessy, ‘‘The DASH Prototype: Implementation and Performance,’’ *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.

[19] J. Li and M. Chen, ‘‘Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays.,’’ Technical Report, Yale University Department of Computer Science, 1989.

[20] W. Li and K. Pingali, ‘‘Access Normalization: Loop Restructuring for NUMA Compilers,’’ *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 12-15 October 1992, pp. 285-295.

[21] S. Lucco, ‘‘A Dynamic Scheduling Method for Irregular Parallel Programs,’’ *Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation*, 17-19 June 1992.

[22] E. P. Markatos and T. J. LeBlanc, ''Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance,'' TR 420, Computer Science Department, University of Rochester, May 1992.

[23] D. Mosberger, ''Memory Consistency Models,'' *ACM SIGOPS Operating Systems Review 27*:1 (January 1993), pp. 18-26. Relevant correspondence appears in Volume 27, Number 3; revised version available Technical Report 92/11, Department of Computer Science, University of Arizona, 1993.

[24] T. C. Mowry, M. S. Lam and A. Gupta, ''Design and Evaluation of a Compiler Algorithm for Prefetching,'' *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 12-15 October 1992, pp. 62-73.

[25] B. Nitzberg and V. Lo, ''Distributed Shared Memory: A Survey of Issues and Algorithms,'' *Computer 24*:8 (August 1991), pp. 52-60.

[26] D. Padua and R. Eigenmann, ''Polaris: A New Generation Parallelizing Compiler for MPPs,'' Technical Report 1306, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1993.

[27] K. Petersen and K. Li, ''Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support,'' *Proceedings of the Seventh International Parallel Processing Symposium*, 13-16 April 1993.

[28] C. D. Polychronopoulos and others, ''Parafrase-2: A Multilingual Compiler for Optimizing, Partitioning, and Scheduling Ordinary Programs,'' *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[29] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis and D. A. Wood, ''The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers,'' *Proceedings of the 1993 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 10-14 May 1993.

[30] G. Shah and U. Ramachandran, ''Towards Exploiting the Architectural Features of Beehive,'' GIT-CC-91/51, College of Computing, Georgia Institute of Technology, November 1991.

[31] V. S. Sunderam, ''PVM: A Framework for Parallel Distributed Computing,'' Technical Report ORNL-TM-11375, Oak Ridge National Laboratories, September 1989.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein and K. E. Schauser, ''Active Messages: a Mechanism for Integrated Communication and Computation,'' *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.

[33] M. E. Wolf and M. S. Lam, ''A Loop Transformation Theory and An Algorithm to Maximize Parallelism,'' *IEEE Transactions on Parallel and Distributed Systems*, October 1991, pp. 452-471.