

A Data Locality Optimizing Algorithm

Monica S. Lam
Computer Systems Laboratory
Stanford University

1. INTRODUCTION

“A Data Locality Optimizing Algorithm” was one of the first papers published as part of the SUIF parallelizing compiler research project, which lasted from 1989 to 2001. The main research objective of the SUIF project was to improve data locality for uniprocessors and to maximize parallelism and minimize communication for multiprocessors. We focused on two approaches: loop transformations on kernels and interprocedural analysis to find coarse-grain parallelism. SUIF stands for the Stanford University Intermediate Format, and the compiler infrastructure developed has been made publicly available and used in many compiler research projects.

This paper presented a loop transformation algorithm to enhance data locality for uniprocessors and multiprocessors. I would not have guessed in 1991 when this paper was published that it would take ten years before we arrived at a solution that I felt satisfactorily solved the complete problem. I would like to take this opportunity to describe our discoveries along this journey.

Michael Wolf and I started our research with the goal of improving data locality on uniprocessor machines by automating blocking. We found ourselves solving the parallelization problem before we came up with our data locality algorithm. Both our algorithms for parallelism and data locality use unimodular transforms, which we developed to unify loop interchange, reversal and skewing. Our unimodular transformation algorithms are applicable only to perfectly nested loops and are built upon the imprecise abstraction of direction vectors. Jennifer Anderson and I studied the problem of automatically decomposing computation and data across multiprocessors, focusing on the minimization of communication across sequences of loops. Amy Lim and I revisited data locality and parallelization with the goal of handling general loop structures. We generalized unimodular transforms to affine partitioning, which unifies in addition the techniques of fusion, fission, reindexing, scaling and statement reordering. Operating directly on access functions in the code instead of direction vectors, our parallelization algorithm is provably optimal in maximizing parallelism and minimizing communication. This approach also generalizes blocking and array contraction, two important optimizations for locality.

I wish I could claim that I have carefully broken down this challenging problem into a ten-year research plan. Instead, we had simply picked pressing research problems at the time and provided the best solution we could at each step. We discovered the structure of the problem part by part as we solved each subproblem. In fact, I think that the success of this research rested partly on the fact that each step along the way addressed some real-life problem. This

helped us focus on important issues and ensured that the basic approach was correct.

2. SYSTOLIC ALGORITHMS

Our approach to loop transformations was inspired by systolic algorithms, which were intended to be implemented directly in silicon. Over a hundred papers have been written on systolic algorithms showing that many numerical algorithms can be mapped onto regular arrays of processing units communicating in a simple and regular manner. Most of these numerical algorithms are trivially parallelizable and the challenge in systolic array design is in minimizing communication between processors, which of course is also the key to effective parallelism on multiprocessors.

Not only did systolic array research teach us what efficient parallel algorithms look like, the design methodology that emerged provided important insights to automatic parallelization. It was shown that a systolic computation can be represented as an index set and its data dependences can be represented as distance vectors between points in the index set. Systolic array designs can be captured by geometric transforms that project the index sets representing the computations onto axes representing time and space. Dependences must point forwards in time for the design to be valid, and cross-processor dependences represent communication. This geometric model forms the basis of our work on unimodular transforms.

3. LOCALITY AND PARALLELISM

What has data locality got to do with parallelism? In each of my attempts to optimize data locality, first with Michael Wolf then with Amy Lim, we ended up finding an algorithm for parallelization first before solving the locality problem. We did not realize that parallelism is a special case of locality optimization.

Consider the special case where a computation is made up of totally independent threads of operations that share no common data, and that the target machine has only single-word data cache lines. Since all data reuse happens within a single thread, clearly executing these threads sequentially would maximize the advantage of data reuse. That is, to maximize data locality, we must first tease apart all the independent threads of computation, which is precisely the basic parallelization problem. Data locality optimization is more complicated as it must also consider the sharing of common input data and multiple-word cache lines.

Moreover, if there is no parallelism in the program, all operations must be sequentially ordered. Thus, parallelization analysis is also useful for locality optimization in that it identifies the opportunities for code transformations.

4. FROM UNIMODULAR TO AFFINE

The following outlines the research that led us from unimodular transforms to affine partitioning, highlighting the important lessons we learned along the way. All the techniques described here are applicable only to codes that access arrays using indices that are affine functions of enclosing loop indices.

4.1 Unimodular Transforms

Blocking, also known as tiling, was a hot topic in 1991. Fast on-chip floating-point units had just started to appear in general-purpose microprocessors, making them attractive as computation engines for numerical applications. However, unlike vector machines whose memory subsystems are designed to support scientific computing, microprocessors rely on caches which have been designed for integer applications. Blocked algorithms, originally developed by numerical analysts to minimize disk accesses, are found to be effective for machines with caches. Thus, blocking helps establish microprocessors as a lower cost alternative to vector machines.

Most of the work on loop transformations at the time was based on pairwise source-to-source transformations developed for vectorization. Influenced by systolic arrays, Michael Wolf and I modeled transformations of perfect loop nests as unimodular transforms on the iteration space, whereby unifying the transforms of loop interchange, reversal and skewing[9]. The domain of systolic algorithms was limited to computations whose dependences can be represented as distance vectors, and can be executed in $O(n)$ time where n represents the number of iterations in each loop. We generalized the domain to include all perfect loop nests, augmenting distance vectors with direction vectors from the vectorization literature. In this model, time may be multi-dimensional: a transform is considered legal if and only if the transformed dependences are lexicographically positive.

Both systolic synthesis and vectorization represent data dependences by the differences between related iterations, and optimize the code according to the principle that dependence vectors must not cross iteration boundaries in a parallel loop. Direction vectors, while grossly inaccurate, are adequate for unimodular transforms for perfect loop nests. They are too imprecise, however, for the more general transformation of affine partitioning on arbitrary loop nests.

The most important and lasting contribution this work made was the concept of a canonical form for parallelism. In this representation, each fully permutable subnest is made as large as possible, starting with the outermost nest. (A sequential loop is trivially fully permutable.) Each such subnest can be wavefronted, blocked, and executed in at most $O(n)$ time, where n is the number of iterations in one loop. This hierarchical representation was instrumental to the development of our parallelization algorithm based on affine partitioning.

Our paper “A Data Locality Optimizing Algorithm” presented an automatic blocking algorithm for perfect loop nests on uniprocessors and multiprocessors[8]. The paper’s most important contributions are (1) a mathematical model for evaluating data reuse in affine data access functions and (2) the basic principle that blocking exploits reuse in multiple dimensions. Experimental results suggest that our proposed heuristics work quite well as long as the loops are perfectly nested. Unfortunately, many codes require optimization across loop nests.

The industry at the time was quite eager to adopt results from compiler research that improved locality. The fact that a matrix multiplication routine was included in spec92, a popular benchmark used to evaluate processor performance, probably helped raise the industry’s interest on blocking. Michael Wolf joined SGI and

implemented a version of unimodular transforms and blocking in the SGI compiler. Many other commercial compilers implemented some form of a blocking algorithm.

4.2 Data and Computation Decompositions

In the early part of the 1990s, many multiprocessor architectures, with shared or distributed address spaces, were commercially available. Examples of distributed memory machines include the Cray T3E, the IBM SP2, the Intel Paragon and Thinking Machine’s CM-5; examples of shared memory machines include the Digital AlphaServer and the Silicon Graphics Power Challenge. Minimizing communication between processors is critical to developing efficient programs for these machines. Many companies and universities worked together to create High-Performance Fortran (HPF), a Fortran-90 dialect augmented with user-specified data decompositions. HPF compilers must automatically determine, based on the data decompositions, how the computation should be decomposed across the processors and generate the necessary communication code.

Jennifer Anderson and I recognized that data and computation decompositions are inherently tied together and should be solved at the same time. By the time users can specify the data decompositions properly, they must have already figured out where the parallelism is. Furthermore, it may be simpler to derive both data and computation decompositions automatically instead of having to handle all the data decompositions that can be specified.

Our algorithm[1] to find data and computation decompositions has two steps. It first applies unimodular transforms to find all the parallelization opportunities in each loop nest. Second, it chooses between the available ways to parallelize the loops so as to minimize communication. This two-step approach makes the algorithm simpler but suboptimal. This work’s main contributions are (1) the basic communication minimization principle that “an iteration and the data it uses should be assigned to the same processor to avoid communication”, and (2) heuristics that operate on data access functions in the code directly than direction vectors.

4.3 Affine Partitioning for Parallelism

In 1994, Amy Lim and I started working on an integrated algorithm that improves parallelism and minimizes communication of arbitrary loop nests. There were many outstanding questions to answer: How should we generalize unimodular transforms to arbitrary loop nesting structures? What is the analog of blocking, which is defined only for perfect loop nests?

Our solution to the first question is to represent the structure of an arbitrary program faithfully; it is simply not possible to find the best transform otherwise. A simple statement is treated as a loop with a single iteration. Thus, every dynamic operation is uniquely identified by the values of the indices of the enclosing loops. Loop indices can be organized as a tree reflecting the block structure of the program. A transformed program has a new tree of indices. In affine partitioning, an affine transform is created for each operation, mapping its old index values to new index values. This model encompasses all the possible combinations of unimodular transforms (interchange, skewing and reversal), fusion, fission, reindexing, scaling and statement reordering.

Our technique does not use direction vectors, but operates directly on affine data access functions. To find parallelism without synchronization, we require that “if two operations use the same data, they should be mapped to the same processor”. By eliminating the mention of data mappings from the principle Jennifer and I used, this constraint allows arbitrary data mappings and is thus more powerful. Using the Farkas lemma[7], which we learned

from studying Feautrier's work[2], we showed that we can reduce the problem of finding synchronization-free parallelism to finding all independent solutions to a set of linear constraints[3].

We solved synchronization-free parallelization quickly, but took much longer to figure out how to find parallelism that requires synchronization. From the work on unimodular transforms, we knew that it is useful to find largest outermost fully permutable loop nests. Data dependence considerations dictate that "if two operations from a sequential program use the same data, their relative ordering must be preserved". Our key discovery was to realize that if there are multiple schedules that can satisfy the data dependence constraints, then there must be flexibility in scheduling, which in turn means that the code is parallelizable. The beauty is that we can find the largest fully permutable loop nest in a manner similar to how we find the largest fully parallel loop nest: that is, by finding all the independent solutions to a set of linear constraints. The difference is that for permutability, we seek a mapping of the computation to the time axes; whereas for parallelism, we seek a mapping of the computation to the space (processor) axes.

To find all the parallelism in a program, the two basic steps are repeated recursively to create the same canonical program representation used in unimodular transforms. The difference is that we introduced a small step to break apart the strongly connected components in the body of each fully permutable subnest. Our algorithm is proven to maximize the degree of parallelism while minimizing the degree of synchronizations[4, 5].

4.4 Affine Partitioning for Locality

Almost exactly ten years after Michael Wolf and I published the data locality algorithm for perfect loop nests, Amy Lim, Shih-wei Liao and I published an algorithm for optimizing locality across arbitrary loop nests based on affine partitioning[6]. The algorithm first puts the code in the canonical form for parallelization. This code minimizes synchronization but may not have good data locality. Next, the algorithm analyzes data reuse in the program using the techniques in the original locality optimization paper. To improve locality, strongly connected components sharing reuse are fused and parallel threads sharing reuse are interleaved. The latter is achieved by a generalization of blocking for imperfect loops. Instead of "stripmine and interchange", blocking now means "stripmine and distribute". Here, outer parallel loops with reuse are stripmined and moved into the innermost loops. Moreover, by grouping all the related instructions together, this approach exposes many more opportunities for array contraction than ever possible before.

From our experiments, we see that this algorithm works well for codes as long as their array access functions are affine. The transformed code often looks significantly different from the original code and can be much more efficient than code written by a reasonably experienced application developer.

5. CONCLUDING REMARKS

This work builds upon years of research on loop transformations for vectorization and parallelization, most notably by research groups led by David Kuck at University of Illinois at Urbana-Champaign, Ken Kennedy at Rice University, Fran Allen at IBM research, Paul Feautrier, and Francois Irigoien in Ecole des Mines. We have come a long way starting with the pioneering work that discovered the many equivalent ways of expressing the same computation, interactive systems that help programmers apply transforms, and to the invention of program abstractions and automatic transformation techniques.

Programs whose data access functions are affine are special in that dynamically executed operations have very simple relation-

ships with the data elements they access. Such information is fully exploited by array partitioning. Affine partitioning solves the fundamental scheduling constraints, as imposed by a program's data dependences and the goal to eliminate communication, without introducing any imprecise abstractions. Affine partitioning appears also to be sufficiently general and powerful, as it covers most of the loop transformations discovered through years of parallelization and vectorization research.

6. ACKNOWLEDGMENT

I would like to thank John Hennessy for his help through the years; I cannot imagine a better research environment to have started my career than the one he provided. I also want to thank the many members of the SUIF group for their dedication to the project: Gerald Aigner, Saman Amarasinghe, Jennifer Anderson, Gerald Cheong, Amer Diwan, Robert French, Anwar Ghuloum, Mary Hall, David Heine, Shih-wei Liao, Amy Lim, Jing Yee Lim, Vladimir Livshits, Dror Maydan, Todd Mowry, Brian Murphy, Karen Pieper, Patrick Sathyanathan, Mike Smith, Steven Tjiang, Chau-Wen Tseng, Robert Wilson, Christopher Wilson and Michael Wolf. The funding of the SUIF project was provided by DARPA, NSF and the Department of Energy. I wish to thank my DARPA program managers, Gil Weigand, Bob Lucas and Frederica Darema, for their support of this project.

REFERENCES

- [1] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.
- [2] P. Feautrier. Dataflow analysis of scalar and array references. *Journal of Parallel and Distributed Computing*, 20(1):23–53, February 1991.
- [3] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 92–106. Springer-Verlag, August 1994.
- [4] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.
- [5] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.
- [6] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–112, June 2001.
- [7] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester, 1986.
- [8] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.
- [9] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Transactions on Parallel and Distributed Systems*, 2(4):452–470, October 1991.