Further Reading in High Performance Compilers for Parallel Computing

Michael Wolfe

November 9, 1994

To keep the book *High Performance Compilers for Parallel Computing* to a reasonable length, I left out most of the original references. The complete references are included in this document, which has one section for each chapter of the text.

1 High Performance Systems

A description of commercial and research compilers aimed at high performance and parallel computer systems can be useful to understanding what people have found important and what lessons they have learned. While specific optimizations are often published, overview papers of whole compilers are rare. General compiler techniques, such as parsing, type inferencing, and creating an intermediate form, can be found in standard textbooks such as those by Fischer and LeBlanc (1988) and Aho, Sethi, and Ullman (1986). Cohagan (1973) and Wedel (1975) describe some of the analysis used in one of the first automatic vectorizing compilers, which was targeted for the Texas Instruments Advanced Scientific Computer. Scarborough and Kolinsky (1986) describe the vectorizing compiler for the IBM 3090 Vector Facility. Coleman (1987) describes a commercial compiler for the Unisys ISP vector processor. Allen (1988) and Allen and Johnson (1988) describe the compilers for the Ardent Titan, a single-user, high-performance graphics engineering vector multiprocessor workstation, and Mercer (1988) describes the Convex compiler of the same era. Padua and Wolfe (1986) give an overview of the state of parallel compiler technology in the mid-1980s. Tjiang et al. (1991) describe a compiler that uses a single intermediate representation for both parallel and scalar optimizations. Yasue, Yamana, and Murakita (1992) discuss a Fortran compiler for a dataflow machine, which uses many of the techniques presented in this text. Zima and Chapman (1991) and Wolfe (1989b) give reasonable overviews of techniques used in parallelizing compilers of the late 1980s; Banerjee et al. (1993) also give a much briefer summary. Kennedy (1992) summarizes many of the challenges of software development, including languages and compilers, for supercomputer and parallel systems in particular.

The Parafraze Analyzer and compiler research group at the University of Illinois developed many techniques that affect modern research and commercial compilers; see Kuck et al. (1984). Parafraze-2, described by Polychronopoulos et al. (1989a, 1989b), is a successor project with even more ambitious goals. A contemporaneous project at Rice University developed the PFC parallelizing compiler (Allen and Kennedy [1984a]); this project had a direct impact
on several commercial compilers, such as at Convex (Mercer [1988]) and IBM (Scarborough and Kolsky [1986]). PTRAN was developed at IBM's T. J. Watson Research Laboratory around the same time (Allen et al. [1987, 1988a]). Together, these three research efforts developed many of the analysis methods on which modern parallel compiler research and development are based.

Several interactive tools have been developed in research environments. Smith and Appelbe (1988, 1989) describe PAT, an interactive parallelizing assistant tool. ParaScope is another interactive parallelizing tool, described by Balasundaram et al. (1989), Kennedy, McKinley, and Tseng (1991a, 1991b) and Cooper et al. (1993); it is an outgrowth of the PFC project at Rice University. Irigoin, Jouvelot, and Triolet (1991) discuss the PIPS parallelizer, which uses extensive interprocedural analysis to find additional parallelism.

A summary of various parallel computer systems is given by Hwang (1993); he includes a chapter on parallel programming models, languages, and compilers. Another summary of the features of many high performance and parallel computers can be found in the book by Stone (1990). The terms SIMD and MIMD were introduced by Flynn (1972). One of the early parallel supercomputers was the Illiac-IV, described by Barnes et al. (1968). The first widely used, commercially successful supercomputer was the Cray-1, described in an overview by Ruset (1978). Three commercial supercomputer projects in the late 1970s are presented by Kozdrowicki and Theis (1980). Several supercomputers of the mid-1980s are discussed by Stevens (1986). The SPMD programming model was introduced by Damera et al. (1988) for shared-memory parallel programming. A good summary of issues related to modern computer architecture is found in the book by Hennessy and Patterson (1990).

Current high performance and parallel computer system designs can often be found in the proceedings of various conferences, such as the annual International Conference on Parallel Processing, the International Symposium on Computer Architecture, the Supercomputing conferences, or the Compcon meeting. Lundstrom and Barnes (1980) describe the Burroughs design for a Flow Model Processor, a large multiprocessor. Gottlieb et al. (1983) and Edler et al. (1985) discuss the design of the Ultracomputer, a large-scale shared-memory multiprocessor designed at New York University. Gajski et al. (1983) describe the Cedar, a hierarchical multiprocessor designed and constructed at the University of Illinois; Eigenmann et al. (1990a, 1990b) summarize the Fortran compiler for the Cedar. Behr, Giloi, and Mühlenbein (1986) discuss design of the Suprenum machine, a parallel supercomputer designed and constructed in Germany. Pfister et al. (1985) describe the IBM RP3 prototype scalable shared-memory multiprocessor. The Alliant FX/8, a shared-memory vector multiprocessor aimed at the cost- and performance-conscious market, is described by Perron and Mundie (1986). The Convex C-1, a vector uniprocessor marketed around the same time, is discussed by Wallach (1986). The DASH project constructed a prototype large-scale cache-coherent shared-memory multiprocessor, described by Lenoski et al. (1992, 1993). A high level view of the Thinking Machines Connection Machine CM-5, a modern highly, parallel supercomputer, is given by Hillis and Tucker (1993).
2 Programming Language Features

A history of the early Fortran language and compilers is given by Backus (1981); several interesting comments are made there, including some from users who were unhappy with the speed of the compiler, and an excerpt from a November 1954 Fortran Preliminary Report stating that “...Fortran should virtually eliminate coding and debugging...” (relative to machine language programming, of course). Fortran 90 programmers can find a description of the language by Metcalf and Reid (1990), whereas compiler writers may want the complete details by Adams et al. (1992). High Performance Fortran is fully described in the handbook by Koelbel et al. (1994), which borrows a great deal from the efforts of the compiler for the German Supremum machine (Zima, Bast, and Gerd t [1988]) and from the Fortran D project at Rice University (Hiranandani, Kennedy, and Tseng [1992b]). Both the “traditional” C language and ANSI standard C are described by Harbison and Steele (1991). C++ is described by Stroustrup (1991) and Ellis and Stroustrup (1990), and in a primer by Lippman (1991). Pros and cons of using C for numerical computing are discussed by MacDonald (1991).

The Blaze language (Melstrom and Rosendale [1987]) had a parallel loop, with copy-in copy-out semantics, like our dopar. A type of loop in which the iterations can be executed in any order is described by Wolfe (1992b); the same loop model is used by Solworth (1992) to generate better parallel code for operations on linked lists. The doacross model of parallel loop execution is described by Cytron (1986). Yang et al. (1993) give a method to integrate a parallel loop with reduction operations to improve performance. Albert et al. (1988, 1991) describe the use of a forall construct for programming parallel machines in data-parallel style. Philipson and Tichy (1992) use a forall statement in an extended Modula-2 language with two flavors: synchronous, like the one we present here, and asynchronous, which is more like a dopar. Midkiff, Padua, and Cytron (1989, 1990) discuss the problems that can arise when trying to analyze and optimize programs with poorly designed parallel constructs. The general parallel loop model discussed in this chapter is presented more extensively by Li and Wolfe (1994).

Refined Fortran and Refined C were designed as extensions to Fortran and C in which it is possible to express potential aliasing relationships that the compiler can use to identify more parallelism; the extensions are described by Dietz and Klappholz (1986) and by Klappholz, Kallis, and Kong (1989, 1989). In the 1980s and early 1990s, an ad hoc group of parallel computer system vendors and academic researchers tried to design a shared-memory computational model and extensions to Fortran; the group was called the Parallel Computing Forum (PCF), and the proposed extensions were called PCF Fortran (Leasure [1991]). A summary of various parallel Fortran language dialects is presented by Guzzi et al. (1990), and general language topics for parallel programming are presented by Pancake (1993). Karp and Babb (1988) describe the use of 12 Fortran dialects for nine parallel machines to solve a simple problem; Babb (1988) describes the process that generated these 12 languages in detail.

Parallel languages are often developed for particular machines, such as for the Thinking Machines Connection Machine (Thinking Machines Corporation [1991]) or Sequent parallel processor (Sequent Computer Corp. [1986]). Sometimes fixed hardware characteristics, such as the number of processors, are visible in the language, as in Glynpir, the first high level language for the Iliac IV (Lawrie et al. [1975]), and DAP-Fortran (Flanders et al. [1991]). The C*
language was developed as a SIMD, data-parallel language for the Connection Machine; Quinn, Hatcher, and Jourdenais (1988, 1990) and Hatcher and Quinn (1991) describe a compiler for C* that is targeted at MIMD shared-memory or message-passing multiprocessors. Array languages similar to APL are often explored for parallel programming, as discussed by Ortiz, Pinter, and Pinter (1991).

Advocates of higher level programming languages have always been hampered by the relatively weak performance of these languages on numerical codes compared to sequential languages like Fortran or C with aggressive optimizing compilers. Gopinath and Hennessy (1989) describe one of the important optimizations for functional languages, copy elimination. A great deal of work in the SISAL project has attempted to address the performance complaint (Feo, Cann, and Oldehoeft [1990]). Kuck, McGraw, and Wolfe (1984) debate whether functional programming languages can replace Fortran as the premier high performance language. Cann and Feo (1990) describe a small experiment that shows that optimized SISAL code can approach and even beat the performance of compiler-optimized Fortran code; Cann (1991, 1992) follows this up with more evidence that performance is no longer an obstacle for declarative programming languages. Haskell and Crystal are declarative languages that contain primitives to support scientific computation; Anderson and Hudak (1990) describe compiler analysis for Haskell, while Chen (1986b) and Chen, Choo, and Li (1988) describe the Crystal compiler. In both cases, many of the analysis techniques are similar to those described in this text.

Sometimes particular language rules have unexpected side effects with respect to optimization. Cooper, Hall, and Torczon (1992) describe a situation where inlining in a Fortran compiler weakened the analysis. In this case, the program passed parts of a single array to distinct reference arguments; since the subroutine defined one of the arguments, the compiler used the Fortran rule that the arguments must not be aliased. After inlining the subroutine, this rule no longer applied, and the compiler analysis was less precise.


3 Basic Graph Concepts

Any general textbook on discrete math will cover basic concepts of sets and logic, such as the books by Stanat and McAllister (1977) or Tremblay and Manohar (1975). Logic and discrete math are described by Gries and Schneider (1993) and Stanat and McAllister (1977). Aho, Hopcroft, and Ullman (1974) give many graph algorithms that are in common use in compilers. Cormen, Leiserson, and Rivest (1990) describe and analyze many algorithms that are in common use; they give an alternate algorithm for finding strongly connected components for a directed graph. The algorithm to find strongly connected components shown here is due to Tarjan (1972), who proves it correct; it is also described by Aho, Hopcroft, and Ullman (1974).

Control flow graphs and dominators have long been used in compilers. Lengauer and Tarjan (1979) describe a fast and efficient algorithm to find the dominator tree of a CFG. Ferrante and Ottenstein (1983, 1987) define the concept of control dependence in terms of postdominators, and use the control depen-
dence graph to replace the control flow graph; they use an irreflexive definition of postdominance, and a different, but equivalent, definition of control dependence. Cytron et al. (1989, 1991) describe the algorithm for finding dominance frontiers, and show the relationship between the postdominance frontiers and control dependence graph.

Sarkar (1989) uses control dependence relations to reduce the execution cost of inserting profiling counters; one counter is used for two (or more) basic blocks that are control-equivalent, and one counter is used for both the true and false paths of conditionals. The PTRAN compiler uses the forward control dependence graph (without back edges) to simplify program analysis (Cytron, Ferrante, and Sarkar [1989]). Our definition of dominance frontier is slightly different than that used in previous work. Cytron, Ferrante, and Sarkar (1990) show a more concise algorithm for finding dominance frontiers and control dependence relations.

The definition we use for natural loop is equivalent to Tarjan's definition of interval (Tarjan [1974]). The reverse postorder numbering is equivalent to the depth-first ordering used to speed up certain data-flow analysis algorithms (Hecht [1977]).

4 Review of Linear Algebra

Many of the techniques reviewed here for dealing with integer matrices, as well as more details on integer programming, are discussed in greater detail in the books by Nemhauser and Wolsey (1988) and Schrijver (1986). Banerjee (1993) also discusses these topics in the context of restructuring compilers. General integer programming tries to find a solution that optimizes some cost function in a region bounded by linear inequalities. The techniques in which we are interested have no cost function, so we may be able to take advantage of simpler solvers.

Euclid’s algorithm to find the greatest common divisor is the first mathematical algorithm, developed several thousand years ago. Lamé (1844) showed that the number of iterations for two integers is bounded by five times the number of digits in the smaller number. Bradley (1970) discusses finding the gcd of a sequence of numbers, along with finding a solution to the gcd equation.

Fourier-Motzkin projection is discussed by Williams (1976, 1983), Dantzig and Faves (1973), and Duffin (1974). Aucourt and Irigoin (1991) use Fourier-Motzkin projection to find loop limits after restructuring (reordering) the loops. Kuhn (1980) uses Fourier-Motzkin projection to solve data dependence problems. Maydan, Hennessy, and Lam (1991) and Wolfe and Tseng (1992) both reduce the number of unknowns by first eliminating equalities, then use Fourier-Motzkin to eliminate inequalities, in data dependence problems. We will show how to use our algorithms to address these problems.

The extreme value method for rectangular regions was used by Banerjee (1976) to solve data dependence problems. Banerjee (1988a) extended it to nonrectangular regions, and also describes the algebraic techniques to reduce integer systems of equations. Another method to reduce integer systems of equations was developed by Pugh (1991b, 1992a), who introduces a new operator (mod, pronounced “mod-hat”), which allows the system to reduce more quickly. Pugh also introduced the concept of a dark shadow. His system, called the Omega Test, solves general integer programming problems as well as more
complicated Presburger formulas. In the domain studied in this chapter, when the Fourier-Motzkin projection technique is inexact and the dark shadow test fails to find a solution, Pugh’s system can “split” the problem into a set of problems, one of which must have an exact solution for the original system to have a solution.

Feautrier (1988a) describes a scheme called parametric integer programming to solve integer programming problems that arise in the framework of a compiler.

5 Data Dependence

The concepts of data dependence have been well-known for quite some time (Bernstein [1966]); data dependence distance vectors have long been used in the systolic array community to generate regular communication patterns (Karp, Miller, and Winograd [1967]). Allen (1987) coined the terms dependence level, loop carried and loop independent dependence. Kuck et al. (1981) and Allen, Callahan, and Kennedy (1987) show how to use dependence graphs for parallelization and other optimizations. Before control dependence relations were widely used, some compilers handled control conditions by converting them to data dependence relations (Allen et al. [1983]).

The distinction between address-based and value-based dependence relations arises frequently. Brandes (1988) uses the term direct dependence for value-based dependence; using direct dependences, more aggressive dead code elimination can be done, and variable renaming can be used to eliminate false dependence relations.

Banerjee (1988b) introduces basic concepts of dependence testing. Data dependence levels are introduced by Allen (1987). Wolfe (1991b) describes the relations of various data dependence abstractions, such as dependence level, direction vector, distance vector, and semantic information such as reduction directions, and shows the strengths and weaknesses of each. Pugh (1992b) challenges some traditional definitions of dependence distance as ill-defined; we have attempted to address his concerns here.

Cartwright and Felleisen (1989) discuss the meaning of data dependence, pointing out the difference between preserving the actual behavior and the observable behavior of correct, terminating programs. Pingali et al. (1990, 1991) and Johnson, Li, and Pingali (1991) describe an intermediate representation called dependence flow graphs, which subsumes data and control dependence graphs and has executable semantics.

The program dependence graph is described more fully by Ferrante, Ottenstein, and Warren (1987); their description uses region nodes to collect graph nodes with identical control dependence predecessors. Horwitz, Prins, and Reps (1988) show that PDGs are adequate, in that they distinguish between programs that are not equivalent.

6 Scalar Analysis with Factored Use-Def Chains

The algorithms for finding the FUD chains (equivalent to the static single assignment, or SSA, form) are presented by Cytron et al. (1989, 1991). A factored SSA form is described by Choi, Cytron, and Ferrante (1994), which closely resembles FUD chains. An asymptotically faster algorithm for placing φ-terms is given by Cytron and Ferrante (1993).
An alternative "sparse" data-flow graph representation for program analysis is shown by Beck and Pingali (1990), Beck, Johnson, and Pingali (1991), and Johnson and Pingali (1993). The program dependence web (Ballance, Mc-cabe, and Ottenstein [1990]) is another representation that encompasses the advantages of several others. One step to build the program dependence web is construction of the gated single assignment form, where $\phi$-terms are replaced by gates with a predicate to choose the appropriate operand; Havlak (1993) discusses how to construct a version of the gated single assignment form. Choi, Cytron, and Ferrante (1991) explore using sparse data-flow graphs for solving any data-flow problem.

Jouvelot and Debbenei (1989) describe an analysis technique that finds induction variables, invariant expressions and other generalized reduction operations in loops. Ammourgellat and Harrison (1990) use abstract interpretation with a set of recurrence templates to find induction variables and other recurrence operations. Haghhighat and Polychronopoulos (1992) use symbolic interpretation to find induction variables and perform other scalar analysis; they also use symbolic analysis to improve the results of parallelization (1993). Wolfe (1992a) describes the method presented here for finding induction variables using FUD chains; these advanced methods can also identify and classify a variety of other interesting patterns that are useful for dependence analysis. Note that we are interested in finding induction variables for use in dependence analysis, which is different from the goal of finding induction variables for strength reduction (Allen, Cocke, and Kennedy [1981]).

The constant propagation algorithm shown here was developed by Wegman and Zadeck (1985, 1991). An interprocedural constant propagation algorithm is described by Callahan et al. (1986). Handling aliases, such as pointer aliases, in data-flow analyses based on FUD chains or SSA form is discussed by Cytron and Gershbein (1993). Srinivasan, Hook, and Wolfe (1993) explore construction of the FUD chains or the SSA form in programs with explicitly parallel constructs.

7 Data Dependence Analysis for Arrays

Initial work in array dependence testing was done in the context of systolic array generation for uniform recurrence equations (Karp, Miller, and Winograd [1967]). Many of the ideas were also used in early compiler work before data dependence concepts were fully developed (Firestone [1973]). The extreme value test, developed by Banerjee (1976), was one of the first attempts to apply serious mathematics to the problem of dependence between general array references; its use can also be found in the paper by Banerjee et al. (1979). The gcd and extreme value test for direction vectors are described by Wolfe (1982) and Wolfe and Banerjee (1987). The exact test for single variables is described by Banerjee (1979). The direction vector hierarchy is described by Burke and Cytron (1986). Linearization of array subscripts is proposed by Banerjee et al. (1979). The idea of a self-loop was used in the PTRAN compiler (Burke and Cytron [1986]).

Wallace (1988) describes using limited Gaussian elimination on the dependence matrix to determine whether the dependence equation has a unique solution and to diagonalize the dependence matrix so the generalized gcd test can be used; if those fail to determine independence, he suggests constructing a constraint matrix (adding slack variables) and attempting to find a solution using a variation of the simplex method. Integer and linear programming techniques
are also discussed by Li and Yew (1989).

Li, Yew, and Zhu (1989, 1990) describe the \( \lambda \)-test for multidimensional array subscript dependence analysis, which constructs various linear combinations of the dependence equations for each dimension and solves each linear combination; in certain circumstances, the \( \lambda \)-test can be shown to be exact (Li [1989]). Goff, Kennedy, and Tseng (1991) describe the dependence analysis developed for the PFC and ParaScope tools; these tools substitute information from one equation to simplify another, and take advantage of the special structure of the equations to separate the system into smaller systems. Grunwald (1990) compares this analysis with the \( \lambda \)-test, both of which find combinations of dependence equations to eliminate unknowns. Maslov (1992) discusses splitting dependence equations.

The extreme value test was used in a great deal of early work, including that of Allen (1987) and Wolfe and Banerjee (1987), where it is usually referred to as Banerjee's test. Banerjee (1988a) describes the extensions to triangular loop limits. Psarris, Klappholz, and Kong (1991) and Klappholz, Psarris, and Kong (1990) argue that in practice, the extreme value test is accurate for single dependence equations, even when direction vector constraints are considered (Psarris [1992]). Kong, Klappholz, and Psarris (1990, 1991) and Psarris, Kong, and Klappholz (1991, 1993) describe the I-test, a refinement of the extreme value test that improves its accuracy and incorporates the gcd test as well. The I-test finds lower and upper bounds for the dependence function \( L \leq a_1 i_1 + a_2 i_2 + \cdots + a_n i_n \leq U \), and checks whether there is a multiple of \( q = \gcd(a_1, a_2, \ldots, a_n) \) in the range \([L : U]\). If there is not, there is no solution to the dependence equation; if there is, then one of the unknowns is eliminated (as with the extreme value test) and the technique iterates. Petersen and Padua (1992, 1993) describe experiments to evaluate the relative accuracy of various dependence tests, and the importance (or lack thereof) of improving test accuracy.

Kuhn (1980) proposed using Fourier-Motzkin projection for dependence testing, though he did not first consider removing equalities to reduce the number of unknowns. The generalized gcd test was introduced by Banerjee (1988a). Wolfe and Tseng (1992) propose to follow the generalized gcd test with some simple analysis to find exact dependence distances, when possible, and with Fourier-Motzkin projection otherwise; they eliminate the unknowns in a fixed order, the most deeply nested loop first. Maydan, Hennessy, and Lam (1991) describe a series of techniques to eliminate inequalities in the dependence system (including a loop residue step [Shostak (1981)] not presented here) that empirically determine all dependence relations exactly. Eisenbeis and Sogno (1992) discusses an approach based on integer programming techniques, including some techniques to improve the run-time performance. The Omega test for data dependence is described by Pugh (1991b, 1992a); it includes additional analysis to find an exact solution even when Fourier-Motzkin projection gives inexact integer results.

Nontrivial dependence systems include unknown symbols other than loop induction variables, and have been studied from many points of view. Lichnewsky and Thomassett (1988) describe a dependence testing scheme for a vectorizing compiler that uses extensive symbolic manipulation, including symbolic assertions in the program. Haghighat (1990) also explores handling unknown symbols in dependence analysis. Li and Chen (1990) point out that unknown loop limits can be treated like additional variables in the dependence system, and that additional constraints can sometimes be found in conditional expressions. Klap-
phonolz and Kong (1992) also discuss adding conditional expression constraints in the extreme value test.

The complexity of the data dependence problem was studied by Shen, Li, and Yew (1989, 1990); they also show statistics on how often different dependence tests are used in their system, and the success rate of each. See the papers by Goff, Kennedy, and Tseng (1991) and Maydan, Hennessy, and Lam (1991) for comparative studies using different sets of dependence tests. Wolfe (1993) gives guidelines on effectively engineering a data dependence program.

Compiler analysis can compute kill information for scalars precisely, but for array references the analysis is more complex; classical memory-based data dependence analysis is inherently imprecise because it ignores array kills. Ribas (1990) finds the latest assignment for each element, computing self-kills by using integer programming. Maydan, Amarasinghe, and Lam (1992, 1993) present a data-flow framework for array references, which is similar in concept to, but simpler than, the method by Feautrier (1991). Pugh and Wonacott (1992, 1993) describe methods to compute array kills by extending the Omega test to solve more general problems. Maslov (1994) describes another efficient array data-flow analysis method that uses the Omega test to reduce the problem form. Duesterwald, Gupta, and Sohia (1993) give a simpler scheme based on data-flow analysis. Kalis and Klapholz (1991) shows another scheme to integrate array references into a classical data-flow framework.

Tzen and Ni (1993b) capture the dependence relations with irregular dependence distances by computing the convex hull of the free variables for the solutions to the dependence equation.

8 Other Dependence Problems

Pointer analysis is the subject of a great deal of research. Jones and Muchnick (1981) present some of the first work to try to automatically determine the structure of the dynamic data structure from program analysis. Landi and Ryder (1991) give a classification of the general problem. Chase, Wegman, and Zadeck (1990) add a heap reference count to the storage shape graph, which counts the number of unnamed (nonvariable) pointers to each object; if the heap reference count is one, then the data structure must be a simple cycle, list, or tree. Larus and Hilfinger (1988) use an alias graph instead of a storage shape graph to represent pointer aliases. The pointer analysis presented here is shown in more detail by Emami, Ghiya, and Hendren (1994).

Hendren and Nicolau (1989, 1990) discuss a path matrix representation of pointer paths to determine aliasing or dependence for programs that construct and traverse dynamic data structures. Horwitz, Pfeiffer, and Reps (1989) build on the work of Jones and Muchnick (1981). Loeliger et al. (1991) show some results of an implementation of interprocedural pointer tracking in a commercial C compiler. Smith (1991) shows the importance of pointer analysis and pointer arithmetic when compiling C for vector machines (and by extension, for parallel machines). Guarna (1988) also looks at pointers as they are used in loops. Landi and Ryder (1992), Landi, Ryder, and Zhang (1993), and Choi, Burke, and Carini (1993) discuss interprocedural pointer alias and side effect analysis. See the short article by Marlowe et al. (1993) for a discussion of two general pointer-aliasing approaches. Realizing that pointers are used to implement complex data structures that may have regular characteristics, Hendren, Hummel, and
Nicolau (1992) have developed a language for annotating programs to describe characteristics of the actual data structure that is being constructed and/or traversed with the pointers. Hummel, Hendren, and Nicolau (1992) show how these annotations can be used to improve analysis; Hummel, Hendren, and Nicolau (1994a) describe a simple language for expressing alias properties; and Hummel, Hendren, and Nicolau (1994b) show a scheme to find data dependence by a theorem-proving technique, given a set of simple axioms defined by the annotations.


Interprocedural alias analysis gives rise to subtle problems; one of these is explored by Burke and Choi (1992), and the analysis is studied in general by Cooper (1985). A fast algorithm to find aliases for formal reference arguments and global variables is described by Cooper and Kennedy (1989); the same algorithm is refined by Mayer and Wolfe (1993). The $P^2$ programming environment (Carle et al. [1987]), supports extensive interprocedural analysis (as described by Cooper, Kennedy, and Torczon (1986)) such as alias analysis, interprocedural constant propagation, recompilation analysis, linkage tailoring, and so on. Callahan (1988) uses a program summary graph, with entry, exit, call, and return nodes for each procedure and call statement, to capture flow-sensitive interprocedural data-flow information. Hall et al. (1993a) describe a general interprocedural analysis tool that can be used to solve many problems. Callahan et al. (1986) developed the interprocedural constant propagation that we present here. Grove and Torczon (1993) discuss an implementation of interprocedural constant propagation, with the costs and benefits of more powerful jump function implementations analyzed.

Tang (1993) describes a method to summarize the region of arrays that are referenced or defined by projecting the actual references from the loop iteration space onto the array dimension space.

Troilet, Irigoin, and Feautrier (1986) show how summarizing the effects of a procedure can be used to parallelize loops containing calls to that procedure. Callahan and Kennedy (1987, 1988a) summarize the array subregions that are used or modified by a subroutine call using a structure they call “regular sections.” Balasundaram and Kennedy (1989b) and Balasundaram (1990) give a simpler, more efficient scheme to address the same problem. Havlak and Kennedy (1990, 1991) describe experience with an implementation of regular section analysis and give a good summary of various regular section methods.

Li and Yew (1988a, 1988b, 1988c) describe an alternative scheme that does not summarize references, but describes the region of the array accessed by each array reference. A fast data-flow algorithm that uses rectangular regions is used by Gross and Steenkiste (1990).

Constructing the call graph in the presence of procedure parameters is discussed by Ryder (1979) if there is no static or dynamic recursion. An extension that allows recursion is presented by Callahan et al. (1987). Hall and Kennedy (1992) show the efficient but less precise method described here.
9 Loop Restructuring

Some of the transformations presented in this chapter have been well known for some time; see the descriptions by Allen and Cocke (1972) and Loveman (1976, 1977).

Automatic loop interchanging was first implemented in the compiler for the TI Advanced Scientific Computer (Cobagan [1973]); it was discussed more generally by Wolfe (1982) and Allen (1983). Simple loop interchanging was published by Allen and Kennedy (1984b), and handling nontrivial loop limits is shown by Wolfe (1986a). A more general discussion is presented by Banerjee (1989). Auncourt and Irigoin (1991) discussed using Fourier-Motzkin projection to find loop limits after restructuring (reordering) the loops.

Smith, Appelbe, and Stirewalt (1990) and Wolfe (1991a) discuss incrementally updating the dependence graph after restructuring the program. These techniques were used in Tiny, an experimental loop restructuring research tool (Wolfe [1991c]). The ParaScope interactive tool updates both the program source and the dependence information after an edit by the programmer (Kennedy, McKinley, and Tseng [1993]). Warren (1984) discusses a hierarchical representation of data dependence to support statement reordering and loop fusion.

Fission for loops containing conditionals requires computing the expression that controls the condition and saving the result in a temporary array. Several approaches have been described in the literature. If all the statements with the same flow constraints are kept in the same loop after fission, no conditional arrays are necessary. Towle (1976) and Baxter and Bauer (1989) suggest using conditional arrays only when statements with the same control constraints end up in different loops after fission. Kennedy and McKinley (1990) describe the process of loop fission when the loop contains conditionals or a loop dependent exit. Hsieh, Hind, and Cytron (1992) demonstrate loop fission using a general control flow graph and with multiple exits.

Eigenmann and Blume (1991, 1991, 1993) describe additional program transformations, aimed at allowing the generation of parallel code in more cases; in particular, these include array privatization and classifying generalized induction variables. Polychronopoulos (1987b) introduces loop coalescing with the goal of simplifying code generation and scheduling for parallel computers.

There have been many attempts to place a number of transformations in a single framework, such as a matrix formulation. Banerjee (1990) and Wolf and Lam (1990, 1991a) explore unimodular transformations for tightly nested loops. Sarkar and Thekkath (1992) and Wolfe (1991a) present similar schemes for characterizing a number of transformations, though the transformations are not composable in the same sense that matrix operations are composable. Linear loop transformations have long been used in the systolic array community to map regular algorithms onto a regular processor ensemble (Karp, Miller, and Winograd [1967], Chen [1986a]). Nonlinear and piecewise linear transformations are described by Lu (1991).

Loop skewing (Wolfe [1986b]) was developed as an alternative implementation of the wavefront (Murakoca [1971]) or hyperplane (Lamport [1974]) method of parallel execution for loops with dependence relations; the wavefront method is effectively the same as using unimodular transformations, focused on parallel execution. Circular loop skewing is presented as a loop transformation by Wolfe (1989a).

Irigoin and Triolet (1988) and Wolfe (1989c) introduce the concept of tiling
for locality and parallelism. Ramanujam (1992) gives another derivation of tiling, in terms of hyperplanes that divide the iteration space into tiles, including code generation methods. The method presented here for code generation for nonunimodular transformations is derived from the work by Li and Pingali (1992a, 1994). Barnett and Lengauer (1992) and Ramanujam (1992) also discuss nonunimodular loop transformations.

Interprocedural program transformations can also play an important role in a restructuring compiler. Procedure cloning is well known in the partial evaluation literature, where it is called _polymorphic pattern evaluation_. Cooper, Hall, and Kennedy (1990) discuss cloning in the framework of a programming environment that manages interprocedural issues. Hall, Kennedy, and McKinley (1991) describe interprocedural loop extraction and loop embedding and their use in a programming environment. Cooper, Hall, and Torezoun (1991) show results of an experiment with procedure inline substitution, including increases in object code size and compile time, and reductions in execution time.

Whitfield and Soffa (1991) have developed a language for writing optimizations, including the types of restructuring described in this chapter.

Scalar expansion can remove most anti- and output dependence relations due to scalar assignments in a loop. Array expansion can be used when arrays are assigned, but are not indexed with all the loop indices (the so-called “missing index” problem). Chen and Chang (1987) show array expansion for simple cases.

Torres et al. (1993) discuss code generation when alignment is combined with linear loop transformations by giving each statement an alignment offset from other statements.

## 10 Optimizing for Locality

McKellar and Coffman (1969) noted that using a submatrix formulation of matrix operations—essentially tiling—augmented with the proper data layout will improve memory hierarchy performance; at that time, the memory hierarchy in question was paged virtual memory. The earliest work to use automatic program restructuring to optimize for the memory hierarchy was done by Abu-Sufah (1978) and Abu-Sufah, Kuck, and Lawrie (1981). The importance of properly taking advantage of the memory hierarchy is conveyed by the design of the Level-3 BLAS (Dongarra et al. [1990]). Callahan and Porterfield (1990) show that up to half the time of some supercomputer applications can be wasted in processing cache misses. Goodman (1983) shows performance characteristics when the cache memory is used to reduce traffic between the processor and the memory. Lam, Rothberg, and Wolf (1991) discuss many aspects of optimizing for locality, such as choosing the tile size and copying data to reduce cache interference. Temam, Granston, and Jalby (1993) give other heuristics for choosing data to copy to separate, conflict-free locations.

Gannon, Jalby, and Gallivan (1987, 1988) describe early work on an automatic scheme to manage a memory hierarchy automatically in a parallel programming environment using the concept of a reference window, which is the number of loop iterations from when the data element is first reference until it is no longer needed; minimizing reference windows will improve the performance of the memory hierarchy. Eisenbeis et al. (1990) use a similar scheme to copy frequently used data into a fast local memory, and Bodin et al. (1992)
use reference windows to manage cache utilization on a uniprocessor. Nicolau (1987, 1988) uses an optimization essentially equivalent to tiling (following by fully unrolling the element loops) to optimize for locality when compiling for fine-grain parallel machines, such as VLIW machines. Wolf and Lam (1991b) distinguish between reuse and locality, and between self-reuse and group-reuse, and show a scheme to quantify reuse and to try to improve locality. Wolf (1992) uses unimodular transformations to optimize for locality and parallelism. Carr and Kennedy (1989, 1992) show the strengths and weaknesses of tiling in a compiler; they show several cases in which tiling fails, and that index-set splitting can address some of these cases.

Ferrante, Sarkar, and Thrash (1991) show compiler analysis methods to estimate how many cache lines will be needed for each reference in a loop, which can be used to guide compiler optimizations.

Gornish, Granston, and Veenendaal (1990) describe a scheme for inserting prefetch operations at the earliest point in the program after which the prefetched data is guaranteed to be used. Callahan, Kennedy, and Porterfield (1991) describe a simple but effective method for inserting prefetch operations for algorithms with regular data access patterns. Mowry, Lam, and Gupta (1992) evaluate another scheme for inserting prefetch operations with the compiler.

Prepaging is the analog in virtual systems to cache-line prefetching; Di Antoine, Filert, and Viteletti (1989) describe the results of some experiments showing the effectiveness of prepaging on programs with very large data sets.

11 Concurrency Analysis


Self-scheduling for nested loops is discussed by Tang and Yew (1986) and Fang et al. (1990). One of the first tapering scheduling methods was guided self-scheduling, developed by Polychronopoulos (1986) and described by Polychronopoulos and Kuck (1987). The factoring scheduling method is discussed by Hummel, Schonberg, and Flynn (1991, 1992), who show it to be experimentally better than simple chunk scheduling or guided self-scheduling; if the coefficient of variance of the execution time for the iterations is known, factoring can be modified to compute a more optimal batch size. Lucco (1992) discusses dynamic scheduling for loops with irregular iteration execution times. Tzen and Ni (1991, 1993a) describe trapezoidal scheduling, a simpler scheduling algorithm that gets much of the benefit of other tapering schemes, but with a simple (cheap) tapering method. Haghifat and Polychronopoulos (1993) use symbolic analysis to try to determine at compile-time how to schedule the loop iterations. The idea of using a queue of generator procedures to assign work in a parallel system was implemented in the BBN Butterfly multiprocessor. This system also allowed programs to scatter data structures across the many memory modules to reduce network contention (Bettsberg and Thomas [1986]).

The cycle shrinking optimization is due to Polychronopoulos (1987, 1988a); he also describes a scheme to strip mine a loop using dynamically chosen (at run-time) strip sizes such that the iterations in each strip are data independent, allowing the element loop to be executed in parallel. Saltz, Mirchandaney,
and Crowley (1989, 1991) describe a generalization of this where the iterations are sorted into independent groups to minimize execution time. Robert and Song (1992) describe several improvements to this approach. Ayguadé et al. (1990) show a method where each statement is scheduled independently. Shang, O’Keefe, and Fortes (1994) relate cycle shrinking to linear loop transformations. See Polychronopoulos (1988c) for a number of parallelization techniques.

An alternative to cycle shrinking, when the dependence distances are all known, is to schedule the operations to achieve as much parallelism as possible; such a scheme is described by Ayguadé et al. (1989) and Labarta et al. (1991).

Several schemes have been proposed to partition iterations so that each partition is independent of the others, and can be executed in parallel. Peir and Cytron (1987, 1989) show how to partition the loop based on unimodular transformations of the dependence matrix. Liu, Ho, and Shen (1990) use statement alignment to improve the partitions. Fang (1990) and Fang and Lu (1991) show a scheme that takes into account cache locality, and attempts to reduce or eliminate the effects of false sharing. D’Hollander (1990, 1992) uses unimodular transformations on the dependence distance matrix to reduce it to diagonal or triangular form; applying the same transformations to the iteration space relabels the iterations such that independent partitions can easily be found. Other partitioning schemes are presented by Shang and Fortes (1988) and Huang and Sadayappan (1991).

Alignment and code replication to help parallelization are discussed by Allen, Callahan, and Kennedy (1987). Fang (1990) discuss compiler analysis and code generation for parallelizing while loops (noniterative loops). Dietz (1988) handles while loops by splitting out the part of the loop that determines the exit condition, counting the number of iterations, and then executing the remainder of the loop as a countable loop. A similar approach is described by Wu and Lewis (1990). Lu and Chen (1991) discuss parallelization of loops with pointers.

Kogge and Stone (1973) use recursive doubling techniques to solve n-th order linear recurrences in parallel. Callahan (1991) discusses certain forms of recurrences that can be solved efficiently and in parallel. Amargullat and Harrison (1990) use abstract interpretation with a set of recurrence templates to detect and classify simple recurrence relations in loops. Radon and Feautrier (1993) can find and classify a wide range of recurrences in loops, involving both scalars and arrays.

For some algorithms, the execution order of the iterations is immaterial. An example is a set of accumulations; the order in which the accumulations occur is unimportant, if we assume that the roundoff error differences don’t matter. With such an algorithm, additional parallelism can be found by allowing any iterations that don’t have def-def conflicts to execute in parallel. Graph coloring techniques are used by Hege and Stiehben (1991) to find the nonconflicting iterations.

For many scalar compilers, the optimization order is chosen when the compiler is written; for instance, the compiler may perform instruction scheduling before or after register allocation (or may do scheduling twice). Some optimizations, such as dead code elimination, may be repeated several times. A compiler with a choice of many transformations to apply must also determine an order in which to apply them. For parallel computer systems, a fixed order may not be sufficient. Choosing the best sequence of transformations is a difficult problem, and depends on the characteristics of the program as well as on the target machine. This problem is addressed by Wolfe (1987, 1988a) by using a simple
machine model. Brandes and Sommer (1987a, 1987b) use a rule-based interactive parallelizing programming environment; the user can write rules to capture the program characteristics that affect performance.

A subproblem is the loop mode assignment—the choice of which loop or loops to execute in parallel, vector, or sequential mode. Koblenz and Noyce (1989) describe a mechanism to solve the loop mode assignment problem using a tree representation of the potential ways to parallelize the loops; additional cost analysis determines which of these generates the best performance. Hyler et al. (1988) discuss the compiler option of generating multiple versions of a loop, and several means of choosing between the two versions.

Choosing the best transformation to apply next requires the compiler to be able to predict the performance of the target system on the transformed code, at least relative to the untransformed code; one such scheme is shown for the Cyber 205 vector supercomputer by Arnold (1983). A performance prediction scheme for the Alliant FX/8, a parallel vector multiprocessor, is described by Gallivan et al. (1989), assuming that the assembly code is available. Atapattu and Gannon (1989) describe a model of the memory subsystem for the Alliant machine. Fahringer (1994) describes a portable performance predictor that uses the results of a set of “training runs” to learn what performance parameters are important; the parameters estimate work distribution, communication overhead, and data locality. Appelbe and Smith (1992) discuss improving parallelism by using variable replication and loop alignment.

Several schemes try to avoid the “one transformation at a time” paradigm. Lamport (1974) describes a wavefront scheme essentially equivalent to a unimodular transformation of the index set. Dowling (1990) shows that such an approach can always move all the loop carried dependences to the outer loop, making all inner loops parallel. Wolf and Lam (1991a) use unimodular transformations to convert the loop to a fully permutable form, then find coarse-grain (outer loop) or fine-grain (inner loop) parallelism; their method, described more fully by Wolf (1992), includes tiling for locality. Pugh (1991a) describes a scheme to find a schedule for each statement in a loop, then interleaves the schedules for each statement to get the schedule for the whole loop. Ayguadé and Torres (1993) describe a similar, but simpler, scheme.

Several interactive parallelizing tools have been developed. Smith and Appelbe (1989) and Appelbe, Smith, and McDowell (1989) describe Start/Pat, a parallelizing assistant tool, that interactively converts sequential programs to use Cray microtasking directives or other parallel Fortran dialects. Callahan et al. (1988) and Kennedy, McKinley, and Tseng (1991a) describe ParaScope, which contains a rich set of loop transformations. Hall et al. (1993b) describe experiences using this tool.

Allen et al. (1988b) describe how to use the control dependence relation to determine maximal parallelism in a program; the data dependence relations add constraints that are satisfied either explicitly (with synchronization) or implicitly (by scheduling).

Various heuristics have been developed to schedule task graphs (a la Petri nets) onto parallel processors; some of these have been adapted for use in general-purpose compilers for parallel computers, such as described by Girkar (1988). Watts, Soß, and Gupta (1992) use transformations such as loop fusion to create tasks, then construct and schedule the resulting task graph. Polychronopoulos (1988b) proposes a scheme whereby the compiler partitions the program into parallel tasks and generates drive code to schedule the tasks.
onto the processors without operating system intervention or overhead. Polychronopoulos (1991) discusses how to construct task graphs from the data and control dependence relations of procedure and loop bodies, and how to generate code that will unfold the parallelism dynamically; Girkar and Polychronopoulos (1991) show how to optimize the generated code in this model.

Burke et al. (1988) shows how PTRAN extracts parallelism based on the control dependence graph. Kasahara et al. (1991) describe a scheme to extract parallelism at several granularities; large grain macrotasks are scheduled onto processor clusters, and iterations of parallel loops in a macrotask are scheduled onto processors in a cluster. Polychronopoulos and Banerjee (1987) discuss processor allocation to maximize speedup when large-grain outer loop parallelism and fine-grain inner loop parallelism are both available.

Privatization or scalar expansion can be used to eliminate false dependence relations (anti- and output dependence). If there are no loop carried flow dependence relations, privatization is feasible; if there are loop carried flow dependence relations, scalar expansion must insert the correct subscripts into the array so that the fetch gets the proper value. Either transformation is relatively simple, once the def-use information is available. To privatize or expand arrays, value-based dependence information must be available. Array privatization is feasible when there are no loop carried flow dependence relations for that array. Array expansion requires the compiler to determine the last assignment to each array element, so that the proper subscript expression can be inserted in the expanded dimensions. This is discussed in detail by Feautrier (1988b). Data-flow analysis techniques to determine when arrays can be privatized are given by Li (1992) and Tu and Padua (1993). A run-time technique to test for situations when privatization will allow execution of a loop as a doall is discussed by Rauchwerger and Padua (1994).

The problem of finding parallelism in sequential code is simply a matter of inspecting the dependence relations. While restructuring the program may allow additional parallelism, executing the remaining nonparallel parts of the program sequentially retains program correctness. The inverse problem is generating sequential code from a parallel program, which is discussed by Ferrante and Mace (1985) and Ferrante, Mace, and Simons (1988). Sarkar and Cann (1990) discuss compiler methods for a single assignment language, where the goal is to restrict the language parallelism to that which is useful in the target machine.

A related problem is automatically determining when a parallel program is correct. One aspect of this is finding race conditions in a parallel program, when two tasks that are executing in parallel can access the same location (and one of the accesses is a write to that location). This is studied by Balasundaram and Kennedy (1989a). Emrath, Ghosh, and Padua (1989) and Callahan, Kennedy, and Subhlok (1993) also study this for parallel programs with certain types of synchronization primitives. Schonberg (1989) and Mellor-Crummey (1991) describe methods to monitor parallel programs to detect data races dynamically. Netzer and Miller (1990) discuss a scheme to distinguish between between actual data races, data races that could have occurred but didn’t, and apparent data races. Netzer and Miller (1992) show how to filter out false races to focus attention on actual problems. Dinning and Schonberg (1990) discuss several dynamic race detection methods for effectiveness and efficiency. LeBlanc and Mellor-Crummey (1987) show a scheme to reproduce the execution behavior of a parallel program, allowing deterministic debugging.

Chen and Yew (1991) argue that restricting parallel loops to those without
loop carried dependence relations reduces parallel performance to unacceptable levels. Scheduling policies for parallel loops that contain synchronization operations can affect performance, as discussed by Tang, Yew, and Zhu (1988). Krothapalli and Sadayappan (1991) discuss methods to remove redundant synchronizations from loops.

All-pairs computations are studied in the framework of linear systolic arrays by Shih, Chen, and Lee (1987). Efficient schedules for multiprocessors are discussed by Theobald and Gao (1991).

Cytron (1984) discusses synchronization and speedup for doacross loops. Midkiff and Padua (1986, 1987) discuss adding synchronization in parallelized loops. Wolfe (1988b) describes several forms of synchronization. Allen and Kennedy (1985) describes general parallelization techniques, including methods to reduce synchronization. Allen et al. (1986) show an interactive tool, PTOOL, to aid the user in determining when synchronization is necessary: Henderson et al. (1990) describe their experiences using PTOOL including some limitations of static analysis. Sarkar (1988) discusses minimizing the number of synchronization operations by using counting versus binary semaphores. Li (1991) discusses the problem of adding post and wait operations for event-variable synchronization, a generalization of the advance and await operations discussed here. Insertion of data synchronization operations is discussed by Zhu and Yew (1987) and Tang, Yew, and Zhu (1990), and an optimization to reduce the number of synchronizations required is presented by Li and Abu-Sufah (1987). Su and Yew (1989) argue that data synchronization is too costly unless the number of processors is large. Tzen and Ni (1993b) show how to find a set of uniform synchronization primitives to satisfy dependence relations with irregular distance in doubly nested loops.

When parallelizing sequential loops, finding special operations, such as reductions, is important. Generalized reduction operations are discussed by Jouvelot and Dehbonei (1989).


Venugopal and Eventoff (1991) discuss using strip mining or tiling to reduce interprocessor cache conflicts. Matsumoto and Hiraki (1993) explore dynamically changing the cache coherence protocols for particular data sets depending on the sharing patterns; for instance, switching between write-update and write-exclusive protocols can be done depending on whether or not the data is shared read-write.

Jalby and Meier (1986) first discussed compiling nested loops into tiles for locality in multiprocessor caches. Wolfe (1989c) discusses the use of iteration space tiles as the unit of parallel execution, while optimizing for locality within tiles. Kennedy and McKinley (1992) show a scheme to find a loop ordering that optimizes locality and parallelism; if the ordering is not feasible, a fall-back position tries to find a “nearby” loop ordering with most of the benefits. Hudak
and Abraham (1990) discuss tiling parallel loops to minimize interprocessor communication. Ramanujam and Sadayappan (1991b) describe how to find extreme vectors to allow iteration space tiling to generate atomic parallel tiles. Ramanujam and Sadayappan (1992) present methods to find extreme vectors in n-dimensional iteration spaces. Boulet et al. (1994) discuss scalable criteria for choosing the best tile shape to reduce interprocessor communication.

Kulkarni et al. (1991) use unimodular transformations to find an outer loop to execute in parallel; this will minimize the number of communication operations, the communication volume, and the load imbalance.

Sarkar and Gao (1991) and Gao et al. (1992) show a scheme to optimize a set of adjacent loops via interchanging, reversal, and fusion, in order to enable array contraction; in the general case, an array may be replaced by a small buffer of values, if the dependence distance is small enough. Kennedy and McKinley (1993) discuss the best way to use loop fusion and fusion to generate the most parallelism (and the fewest sequential loops), and also discuss data reuse.

Girkar and Polychronopoulos (1988) describe a mechanism to execute the body of one loop in parallel with the body of an adjacent loop (or n adjacent loops, in general), which might be useful when loop carried dependence relations prevent execution of the loop iterations in parallel.

Collard (1994) discusses parallelizing a while loop that surrounds one or more nested iterative loops by using linear transformations and speculative execution.

Cytro, Lipkis, and Schonberg (1990) describe rules for determining when sequential code can be redundantly executed on multiple processors, rather than parking the slaves and letting only the master execute the sequential code.

Hatcher et al. (1991) describe generation of parallel code from Dataparallel C, a SIMD-parallel language for a variety of parallel architectures.

Bik and Wijshoff (1993a, 1993b) describe methods to compile operations on sparse matrices; they also describe (1993c) methods to optimize the parallelism in these algorithms by taking advantage of the sparsity to eliminate unnecessary synchronization.

Cytro, Hind, and Hsieh (1989) describe compiler algorithms to find nonloop parallelism, a la fork-join or cobegin-coend, but allowing a general DAG precedence graph between the parallel tasks. Burke et al. (1989) use this along with loop parallelization to identify parallelism at several levels of granularity. Girkar and Polychronopoulos (1992) use a hierarchical representation of tasks and dependence relations to find nonloop parallelism. Kasahara et al. (1990) describe a compiler that breaks the program into macro-tasks, and dynamically schedules these onto processor clusters.

Pfister and Norton (1985) and Kruskal, Rudolph, and Snir (1988) discuss combining networks to improve the performance of synchronization operations on large-scale multiprocessors. Goodman, Vernon, and Woest (1989) describe hardware-supported scalable synchronization schemes, including support for fetch-and-operation primitives in software. Gupta (1989) introduces the fuzzy barrier synchronization primitive, which allows a processor to continue to perform useful work after signaling that it has arrived at the barrier but before checking to see if all other processors have arrived. Anderson (1990) discusses methods to improve the performance of the low-level synchronization primitives (spin-waiting) typically used in shared-memory multiprocessors. Graunke and Thakkar (1990) also discuss the performance of various synchronization algorithms.
Some machine designs directly support higher level parallel primitives, such as the parallel loop control used in the Myrias computer system (Beltrametti, Bobey, and Zorbas [1988]); taking these into account can either simplify or complicate the job of the compiler. The Tera computer system (Alverson et al. [1990]) supports parallelism at many granularities (Alverson et al. [1992]).

Delivering the performance for parallel applications often requires tuning the program, which really means manual intervention. Performance analysis tools, such as Mtool (Goldberg and Hennessy [1993]) can help find critical parts of the program that need tuning.

An alternative approach to automatic parallelization is to use pattern matching, as described by Keller and Paul (1993); such a scheme can restructure the program in ways that seem to violate data dependence relations since the patterns can incorporate semantic information. A portable yet effective parallelizer for Pascal is described by Gabber, Averbuch, and Yehudai (1991).

Sequential consistency was introduced by Lamport (1979). Weaker memory consistency models are discussed by Adve and Hill (1990). Release consistency is introduced by Gharachorloo et al. (1990) and is used in the Stanford DASH prototype multiprocessor.

12 Vector Analysis

Vectorization was the earliest method of automatically finding parallelism in sequential loops (Schneck [1972]). The Parafrase project at the University of Illinois (Kuck et al. [1980]) and the PFC project at Rice University (Allen [1987]) were two of the earliest projects to use data dependence theory to find vector-style parallelism from sequential loops.

Allen and Kennedy (1992) discuss strip mining and optimizations to enhance vector register locality.

Callahan, Dongarra, and Levine (1988) describe a small benchmark of 100 short loops that were used to test the automatic vectorization capabilities of vectorizing compilers in 1988; this benchmark was used for several years thereafter as a marketing tool by the vendors.

As mentioned earlier, choosing an order for the transformations is difficult. Bose (1988) describes an interactive aid for program development and optimization, aimed at the IBM 3090 Vector Facility.

The influence of optimizing for stride on a cached vector machine is described by Erbacci, Parno, and Tagliavini (1989).

Tanaka et al. (1988, 1990) use loop unrolling for loops with recurrences on a vector computer with hardware support for first-order linear recurrences to reduce the recurrence length and improve the execution time.

Tsuda and Kunieda (1988, 1990) describe a vectorizing compiler that makes extensive use of indirect addressing after loop coalescing to vectorize nested loops. Tsuda, Kunieda, and Atipas (1989) show the results of applying vectorizing compiler technology to character string and database join operations.

The scheme presented here for vectorizing loops with conditionals where the condition is in a dependence cycle is taken from Banerjee and Gajski (1984).

13 Message Passing Machines

Early message-passing multicomputers are summarized by Athas and Seitz (1988).

The owner-computes rule was first described by Rogers and Pingali (1989, 1994) and Pingali and Rogers (1990), who implemented one of the first global compilers for a commercial message-passing machine. Koelbel, Mehrotra, and Rosendale (1987a, 1987b) addressed simple nearest-neighbor problems in the context of a simplified Pascal-like language, Blaze. Handling nearest-neighbor communication for distributed data structures is sufficient for some applications (Koelbel, Mehrotra, and Rosendale [1990]). More early work is described by Koelbel and Mehrotra (1991b, 1991a) using Kal, a language based on Fortran. Mehrotra and Rosendale (1990) discuss Kal language constructs and the inspector/executor model used by the compiler. Koelbel et al. (1990, 1991) discuss automatic insertion of communication operations, and the resulting performance, for irregular and regular patterns. Gupta and Banerjee (1992) show how to generate collective communication routines by analyzing the data movement required. Callahan and Kennedy (1988b) describe the design of a general scheme for code generation for distributed memory. Rühl and Annaratone (1990) describe a scheme to compile annotated Fortran to a research message-passing system. André, Pazat, and Thomas (1990a, 1990b) discuss Pandore, a compiler that uses a model of distributed execution similar to that of High Performance Fortran. Philippsen and Tichy (1991) describe the compiler for Modula-2*, another language for writing structured highly parallel programs for shared or message-passing computer systems. Merlin (1991) describes the ADAPT system, which converts Fortran 90 array assignments automatically into message-passing code. Mehrotra and Rosendale (1990) discuss communication patterns and windows during which the communication must take place. Chen and Hu (1992) show how to generate parallel code for several categories of loops. Gupta and Schonberg (1993) describe analysis to minimize communication by reusing data that has already been sent, either to the same processor or to a processor nearer than the owner.

Chatterjee et al. (1993b) describe a method using finite-state machines to generate the local addresses of distributed data. Hiranandani et al. (1994) discuss this as well as other problems of compiling with general block-cyclic distributions. The scheme described here was first presented by Gupta et al. (1993).

One of the early SIMD machines was the Massively Parallel Processor, or MPP, installed in May 1983 at Goddard Space Flight Center; the machine and its software are described in the collection edited by Potter (1985). The MPP and other SIMD machines are summarized by Hord (1990).

Thinking Machines Corp. delivered one of the first compilers that allowed programs to be written in data-parallel style without writing node programs; the
Connection Machine Fortran compiler used extended Fortran 90 array assignments by including a `forall` statement, from which the compiler would generate parallel code (Albert, Lukas, and Steele [1990, 1991]). The early Thinking Machines compilers used the concept of virtual processors, supported by microcode, for large data sets. Christy (1991) argues that the compiler should deal with actual processors, not virtual processors, and this is how the compilers for the Thinking Machines Connection Machine evolved. Sabot (1992) discusses how the compiler eventually was migrated to support the MIMD Connection Machine CM-5. Bromley et al. (1991) describe compiler optimizations to generate code for the SIMD Connection Machine CM-2 that could run at 10 gigaflops; the programs were restricted to have regular stencil data accesses, and the compiler used overlap regions to reduce communication, and cleverly scheduled the PE code to take maximum advantage of the registers, another example of how optimizing for parallelism is not enough.

The design of High Performance Fortran borrowed much from other languages. The `template`, `align` and `distributed` statements are borrowed from Fortran D, a language and compiler developed at Rice University (Hiranandani, Kennedy, and Tseng [1992b]). Hiranandani et al. (1991a, 1991) and Tseng (1993) describe the design of the analysis and optimizations used in the Fortran D compiler. Hall et al. (1992) show how the Fortran D compiler uses interprocedural analysis to preserve the benefits of separate compilation; it also calculates reaching layouts for procedure arguments, clones procedures based on the layouts of their formal arguments, and computes the sizes of the overlap regions based on interprocedural information. Hiranandani, Kennedy, and Tseng (1993) describe some early experiences with the Fortran D compiler, showing that for some types of problems, it approaches the performance of hand-optimized parallel code, while for other cases more optimizations need to be integrated. Hiranandani, Kennedy, and Tseng (1992a, 1994) show the relative enhancements due to particular optimizations in the Fortran D system, in particular the communication optimizations. Bezukus et al. (1993a, 1993b, 1994) describe related work on a compiler based on Fortran 90 array assignments, and Choudhary et al. (1992, 1993) describe how this compiler is integrated with the parallelizing Fortran D system.

Vienna Fortran, described by Chapman, Mehrotra, and Zima (1992), is related to High Performance Fortran, but includes some features not yet addressed by that language. Fahringer, Blasco, and Zima (1992, 1993) describe a performance prediction tool for the Vienna Fortran Compilation System, a compiler that generates message-passing code; the performance prediction tool is used both by the compiler system and by users to help with program transformation and data layout decisions. Chapman et al. (1993) describe the support of Vienna Fortran for dynamic distributions, including some of the compiler analysis necessary to optimize it. Chapman, Zima, and Mehrotra (1992) discuss the procedure interface for distributed data structures, including inheritance of the data layout and conditionals to dynamically test the actual data layout. Chapman, Mehrotra, and Zima (1993) describe an approach for data distribution used in Vienna Fortran that avoids the `template` construct. An extension to include Fortran 90 constructs is described by Benkner, Chapman, and Zima (1992).

Baber (1991) and Otto and Wolfe (1992) describe early precompilers aimed at many of the goals of High Performance Fortran. The programming model of these precompilers is that the user writes a node program using global indexing.
the node program is executed on each node, and parallel work is distributed over the nodes according to the array distribution. Another language design is given by Flinck and Kudome et al. (1992). Kudome et al. (1990) discuss ASPAR, a simple parallelizer for C programs targeted at message-passing machines. Nichols, Siegel, and Dietz (1990, 1993) describe the compiler for FLP, an explicitly parallel language, and show a way to handle mono and poly data for SIMD or SPMD execution. Socha (1990) discusses a method for handling a class of data-parallel algorithms based on grids of data points. Chase et al. (1992) describe Paragon, a language and programming methodology for writing architecture-independent data-parallel applications. Grim (1990) describes an implementation of SISAL for a distributed memory machine.

Tseng (1990b, 1990a) shows compilation of loops for a simple array language onto a systolic array, essentially a message-passing machine with high bandwidth interconnections. Chen and Cowie (1992) describe the optimizations, including transformations of the parallel code, in a prototype Fortran 90 subset compiler for the SIMD CM/2 and MIMD CM/5 by Thinking Machines. Bala, Ferrante, and Carter (1993) give a compiler intermediate representation to facilitate optimization and communication operations.

Ponnusamy, Saltz, and Choudhary (1993) discuss ways to handle problems that are caused by mapping an irregular data structure onto a linear array; in particular, they discuss ways to interface a logical data partitioner to the compiler-generated code. They also discuss scheduling each iteration on a single processor (the owner of the most left-hand sides), rather than scheduling each statement of each iteration on the owner of its left-hand side. Agrawal, Sussman, and Saltz (1993) discuss methods to integrate support into a compiler for multiple communicating meshes, such as multigrid or coupled meshes. Data-flow analysis is used to insert inspector/executor communication for indirect array accesses by von Hanxleden et al. (1992). Das, Saltz, and von Hanxleden (1993) discuss corresponding methods to use when there are multiple levels of indirection. Berryman, Saltz, and Scroggs (1991) describe a set of subprograms to support and optimize communication corresponding to unstructured meshes. Hiranandani et al. (1991b) describe the performance of a scheme that uses hash tables to access local copies of off-processor data.

Kung and Subhlok (1991) show the importance of block-cyclic data distributions for matrix operations. The divide operations needed for implementation of block-cyclic distributions can be done with multiplications, as shown by Granlund and Montgomery (1994).

Gallivan, Jalby, and Gannon (1988) discuss options for solving the problem of finding and transferring subregions of distributed arrays that are accessed locally. Amarasinghe and Lam (1993) describe more aggressive communication optimizations based on exact data dependence analysis.

Kennedy and Roth (1994) discuss loop fusion and index set splitting to optimize SIMD context computation. Weiss (1991) discusses generalized strip mining for SIMD machines to avoid global communication. The SIMD constraint that all processors must execute the same instruction does not necessarily mean that all processors must proceed through their local iteration sets in the same order; von Hanxleden and Kennedy (1992) describe compiler transformations to improve the asymptotic performance of SIMD machines. Several optimizations for SIMD machines are summarized by Knohe, Lukas, and Weiss (1993).

An alternative approach to compiling for a message-passing machine is to treat it more like a shared-memory machine; Rudolph and Polychronopoulos
(1989) describe an experimental system where iterations are scheduled onto the nodes of a message-passing system by guided self-scheduling, rather than by looking at the data layout. Su and Yew (1991) use a similar scheme using static scheduling, allowing flow dependence relations with constant distance to be resolved by direct messages from the processor that produces the value to the memory of the processor that will execute the iteration that consumes the value. Wu and Gajski (1988) show another programming scheme where the compiler generates a static schedule and inserts synchronization given the communication structure of the tasks.

Several research groups have looked at addressing the compilation problem by partitioning the iteration space (through tiling or other mappings) and using that to generate data partitions. Shen and Tai (1991) show a method using linear transformations. Ramanujam and Sadayappan (1990a, 1990b, 1991a) study how to partition the iteration space to find communication-free partitions, or, failing that, use extreme vectors to partition the tiles into atomic units.

Automatic data alignment and distribution are subjects of current research. An optimal solution in a restricted domain—that of computation of a single expression in a simple Cartesian processor space with one element per processor—is shown by Gilbert and Schreiber (1991). Ramanujam and Sadayappan (1989) discuss constructing a data space dependence graph in addition to the iteration space dependence graph to help with automatic data partitioning. Li and Chen (1990b, 1991b) describe the axis alignment graph and methods to partition it. Li and Chen (1990c, 1991a) show a scheme where the data is automatically aligned using an axis alignment graph approach, where collective communication patterns are recognized, and where the choices of which dimensions to distribute and how are made based on the costs of the communication for each pattern.

This work is based on a functional language, Crystal, but the techniques can be used in imperative languages (Li and Chen [1990a], Li [1991]). Knobe, Lukas, and Steele (1988, 1990) and Knobe and Natarajan (1990) describe a similar scheme using alignment preferences specifically targeted at SIMD machines. Balasundaram et al. (1990, 1991) propose an interactive tool that will guide the programmer in making data distribution decisions based on a compile-time performance estimator, where the estimator module is trained with a set of kernel routines. O'Boyle and Hedayat (1992) describe a scheme for automatic data alignment for affine array accesses by finding the Hermite form of the access matrix. Wholey (1992) uses a language with high-level primitive operations and describes a scheme to decide how to allocate machine dimensions to distributed array dimensions. Anderson and Lam (1993) try to automatically discover parallelism and an appropriate data layout by distinguishing arrays that can be simply decomposed from those that require communication or reorganization; code generation and related issues are presented by Amarasinghe et al. (1993). Gupta and Banerjee (1991, 1992a, 1993) use an axis alignment graph to align arrays, followed by analysis of how to distribute each aligned set of dimensions. Feautrier (1993) uses exact value-based dependence relations and finds a data layout that maximizes the number of dependence edges that are satisfied by scheduling the dependent iterations on the same processor. Chapman, Herbeck, and Zima (1991, 1993) discuss support for automatic data distribution in Vienna Fortran, which uses intraprocedural and interprocedural analysis allowing dynamic realignment. Efficient and effective solutions for restricted cases are presented by Sinharoy and Szymanski (1994). Chatterjee, Gilbert, and Schreiber (1993b) discuss mobile alignments, in particular when
data should be realigned between iterations of a loop; Chatterjee, Gilbert, and Schreiber (1993a, 1993a) describe the alignment-distribution graph, a graphical form of the flow of data used for automatic alignment. Snyder and Socha (1990) describe a scheme for finding balanced almost-rectangular data partitions.

Sabot and Wholey (1993) describe CMAX, a tool that automatically converts sequential programs to parallel programs for the Connection Machine; much of the analysis has to do with data layout and avoiding problems that arise from Fortran’s storage and sequence association.

Burns, Kuhn, and Wernme (1992) describe a low-copy message-passing scheme that eliminates much of the software and system overhead associated with message-passing systems; they note that additional synchronization between the system and the application may need to take place; for instance, to insure that the buffer is filled before using or was emptied before reusing. Hsu and Banerjee (1990) discuss the design of a coprocessor that will remove most of the cost of message-passing from the software. Active messages are presented by von Ficken et al. (1992) as a method to reduce the cost of communication by associating a software handler with each message. Thakur, Choudhary, and Fox (1994) discuss details of optimizing the communication that arises from redistributing distributed arrays. Holm, Lain, and Banerjee (1994) discuss code generation strategies to change a SPMD program, such as one resulting from the methods in this chapter, into code suitable for a multithreading or message-driven environment.

Saltz et al. (1990, 1991) propose that the communication latency is so high for message passing that the compiler should leave most of the scheduling and communication decisions until run-time; the argument is that the extra cost at run-time will pay off in lower communication time. Early development of this idea is described by Mirchandaney et al. (1988).

14 Scalable Shared Memory Machines

Markatos and LeBlanc (1991, 1992, 1994) show that the potential load imbalance of using affinity scheduling is outweighed by the improved cache locality, even with small numbers of processors, compared to guided self-scheduling, factoring and other scheduling methods. Squirilante and Lazowska (1993) describe several scheduling policies and explore how they are affected by cache locality. Appelbe and Lakshmanan (1993, 1993) discuss the interaction of loop transformations with affinity scheduling. Li and Pingali (1992b) use loop restructuring to modify the loop so that array accesses in the inner loop have improved locality.

Cache memories and the three Cs classification of cache misses are described by Hennessy and Patterson (1990). Scalable distributed directory cache coherence hardware is described by Thapar and Delagi (1990). Several directory cache coherence protocols are summarized by Chaiken et al. (1990). Weber and Gupta (1989) argue that a directory scheme need only have a few pointers for each cache line, if the applications are appropriately well-written. Gupta, Weber, and Mowry (1990) show several directory schemes and evaluate them according to the directory storage and network traffic requirements.

Compiler-managed cache memories for large-scale multiprocessors have been heavily studied. Cytron, Karlovsky, and McAuliffe (1988) use data-flow analysis to find where to put cache control instructions. Owicki and Agarwal (1989) eval-
valuate the performance of two simple software cache coherence schemes, finding them effective for large scalable systems. Videnbaum (1986) studies a software-cache coherence scheme with three cache control instructions: invalidate the cache, turn the cache on, and turn the cache off. Cheong and Videnbaum (1987) evaluate a simple write-through cache that is kept coherent by flushing the cache before and after each parallel loop. This has the unfortunate effect of losing locality between adjacent parallel loops. In a modification to this approach described by Cheong and Videnbaum (1990b), the software and the cache maintain a version number for each variable; when an access to a variable hits in the cache but that variable has an old version number, the cache value is stale and the cache line must be reloaded. Since the granularity of the version numbers is one per variable, even if that variable is an array, there is still the potential for lots of unnecessary cache traffic. Several of these schemes are summarized by Cheong and Videnbaum (1990a). A similar scheme is described by Min and Baer (1989, 1992). Min and Baer (1990) show through simulations that software-based cache coherence can achieve cache hit rates equivalent to directory-based methods with less network traffic.

Chen and Videnbaum (1991a) describe a modified directory scheme to hold the sharing state of each cache line. In a subsequent paper Chen and Videnbaum (1991b) compare simulated performance of software- and hardware-managed cache memories; the software cache was better at tolerating false sharing for writes. Cheong (1992) proposes a modified cache coherence protocol, with up-to-date and soon-to-be stale cache states identified by the software and kept in the cache. Darnell, Mellow-Crummy, and Kennedy (1992) discuss a scheme where post and invalidate instructions are inserted by the compiler for each memory reference and floated out of loops using vectorization techniques. Darnell and Kennedy (1993) discuss a hardware-assisted software cache coherence scheme, and show that it can achieve a hit ratio close to a hardware snoopy cache coherence protocol; this paper presents a nice summary of various software cache coherence schemes. Lilja and Yew (1993) show compiler optimizations to reduce the memory used by software cache coherence. Cheong and Videnbaum (1988) describe flow analysis methods to minimize the size of the cached data that must be invalidated as stale.

Larus (1993) argues that the flexibility of compiler-managed caching can outweigh the performance advantages of hardware-managed caching. Adev et al. (1991) present experiments comparing hardware and software cache coherence, showing that software schemes are comparable to hardware schemes. Bennett, Carter, and Zwaenepoel (1990) present an adaptive software cache coherence strategy. Li and Sevcik (1994) show the performance of using software cache coherence with affinity scheduling.

Picano, Brooks, and Hong (1991) argue the importance of alleviating the programmer from dealing with cache coherence. Ju and Dietz (1991) show an approach to optimize the program using loop transformations and changing the data layout to reduce cache coherence traffic.

Hudak and Abraham (1991) describe a scheme to partition elements of global arrays into four classes: the exclusive read-write (ERW) set, the shared read-exclusive write (SRFW) set, the shared read-no write (SRNW) set, and the no read-write (NRW) set. Consistency needs to be maintained for the SRFW set, but the ERW can be cached without penalty for interprocessor consistencies, and the SRNW set can be cached or duplicated. They also show through experiments on the BBN TC2000 that recognizing these sets can improve per-
formance significantly. Lilja and Yew (1991) and Monnes-Toussi, Lilja, and Li (1994) discuss using compiler algorithms to optimize the use of directory-based cache memories by determining which data locations or references might need coherence operations; read-only or private data need not suffer the penalty of coherence operations or even directory pointers. Abraham and Hudak (1991) use rectangular and hexagonal tiling to increase the cache locality of programs containing parallel loops nested in a sequential loop.

An alternative design for NUMA machines is to require the software to explicitly load the local cache, rather than using hardware cache miss detection. Granston and Vaidenbaum (1991) discuss a method to detect redundant cache loads, to reduce the memory traffic. Lee, Yew, and Lawrie (1987) show a method that inserts prefetch instructions to alleviate cache latencies. Lilja (1994) notes that prefetching requires the program to know what iterations will be executed on a processor, affecting the scheduling strategy.

Culler et al. (1991) describe a scheme based on multithreading at the software level, which allows a processor to tolerate long latencies for messages or synchronization.

Abdelrahman and Wong (1994) use data placement and work distribution methods essentially equivalent to the techniques used in the preceding chapter for message-passing machines to manage locality on a NUMA machine; in their target machine, the local memory at each processor is part of the global shared-memory.

Smith et al. (1987) describe the ZS-1, the first commercial machine with decoupled access and execute processors. Bird, Rawstorne, and Topham (1993) discusses the effectiveness of a compiler for such a machine, with an additional level of decoupling, adding a control processor.

Some NUMA systems are constructed from standard workstation-class components, with the operating system detecting and handling page faults for pages that are resident on another node. Li and Hudak (1989) discuss operating system methods to resolve page faults and coherence in such a system. Granston (1993) describes compiler analysis and optimizations to reduce page faults in such an environment.

References


References


References


References


References


Thapar, Manu, and Bruce Delagi (1990). Scalable Cache Coherence for Large Shared Memory Multiprocessors. In CONPAR 90 - VAPP IV: Joint International Conference on Vector and Parallel Processing, held in Zurich, Switzerland, September, number 634 in Lecture Notes in Computer Science, Berlin: Springer Verlag, 592–603.


References


